Jül-4354

Institute for Advanced Simulation (IAS)
Jülich Supercomputing Centre (JSC)

Design and implementation of a highly configurable and efficient simulator for job schedulers on supercomputers

Carsten Karbach

JÜLICH
FORSCHUNGSZENTRUM

Design and implementation of a highly configurable and efficient simulator for job schedulers on supercomputers

Carsten Karbach

Berichte des Forschungszentrums Jülich; 4354
ISSN 0944-2952
Institute for Advanced Simulation (IAS)
Jülich Supercomputing Centre (JSC)
Jül-4354

Vollständig frei verfügbar im Internet auf dem Jülicher Open Access Server (JUWEL)
unter http://www.fz-juelich.de/zb/juwel

Zu beziehen durch: Forschungszentrum Jülich GmbH · Zentralbibliothek, Verlag
D-52425 Jülich · Bundesrepublik Deutschland
☎ 02461 61-5220 · Telefax: 02461 61-6103 · e-mail: zb-publikation@fz-juelich.de

Contents

Abstract

The supercomputers maintained by the Jülich Supercomputing Centre (JSC) are used by scientists for simulation projects in computational science and engineering. These systems are run in batch mode and are shared among all active users. Jobs are submitted to the job scheduler, which decides for each job the time of execution. It manages running and waiting jobs, sorts them by a given priority and executes the jobs by fitting them into the current system state. For users it is often difficult to understand the job scheduler's algorithm. Thus, a prediction of the future system allocation of currently running and queued jobs is valuable. It helps users to plan job submissions and supports administrators by optimising the system load.

This master's thesis deals with the design and implementation of a configurable simulation for various global job schedulers. The developed simulation program called JuFo focuses on the job schedulers Loadleveler and Moab as these are the most important batch systems for the supercomputers provided by the JSC. A major design goal is to keep the simulation independent from special job schedulers, which is achieved by the generic configuration of JuFo. Since the simulation of a job scheduler with hundreds of jobs can become time consuming, methods for optimising this process are investigated. Finally, a test framework is developed, which allows for evaluating the accuracy of the schedules provided by JuFo.

Zusammenfassung

Die vom Jülich Supercomputing Centre (JSC) betriebenen Parallelrechner werden von Wissenschaftlern für Simulationen verschiedenster Anwendungsbereiche verwendet. Diese Rechner laufen im Batch-Betrieb und können gleichzeitig von allen aktiven Benutzern angesprochen werden. Sie schicken Jobs an den Scheduler, welcher über den Ausführungszeitpunkt aller Jobs entscheidet. Der Job Scheduler verwaltet laufende als auch wartende Jobs, sortiert sie nach gegebener Priorität und lässt passende Jobs ausführen. Für die Nutzer ist es oft schwierig den Algorithmus zur Platzierung der Jobs nachzuvollziehen. Deshalb wird ein Simulationsprogramm zur Vorhersage des Job Schedulers benötigt. Dieses unterstützt nicht nur die Anwender bei der Planung zukünftiger Jobs, sondern hilft auch Administratoren bei der Optimierung des Schedulers.

Diese Masterarbeit behandelt den Entwurf und die Implementierung eines flexibel konfigurierbaren Simulationsprogramms für verschiedene globale Job Scheduler. Der Entwurf des Simulationsprogramms mit dem Namen JuFo basiert auf der Analyse der Scheduler Loadleveler und Moab, welche für die Supercomputer des JSC eingesetzt werden. Ziel dieser Analyse ist, dass JuFo basierend auf möglichst flexibler Konfiguration für beliebige Zielsysteme eingesetzt werden kann. Da die Simulation hunderter von Jobs zeitaufwendig werden kann, werden Möglichkeiten zur Optimierung erarbeitet. Schließlich wird eine Testumgebung entwickelt, welche es erlaubt die Genauigkeit der Vorhersage zu beurteilen.

Chapter 1

Introduction

The importance of supercomputers like those provided by the Jülich Supercomputing Centre (JSC) is steadily increasing, as researches require more and more computing time for their simulations. The demand for computing time on parallel systems exceeds the capacities of computing centres. Thus, the use of this rare resource of computing time has to be optimized. On the one hand, this can be accomplished by optimizing the algorithms. This includes the reduction of the parallel overhead, which mainly consists of communication, start-up and idling times of the involved processors. On the other hand, administrators of the supercomputers have various options for minimizing idling times of the computing resources. The choice and configuration of the operating system for the supercomputers strongly influences their efficiency.

A very important part of the supercomputer's operating system is the job scheduler. This scheduling system is responsible for efficiently mapping submitted jobs to the available computing resources. Krallmann et al. [1] define the scheduling system as follows:

> "The scheduling system of a multiprocessor receives a stream of job submission data and produces a valid schedule." [1]

Optimizing the job scheduler and fitting it to the specific needs of a given system is one of the major tasks for the administration of a supercomputer. Often it is difficult to test changes in the configuration of the scheduling system. Thus, a configurable simulation of the job scheduler would be a valuable tool to support the administration work. Hence, the topic of this master's thesis is to develop the basis for simulating job schedulers. The simulation will be used for predicting the future schedule of the specific system based on real time data gathered from the simulated system. The most important design goal for this simulation program is to keep it generic and configurable. This will allow to adapt the simulation to different system types without the need of reimplementation for each new supercomputer.

Next to the support of the supercomputer's administration, the simulation is also useful for users by providing a prediction of the supercomputer's schedule. Based on the currently queued and running jobs the simulation could estimate the dispatch time for the submitted jobs. This helps the user to plan further job submissions. The simulation predicts idling

times for the computing resources, which then can be filled with matching jobs by the users. Both use-cases will lead to higher efficiency and load for the specific system.

1.1 Problem definition

The task for this thesis is to develop a generic simulation program for the job scheduler of massively parallel systems. These systems are run in batch mode, which means that the users submit job descriptions to the system. The jobs are executed by the supercomputer at a time determined by the job scheduler. It is assumed that the analysed batch systems reserve the requested resources for each job until their completion. Jobs are not suspended or migrated during their execution as context switches can become very time consuming for highly scalable jobs. The simulation must imitate the job scheduler for a given set of jobs and resources. It must decide, if a job can be started on the system depending on the required resources. The jobs have to be sorted with respect to their current priority and the scheduling algorithm has to be simulated adequately.

The major design goal for this simulation program is the configurability. The simulation has to be tailored to different system types such as JUROPA and JUGENE, which are among others currently maintained by the JSC. While JUROPA is a general-purpose cluster using the Moab Workload Manager for job scheduling (see [2]), JUGENE is needed for calculation intensive and high-scalable applications and uses the IBM Tivoli Workload Scheduler LoadLeveler as scheduling system (see [3]).

The differences and similarities of these supercomputer's scheduling systems are analysed in this master's thesis in order to support as many scheduling features and parameter changes as possible and to minimize the need of adapting the implementation for new systems. Moreover, extensibility plays an important role in the design of the simulation. As the possibilities of configuration are always limited, it might be necessary to extend the simulation's functionality for instance to support adapted or new scheduling algorithms, which cannot be represented in configuration parameters.

The simulation program will be integrated into the workflow of LLview. LLview is a generic monitoring system for supercomputers. It collects real-time information of the monitored system and visualizes it on a single well-arranged display (see [4]). The gathered information of the parallel system is converted into the Large-scale system Markup Language (LML), which defines the structure for XML files describing the current status of a supercomputer (see [5]). Each LML file contains a complete status snapshot of the monitored system and functions as communication layer between LML generating and LML processing applications. Therefore, it is the ideal input for the job scheduler simulator, which needs job and resource information in order to generate a predicted schedule. In addition, LML will also function as the simulation's output format. The job information will simply be extended by additional attributes containing the predicted execution time period for currently waiting jobs. As a result, the scheduling simulator can be seamlessly embedded into the monitoring system as a stand-alone module. To sum up, the integration of the simulator will be implemented in three major steps: LLview gathers the needed information from the supercomputer and generates an LML file, the simulation extends the file and the results are visualized by a monitoring client.

LLview also generates the configuration for the simulator. This is reasonable as it is able to provide inside information of the job scheduler. Integrating the simulation program in this way allows to focus completely on the task of the scheduling simulation. There is no need to define the data-format for the configuration or to consider how the simulator's input is generated. However, it will remain a stand-alone application and can also be used without LLview by providing well-defined LML files as input from an arbitrary source.

1.1.1 Job scheduler definition

For scalability reasons the job scheduling system of a supercomputer is often a combination of different scheduling layers. Arnold [6] describes the scheduling system as a hierarchy of three independently working schedulers:

- the global **job** scheduler responsible for mapping complete jobs to the system resources
- the **task** scheduler, which implements multitasking on a single processor
- the **thread** scheduler, which represents the finest scheduling system and decides, which thread of a process is executed at a specific time frame

The simulation program developed in this project only handles the global scheduler layer. The simulated supercomputer is assumed to run a batch system, which is accepting job submissions, inserting them into queues and executing them in a schedule generated by the global scheduler. The global scheduling system itself can be defined as a combination of three components: "Scheduling policy, objective function and scheduling algorithm" [1]. In this definition the scheduling policy defines high level scheduling rules. E.g. one of these rules could specify that 50 percent of the system resources are reserved for a special user group. The objective function accepts a given schedule as input and calculates a scalar value expressing the quality of the input schedule. This function is needed to evaluate generated schedules in order to compare different scheduling algorithms and choose the best fitting algorithm. The last component of the scheduling system is the scheduling algorithm, which is responsible for producing efficient schedules (see [1]).

As the objective function is only needed for choosing the appropriate scheduling algorithm, the developed simulation program has to focus on handling the scheduling policy and algorithm. The program must be designed to interpret the scheduling policy from its input configuration. Adding or adapting rules of the scheduling policy will be done by changing the configuration.

However, the scheduling algorithms often cannot simply be defined via configuration parameters as their behaviour is too complex and too different for mapping them on a simple set of parameters. The simulator will support a set of popular scheduling algorithms such as First-Come-First-Served or List-Scheduling, which can be chosen in the simulator's configuration. The extensibility of the simulation program will then allow to adapt the provided scheduling algorithms to the specific needs of a supercomputer. New scheduling algorithms can be integrated into the simulation program as additional classes defining the algorithm's strategy. As a result, the program provides an extensible basis for predicting various job scheduling systems.

1.2 Outline of the thesis

This thesis comprises eight chapters. Chapter 2 deals with the analysis of the given problem. It proposes a mathematical definition of the scheduling problem, which has to be solved by the simulated batch systems, and presents the existing scheduler simulation together with the monitoring environment, in which the developed simulation program has to be embedded. The third chapter documents practical examples for approximating the optimal solution of the previously defined scheduling problem. In doing so, the popular and often implemented scheduling algorithms First-Come-First-Served, List-Scheduling and their extension by Backfilling are introduced. It also summarizes the batch system architectures, configuration parameters and scheduling algorithms of Loadleveler and Moab in order to allow for abstracting their similarities into a generic simulation program.

After this analysis of the current state, chapter 4 focuses on the concept of the developed scheduler simulator. Design decisions are documented as well as how the simulation program tries to balance the conflicting targets of generalization, efficiency, flexible configuration and extensibility. The main components of the simulation and their interactions are presented by analysing needed data structures and algorithms. In order to improve the simulation's efficiency chapter 5 investigates the time consuming parts of the implemented algorithms and outlines approaches for optimising them. Afterwards, chapter 6 deals with implementation details including guidelines for extending the developed code basis and the description of used libraries.

Chapter 7 documents how the simulation program can be tested. The process of generating example input files is documented and the simulation's algorithms are retraced with the help of real workload examples. The last chapter summarizes the findings of this project and provides an outlook for extensions of the developed simulation program.

Chapter 2

Problem analysis

This chapter collects information about the scheduling problem, which has to be solved by the simulation program, and documents the current status of the monitoring environment, which forms the technical basis for the simulator. In order to avoid circumscriptions the simulation program, which is developed within this thesis, is herein after referred to as *JuFo*. This especially allows for distinguishing JuFo from the existing simulator, which is part of LLview.

2.1 The scheduling problem

The problem defined in this section has to be solved by the scheduling machines and at the same time by the tool JuFo. This section tries to grasp the intuitively given scheduling problem in a more theoretical approach. In this context job scheduling means to generate a mapping of the input jobs to a dispatch time and to resources, on which the job is executed. JuFo functions as on-line scheduler prediction. The actual job duration is unknown, since the simulator is usually run before the jobs are executed. But the wall clock limit provided by the users, which represents an upper limit for the job's run time, can be used as a corresponding approximation. The following definitions are needed for modelling the scheduling problem:

$$
\begin{aligned}
n &: \text{ number of scheduled jobs} \\
J = \{1,\ldots,n\} &: \text{ set of job indices} \\
M &: \text{ set of possible resource requests} \\
R &: \text{ set of totally available resources provided by the supercomputer}
\end{aligned}
$$

The input data for each job with index $j \in J$ is given by a resource request $r_j \in M$ and a wall clock limit $w_j \in \mathbb{R}_{>0}$. The possible kinds of resource requests in M are not defined explicitly in order to keep the problem definition generic. This set M could contain requests for processors, memory or network topologies, for example. In the same manner the set R can be seen as a template for resources provided by the supercomputer. Note, that the request type has to correspond with the provided resources. If the model only allows to request processors, the provided resources must contain processors as well. Otherwise the

jobs can never be dispatched. The schedule, which has to be generated by the simulation, is then given by

$$s = (s_1, \ldots, s_n)^T$$

where each $s_j \in \mathbb{R}_{\geq 0}, j \in J$ represents the predicted dispatch time of job j. The resource requests r_j have to be mapped to the actual resources in R, which can be achieved with the help of the following function:

$$T = \mathbb{R}$$
$$f : M \times T \rightarrow \mathscr{P}(R)$$

Note, that $\mathscr{P}(R)$ represents the power set of R. It contains every possible subset of R. Function f receives a resource request and a time value within the simulated time span. It returns a subset of the actual resources available at the given time. For instance, the supercomputer could merely provide processors as resources. Valid resource requests would then be subsets of processors, which each job requires. This would lead to $R = \{1, \ldots, m\}$ and $M = R$, whereat m specifies the total number of configured processors. While each element in R identifies a processor, each element in M represents the requested number of processors for a job. Then the task of f is to map a resource request at a given time to a matching subset of available processors, which is returned when calling f. In a real system the central resource manager is responsible for realizing this function.

A generated schedule is only valid, if the resources of all running jobs are pairwise disjoint at any time within the schedule. The set of running jobs of a schedule s at a given time t is defined by $g(t,s)$ with

$$g : T \times \mathbb{R}^n_{\geq 0} \rightarrow \mathscr{P}(J), (t,s) \rightarrow g(t,s) := \{j \in J | s_j \leq t \leq s_j + w_j\}$$

The above condition for the validity of a schedule can now be expressed by

$$f(r_j,t) \cap f(r_i,t) = \emptyset, \forall i, j \in g(t,s), i \neq j, \forall t \in T \tag{2.1}$$

With these definitions given the scheduling problem consists in the search of a valid schedule, which maximizes the objective function. This function introduced in section 1.1.1 needs to implement the following constraints

$$V = \{s \in \mathbb{R}^n_{\geq 0} | (2.1) \text{ is true for } s\}$$
$$o : V \rightarrow \mathbb{R}$$

V contains all valid schedules and the objective function o evaluates their quality. A reasonable example for an objective function is to calculate the total duration needed to execute all queued jobs. The corresponding function can be defined by

$$s \rightarrow o(s) := -\max_{j \in J}(s_j + w_j)$$

The maximum term is negated so that the objective function has to be maximized in order to find the optimal schedule.

Finally, the scheduling problem is given as selecting the schedule $s_{opt} \in V$ so as to

$$o(s_{opt}) = \max_{s \in V} o(s)$$

This defines the task of the schedulers. They have to manage the available resources R and allocate subsets of them to each job. A job requests resources for a coherent period of time. A schedule specifies the placement of the jobs into both the resource and time dimension, which causes this problem to be that complex. Feitelson et al. [7] address this complexity by stating the following

> "Scheduling on a parallel computer is complex since it involves scheduling over two dimensions, time and space"

Among all valid schedules in V, which cover both of these dimensions, the optimal schedule is searched according to the objective function, which allows for comparing the generated schedules.

A more general scheduling problem is examined carefully by Brucker et al. [8]. It defines the *resource-constrained project scheduling problem (RCPSP)*, which consists in the combinatorial optimization of a schedule regarding an objective function and which can be applied to a wide range of practical examples. In this problem, the jobs are called activities and they require *renewable resources* of different types for a specific time period. The number of totally available resources is constant and the activities are only allowed to use disjunct sets of resources simultaneously. Additionally, precedence rules are included, which define dependencies between the activities. This allows to specify, that an activity has to wait for another activity to complete. Schedules are modelled as n-dimensional vectors, too. As a result the scheduling problem defined above represents a specialisation of the RCPSP. However, the precedence rules are disregarded in order to focus on the problem's core, although this concept is reasonable for batch jobs as they might depend on the results of previous jobs. Brucker et al. [8, p. 34] prove that the optimization problem given by RCPSP is *NP-hard*. I.e. there is no known algorithm for solving this problem in polynomial time. That is one reason for applying heuristic solutions to this scheduling problem as finding the exact solution cannot be achieved at acceptable costs.

2.2 Current status

JuFo is embedded as a stand alone module into the monitoring environment based on the following three major components

LLview – the monitoring tool for supercomputers consisting of a server application, which gathers current status information of the monitored system, and a client application for visualizing the gathered information

LML – XML format for the description of a supercomputer's status

PTP – the Parallel Tools Platform is a collection of intertwined Eclipse plug ins for supporting developers of parallel applications (the project documentation can be found in [9])

After providing an overview of the environment built by these tools and after describing the connection of these components each is presented in detail within this section. Moreover, the integration of JuFo into this environment is outlined.

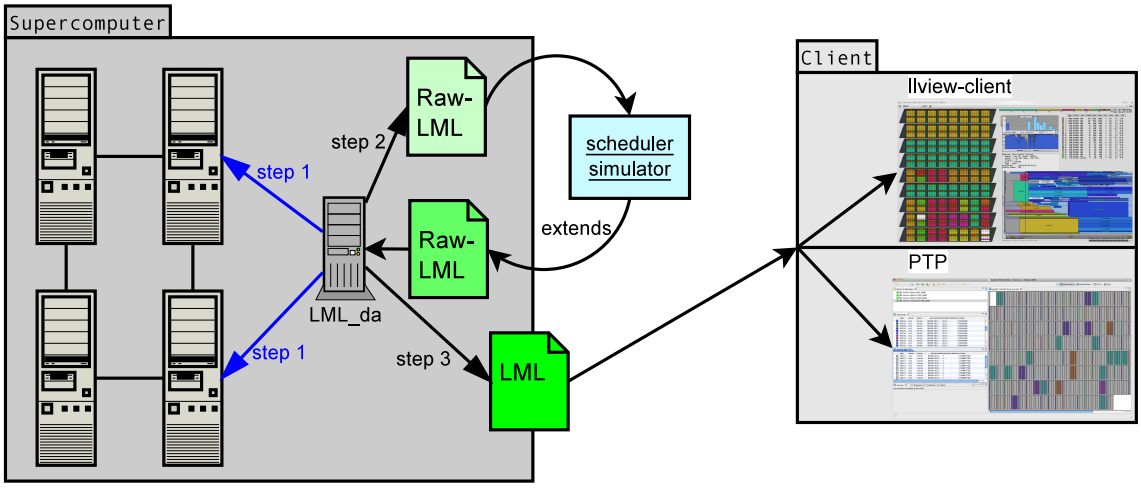

Figure 2.1: overview of the monitoring environment

The overall connection and data flow of the three mentioned components is demonstrated in figure 2.1. The environment, in which the scheduler simulator is embedded, represents a client server architecture. LLview provides the server application *LML_da* for gathering status information of a supercomputer. Thereby relevant information about the resources provided by the monitored system such as details about running and queued jobs, the status of the compute nodes and submitted reservations is collected, which is highlighted by the *step 1* arrows between LML_da and the supercomputer. The second step processed by this server application is the generation of a raw LML file containing the status of gathered jobs, nodes and reservations. LML was designed to describe the status of a parallel system in such a way, that it can be visualized as easy as possible. This simplifies the development of monitoring clients for visualizing LML data. The LML file produced in the third step provides exactly this kind of format close to a possible visualization, while the raw LML file's format is an abstraction of the information directly gathered from the supercomputer.

JuFo requires the raw LML format as input, because its target is not visualization but mapping jobs and reservations to available nodes in order to produce a predicted schedule. Thus, the simulator processes a raw LML file and extends the job information by additional attributes, which hold the predicted dispatch and completion time as well as allocated resources for each job or reasons, why a specific job could not be started within the duration of the simulation. The extended LML file is handed back to LML_da, which converts it in the third step into standard LML. This LML file is subsequently passed via SSH, HTTP or other suitable protocols to the visualization clients of LLview or PTP. Their task is parsing the LML information and converting it into a corresponding graphical representation. Therefore, LML provides XML tags for tables, charts and text boxes, which contain the gathered status information and have simple graphical representations.

LML functions as communication layer between programs, which gather the supercomputer's status information, and those programs, which are responsible for the visualiza-

tion. This allows the exchange of the client and server components as both sides rely only on the creation and interpretation of valid LML. The validity of the generated LML files is guaranteed by a corresponding XML Schema and an additional tool, which checks semantic constraints not covered by the XML Schema. In almost the same manner, LML works as a layer between JuFo and LML_da. The raw LML file contains all necessary information and configuration for the scheduler simulator, which optionally extends this input file. Thus, the simulator becomes an encapsulated module, which could even be excluded from the depicted workflow of figure 2.1.

In the same way other LML processing applications can be embedded into this environment. For example the prediction module implemented as part of LLview is included into this workflow in the manner described except that it has been developed before LML was established and is therefore based on an older XML format.

2.2.1 Job scheduling prediction of LLview

Besides the monitoring components such as a job table or statistical charts LLview already provides a simulator for job schedulers, which is called *prediction* (see [4]). Its actual implementation is named *SchedSim*, which is used in the following to refer to the scheduler simulator implemented as part of LLview. This prediction module is the origin for this thesis and forms a basis for the concept of JuFo. This section analyses the design of SchedSim and outlines the data structures and algorithms of the given implementation.

The prediction module is separated into the simulation and visualization part, which matches with the client server architecture of LLview. The simulation works on a set of jobs, reservations and nodes and produces an estimated schedule based on the current system status. The visualization parses the produced schedule from the added job attributes and plots it in the prediction component shown in figure 2.2.

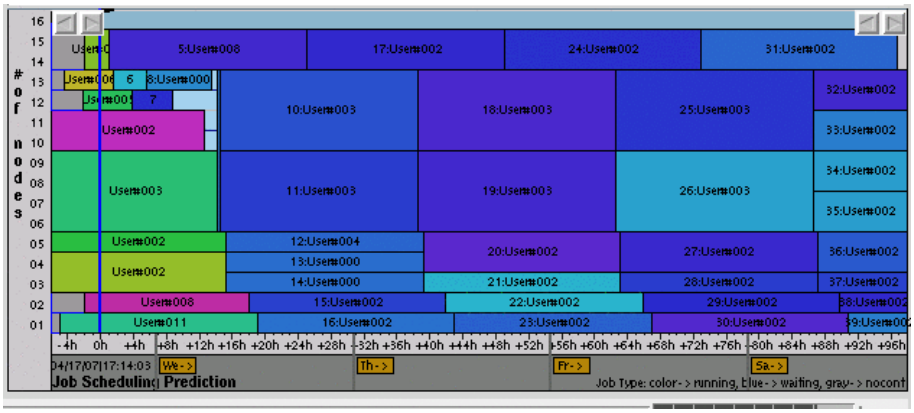

Figure 2.2: prediction component in LLview (source [4])

This graph draws a rectangle for each job in the schedule. The domain represents the predicted time slot covered by the schedule, while the co-domain is divided into the compute nodes allocated to the jobs. However, the rectangles' positions do not define the absolute position of the job on the system, but only the number of nodes allocated to the job.

SchedSim works on a simple structured XML format, which is defined implicitly by LLview.

Only LLview controls this XML format, which allows arbitrary modifications to the XML structure and semantic. The reason for that, is that LLview was not originally designed to be extended by external applications. Consequently the XML format represents an internal format for simply passing data through the different workflow steps of LLview. Due to the idea of possible extensions and enhanced interface definitions for LLview, LML replaces the implicit format.

SchedSim is written in Perl and is especially designed for the supercomputers maintained at JSC. It has been steadily adapted regarding configuration changes and new requirements of the monitored systems. Thus, a huge range of configuration details and job attributes are considered by this prediction module. This improves the accuracy of SchedSim, but also increases its complexity and runtime.

As a result, for JuFo the three modules for gathering status information of the simulated system, for the core simulation and for its visualisation need to be separated from each other as these are independent steps. For this reason LML is used as communication layer between these steps.

Data structures

The main data processed by the prediction software comprises the nodes, reservations, queues and jobs of the supercomputer. Nodes are parts of the machine, which provide processors, memory and other resources to the allocated jobs. The numbers of these resources are passed to the prediction as key value pairs listed for each node. Reservations assign sets of resources to a specific user group in order to provide faster access to the reserved parts of the system. In this context a queue is a category, to which a job is submitted. Each queue is connected to corresponding scheduling policies, which amongst others define how many jobs of one queue can be started or the number of simultaneously started jobs per user. A queue also determines the allowed number of cores, which can be allocated to all jobs of one queue. In addition, the job queue specifies explicitly, which compute resources are allowed to be allocated to jobs of the particular queue. E.g. this allows for cutting the system into halves by assigning each half's compute nodes to a corresponding queue. As a result, the system would act like two disjunct supercomputers. To conclude, jobs are grouped in queues and each queue possesses its own set of scheduling rules. The job data passed to SchedSim holds running and waiting jobs, for which dispatch and completion times have to be estimated.

SchedSim makes extensive use of nested hashes, which hold the simulation data. Although no explicit classes are defined, these hashes and the global functions, which work on them, form a set of implicit classes. The most important classes are depicted in diagram 2.3. However, this figure only displays the main excerpt of the prediction's implementation as this analysis rather intends to give an overview than to document all configuration details.

The *Simulation* class is the entry point of this module. It collects the input nodes, reservations and jobs in corresponding lists. Moreover, the simulation aggregates one *Timeline* and one *SystemState* instance. The Timeline represents a list of events, which occur during the simulated time interval. An event is caused either by a job or a reservation, which are started or completed. It usually changes the system state, because resources are consumed or released by the event source. The events are sorted by their time stamps. The System-

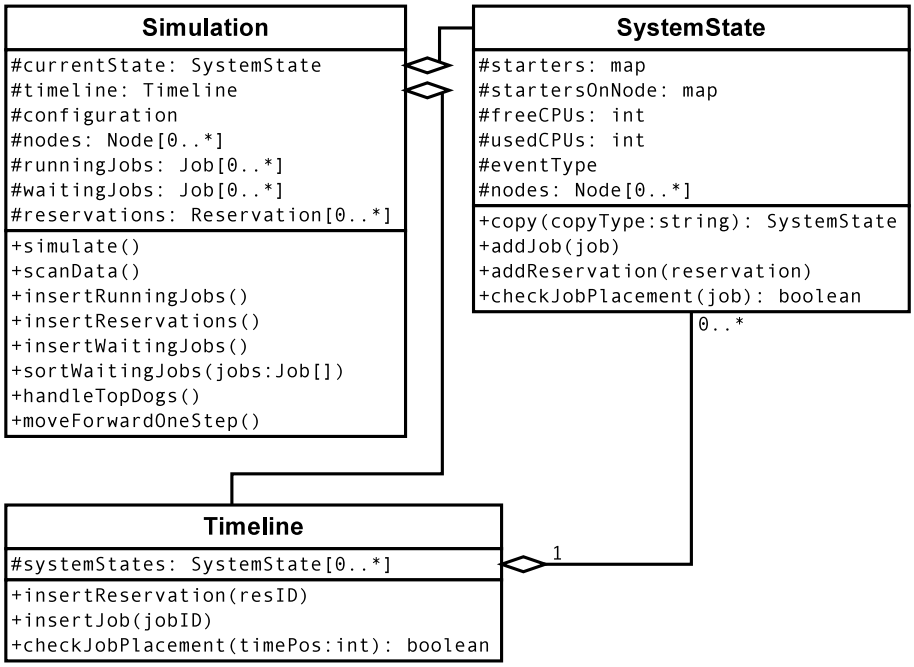

Figure 2.3: major classes within SchedSim

State instance manages the state of the resources provided by the simulated machine. It keeps track of available nodes and checks the constraints given by queues. The SystemState represents the simulation of the resource manager, which is in a real supercomputer responsible for managing and allocating compute resources. In its *nodes* attribute it stores a copy of the Simulation's nodes in order to adjust the number of compute resources provided by the nodes. Events in the Timeline are also saved as SystemState instances. For this purpose SchedSim creates thin copies of the SystemState, which only contain global resource attributes such as *freeCPUs* and *usedCPUs*. The attributes connected to each node like the number of processors used by each queue are not copied, because this information is not required for every event of Timeline. Next to the global state attributes each event stores a type –such as job start or completion– and a time stamp.

The *currentState* instance of the Simulation holds status information about the resources provided by the simulated supercomputer. The status information is altered throughout the simulation process. Each event in the Timeline affects resources by consuming or releasing them. On the one hand, the Timeline is required for logging the generated schedule. On the other hand it allows to determine, if a job started at a given time would interfere with another job or reservation in the future.

As a result, the data model is built by highly configurable nodes, reservations and jobs. The simulation working on this data model is divided into three major packages: Simulation, Timeline and SystemState. They implement the scheduling algorithm, the log of the simulation events and the management of available compute resources. Although they form a very flexible simulation basis, it is hard to extend SchedSim. The scope of each package is too large and they strongly depend on the implementations of the other packages. I.e. changing the implementation of one package would force the adaption of the entire simulation program. JuFo needs to strictly separate these modules so that they only rely

on abstract interfaces, but not on the actual implementation. Moreover, the functional range of the packages has to be reduced by dividing them into independent sub tasks. This helps to understand each package's function and improves extensibility.

Algorithms – overview

At first the Simulation's *simulate* function is called, which consecutively scans the input data, inserts already dispatched jobs and places reservations into the Timeline. Then it triggers the *insertWaitingJobs* function, which executes the main simulation. It sorts the waiting jobs by fixed priority criteria. The Timeline is iterated and after adapting the SystemState by each event with a call of *moveForwardOneStep*, the waiting jobs list is traversed in order to place as many suitable jobs into the system as possible. This approach matches with the List-Scheduling algorithm, which is analysed in section 3.1.2. The prediction module can also be configured to place a given number of highest prioritised jobs at the first matching time slot. These jobs are called *top dogs*. Afterwards, lower ranked jobs are used for backfilling the idling resources before the top dog dispatch (compare with section 3.1.3).

The classes SystemState and Timeline each provide functions for inserting jobs and reservations. Inserting a job into the SystemState means to assign required resources to the job. These resources are consumed and afterwards unavailable throughout the job duration. The SystemState does neither save any past nor future information. It just manages the number of currently available resources and the jobs, which are assigned to them. The insertion of a job into the Timeline is done by adding corresponding events. Jobs, which were dispatched before the simulation's begin, produce only one event for logging their completion time. For the other jobs a second event is added, which records the job start time. Before the actual insertion of a job, the *checkJobPlacement* functions determine, whether sufficient resources for the passed job can be allocated. The corresponding function of the Timeline directly inserts the job if possible.

SchedSim implements two actually independent scheduling algorithms –List-Scheduling and backfilling– in a single function. When re-designing the simulation program the implementations have to be isolated from each other. This allows for independently changing and testing each scheduling algorithm.

Scheduling algorithm framework

After introducing the main classes and the function call hierarchy the following section presents the core algorithms implemented in SchedSim. The most important part of the prediction is the placement of waiting jobs as the insertion of running jobs and reservations is already specified by the batch system of the supercomputer, which selects the assigned resources. Figure 2.4 shows a simplified version of the Simulation's function *insertWaitingJobs* illustrated by a Nassi-Shneiderman diagram (NSD). This function builds a framework for the implemented scheduling algorithm. It is responsible for interactions between Timeline and SystemState.

Its first step is sorting the waiting jobs with the help of a comparator function. It compares

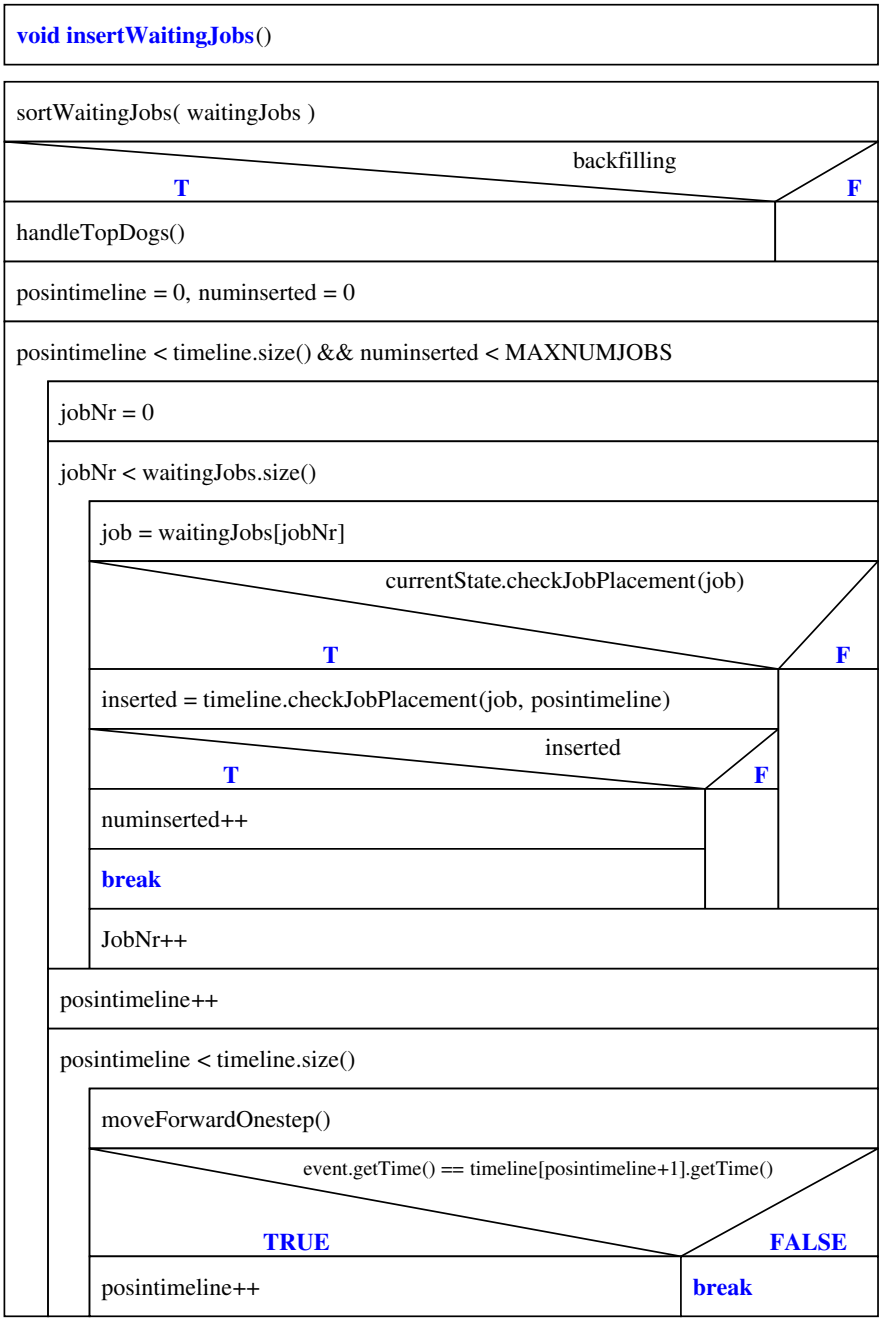

Figure 2.4: NSD for insertWaitingJobs function of the Simulation class

two jobs by a sequence of priority criteria. The first evaluated criterion is the system priority, which can be requested from the scheduling machine. If this value is equal for both jobs the smaller step number gets a higher rank. The step number sorts jobs among each other, which are submitted by the same user and as part of job chains. A job with a higher step number in one chain usually depends on the results of prior job steps. If the jobs are still ranked equally, a few less relevant criteria such as comparing the job names are applied in order to achieve unambiguous sorting. The first jobs in the sorted list are the top dogs, which are inserted into the Timeline by the *handleTopDogs* function explained later. The

number of jobs handled as top dogs can be configured. Another configuration parameter –named *MAXNUMJOBS*– is the maximum number of jobs, which are inserted into the generated schedule. This parameter allows to limit the execution time of SchedSim. The outer loop traverses the events stored in the Timeline. At the beginning of the simulation the Timeline contains events for the completions of already running jobs and events for inserted reservations. The top dog placement adds the corresponding events at the first time slots, where the top dogs can be inserted into the system.

At each event two inner loops are executed: the first loop searches for jobs, which can be inserted into the current SystemState, and adds corresponding events to the Timeline. The second loop actually processes the events and is responsible for altering the SystemState. The first loop iterates over remaining waiting jobs. It tests, whether the current state allows the job's insertion, by calling the *checkJobPlacement* function of the SystemState. If this was successful, it has to be checked by the Timeline's *checkJobPlacement* function, if the job would interfere with any future event such as a starting top dog or reservation. This function has to make sure that the system possesses sufficient resources for the passed job throughout the entire estimated job duration. As this prediction implements an on-line scheduling simulation, there is no reliable information about the job completion time. The users configure a wall clock limit, which is an upper limit for the job execution time. Both *checkJobPlacement* functions are presented later within this section. If a job is inserted successfully into the Timeline, the first inner loop is finished. Afterwards, the second loop executes all successive events with the same time stamp and thereby changes the resource states managed by the currentState instance.

Since JuFo has to realize a set of different scheduling algorithms, a common framework for their implementations has to be extracted from the draft provided by SchedSim. The only function of the Simulation class, which depends on the particular scheduling algorithm, is *insertWaitingJobs*, while the functions for the insertion of reservations and already dispatched jobs are unaffected by the scheduling algorithm. Thus, a reasonable approach for varying these algorithms is to implement a common simulation framework and adapt the *insertWaitingJobs* function to the needs of each scheduling algorithm. Moreover, the strategy for sorting the jobs according to the system priority is an independent task, for which JuFo can introduce an additional package in order to reduce the complexity of the Simulation class. Parameters such as *MAXNUMJOBS* for limiting the run time of the simulation also have to be used in JuFo to ensure that it can be applied as on-line scheduler prediction. Besides a resource manager the new simulation program must use a Timeline to allow the implementation of resource reservations. The used list of events along with a reference to a corresponding system state is a promising data structure.

Top dog placement

The handleTopDogs function is called, if the prediction's configuration defines to use the backfilling algorithm. A shortened version of the implementation of this function is shown in figure 2.5.

It tries to insert a number of *NUMTOPDOGS* jobs into the Timeline by traversing the pre-sorted waiting jobs list. This function implicitly implements a state machine for each job. The initial state of a job is *NOTFOUND*, which means that no start position is found

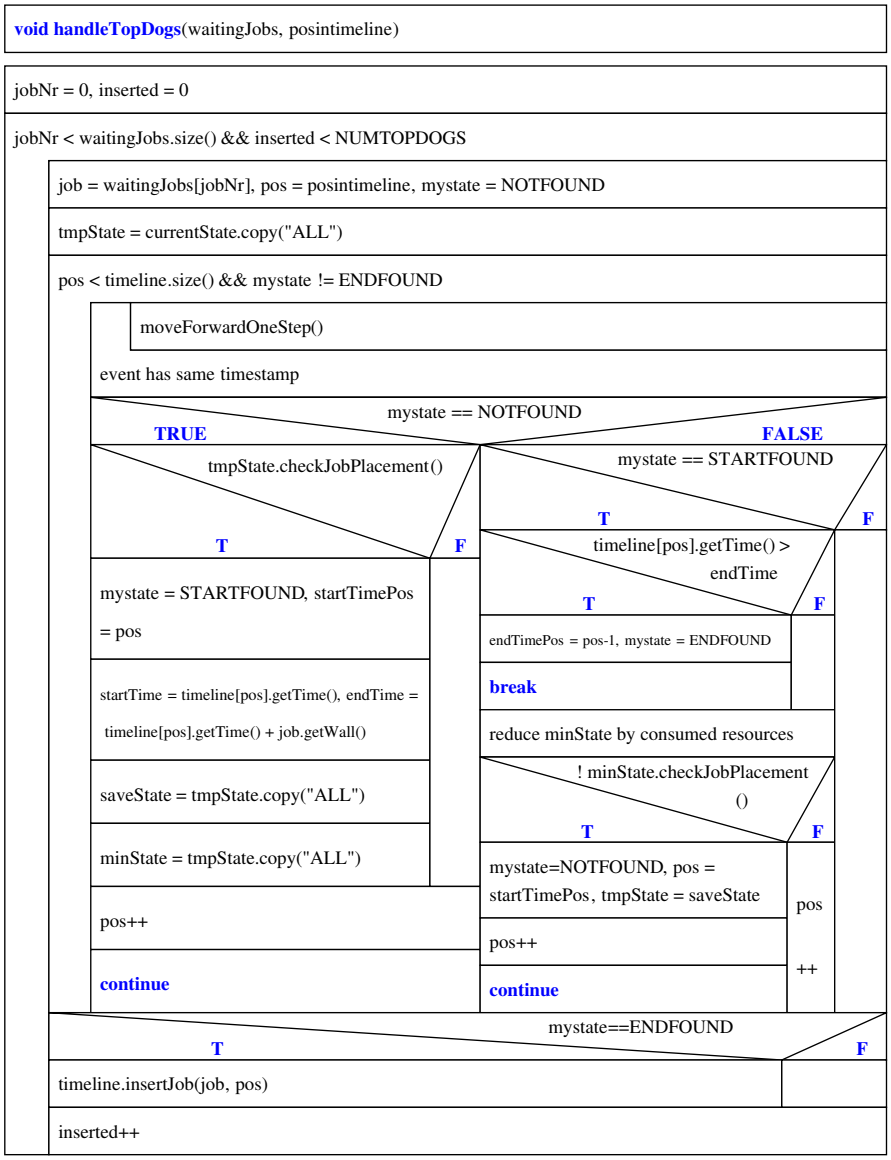

Figure 2.5: NSD for handleTopDogs function of the Simulation class

so far. This state transits to *STARTFOUND*, when the function detects the first Timeline position with sufficient resources for the current top dog. The last state is *ENDFOUND* assigned to the job, if a completely matching time slot is found within the Timeline.

For each top dog this function searches a coherent time slot, in which the SystemState allows the job's insertion at any time and on identical resources. It is not sufficient to simply check the SystemState's checkJobPlacement function at each event within the job duration, because the resource set, on which the job is started, could change every time. Changing the resources during the job runtime is not allowed for this simulation as the migration of jobs is no option for the simulated supercomputers. The currentState is copied completely into the *tmpState* variable. This allows to simulate the subsequent events in the Timeline without altering the actual SystemState. The inner loop implements job state transitions. At first the dispatch position is searched as long as the state is set to *NOTFOUND*. On

the first event with sufficient resources tested by *tmpState.checkJobPlacement()* the state is switched, the predicted dispatch and completion times are calculated and tmpState is copied into *saveState* and *minState*. While minState is needed to store the minimal set of available resources throughout the job's time slot, saveState is used as a checkpoint state. This state is restored, when the job cannot be inserted into a future system state.

After a possible start time is found for the job, each iteration of the inner loop tests, whether the job can still be inserted after updating tmpState by calling *moveForwardOneStep* for all events with the same time stamp. This test is executed on the minState, which is updated in each iteration by comparing it to the current tmpState. All resources, which are marked as unavailable at a single position in the Timeline, are removed from the resources of minState. This ensures, that the job does not need to be migrated from one resource set to another. If the *checkJobPlacement* test fails at one time position, the state machine switches to the initial *NOTFOUND* state and tmpState is restored to the checkpoint state. Furthermore, the search for a new time slot is started at one event position after the *startTimePos* value. This is reasonable as unavailable resources at startTimePos might be released by a job completion at the following event. If a job can be inserted into minState at all events within its estimated duration, the corresponding *endTimePos* is set and *mystate* switches to *ENDFOUND*. Finally, the job is inserted into the Timeline at the detected time slot.

The backfilling algorithm requires the knowledge of available resources throughout a simulated time span. For the top dogs the future events of the Timeline are executed in a forward simulation. This requires to copy the entire system state in order to use it as data model for this forward simulation. In the same manner, for placing a job it is not sufficient to check for available resources only at the current position in timeline. Therefore, JuFo has to allow for efficiently copying the system state and needs to implement forward simulations on a copied state. Furthermore, a data structure for the resources allocated to a job is required. SchedSim manages these resources with a list of the node IDs used by each job. A more efficient data structure would be beneficial.

Placing other jobs

With the two functions for inserting the waiting jobs and reserving time slots for the top dogs described the main idea of SchedSim is outlined. Both functions depend on the *checkJobPlacement* functions of the Timeline and the SystemState, which are explained in the following. The Timeline's function is basically a subset of the *handleTopDogs* functionality. It answers the question, whether a passed job can be inserted at a given position in Timeline. Therefore, it has to be checked at each event within the predicted job duration, if the simulated SystemState possesses sufficient resources. This is equal to the algorithm in the *handleTopDogs* function, when mystate is set to *STARTFOUND*. But if an event is found, which does not allow the insertion of the job, the *checkJobPlacement* function does not search for another dispatch time. It will simply return that an insertion is not possible at the requested position in the Timeline.

The last described function is the SystemState's *checkJobPlacement*, which is responsible for monitoring and allocating the compute resources to the jobs. It simulates the resource manager of a supercomputer. A code diagram of this function is depicted in figure 2.6.

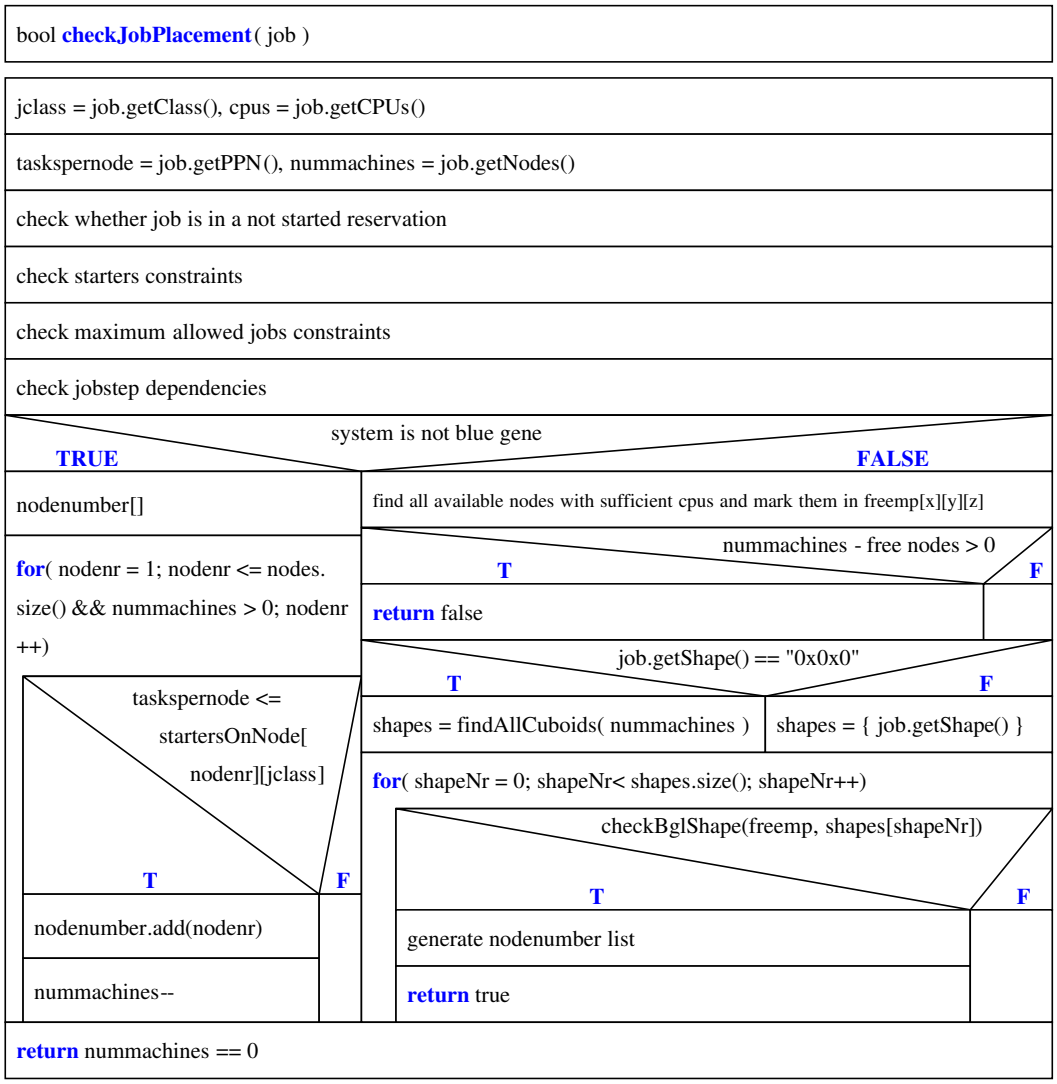

Figure 2.6: NSD for the SystemState's checkJobPlacement function

This function only works on a single position in time. In contrast to the Timeline, it does not need to consider any past or future information of the simulation. The most important job requirements, which have to be met by the SystemState, are collected in the first two blocks of the diagram. A job is submitted to a queue, it requests a certain number of processors, which are spread on a list of nodes. Each of the nodes needs to provide at least a number of *taskspernode* processors. The following four blocks execute simple constraints, which are able to forbid the job's insertion into the SystemState. At first it is checked, if the job is associated with a reservation. If the reservation was not started so far, the job is not allowed to be scheduled either. Afterwards, the function compares the requested processors with the number of available processors for the job queue. Another fast check is to control the constraint given by the upper limit for simultaneously started jobs in total and per user. In addition, the job might be unable to start, if prior job steps are not finished before the current time position.

After all these constraints are achieved, the further behaviour of the function depends on

the simulated system type. The simpler case is given for a cluster system such as JUROPA. The algorithm assumes that for this system type arbitrary nodes can be assigned to a job. The location of the nodes within the system is dispensable. Thus, all nodes are traversed in order to filter a set of *nummachines* nodes, each of which possesses at least a number of *tasks*pernode available processors in the job queue. If a corresponding node set was found, the function returns, that the job is allowed to start, and the found resource set is passed back to the calling function as a job attribute.

For a Blue Gene system searching a suitable set of nodes is more complicated as they have to be arranged in a sub cuboid of the global cuboid formed by all compute nodes. A first step is to detect all nodes, which could be part of the resource set with regard to the required processors on each node. For this reason the tridimensional array *freemp* is filled with a zero for each node, which does not possess sufficient resources, and a one for the suitable nodes. If the number of applicable nodes is lower than the total number of requested nodes, the job cannot be started and the *checkJobPlacement* function is aborted. The structure of *freemp* complies with the logical structure of the system nodes. They are placed in a tridimensional torus network, which is examined carefully in section 3.2.1. Hence, the *freemp* array allows to locate a subset of nodes, which matches with the job request. A Blue Gene system allows the jobs to define the shape of the sub cuboid, in which the assigned nodes should be arranged. The shape is defined by three values, which represent the edge lengths in each dimension of the cuboid. If no shape is determined by the user, the function looks for all possible node cuboids with a volume equal to *nummachines*. The following loop iterates over all found shapes and tests, whether *freemp* contains a subset of suitable nodes in the current shape by calling the function *checkBglShape*. This function is quite time-consuming as it has to check for a valid sub cuboid starting at each location in the *freemp* array. If an available set of nodes in a proper shape is found, it is saved as a job attribute and the function returns successfully just like described in the case of a cluster system.

To conclude, a job needs to fulfil a lot of constraints before it can be dispatched. JuFo needs to simplify the configuration of these constraints. Furthermore, the given implementation of the system state handles two resource manager types –JUROPA and JUGENE– simultaneously. In the new simulation program their implementations must be separated from each other. A common abstract interface must be defined so that the Simulation class can use any implementation without knowing the actual type. This also allows for adding another system state type without adapting the implementation of the simulation framework.

Summary

LLview's scheduler prediction is an advanced simulation especially designed for the supercomputers maintained at JSC. It processes highly configurable input data comprising jobs, reservations and nodes. The implementation is composed of a Simulation class, which applies the scheduling algorithm, a Timeline for logging the events and the SystemState, which is responsible for allocating resources to the jobs and reservations. The scheduling algorithms List-Scheduling and Backfilling are supported. Moreover, the prediction module is aware of the simulated system's topology since the SystemState class manages each node's available resources and sub cuboids are searched for jobs requesting the torus

network on Blue Gene systems. On the one hand, the huge range of configuration details considered by the prediction improves its accuracy. On the other hand, its efficiency suffers from these details and the growing size of simulated nodes and jobs. In order to keep the run time of the simulation short it is possible to restrict the number of inserted jobs.

However, with the implicitly defined XML input format and the fact, that the implementation often had to be adapted to new system configurations, the extensibility of SchedSim is limited. As a result, a more generic redesign of this prediction module is needed. A new prediction should provide the same level of configuration, avoid the need of full reimplementations for new systems and should also feature enhanced efficiency.

2.2.2 LML

The Large-scale system Markup Language (LML) defines the structure of an XML format for describing status information of massively parallel systems. It was evolved as communication layer between applications, which gather information about supercomputers, and those, which process and especially visualize the collected data. Therefore, LML is designed to make the visualization of the collected data as easy as possible by providing XML tags, which are close to graphical representations. Figure 2.7 depicts the main components of an LML file.

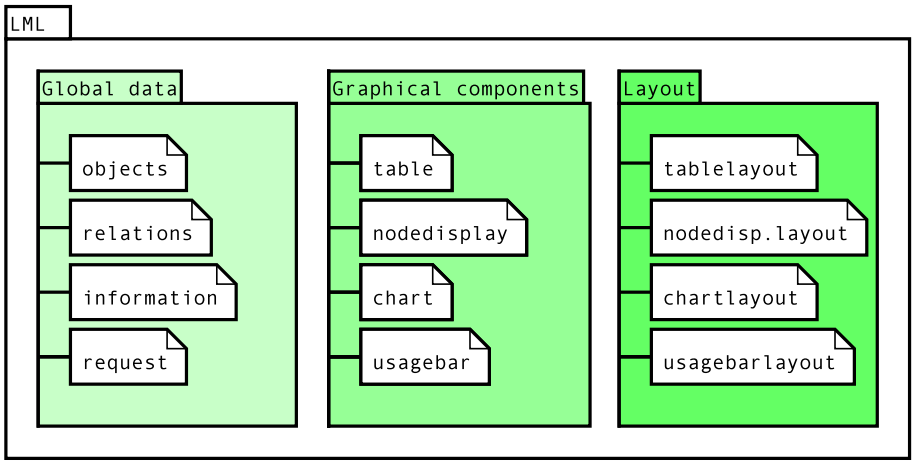

Figure 2.7: main structure of an LML document

The XML elements within each LML file can be categorized into the groups *global data*, *graphical components* and *layout*. Global data comprises a list of objects such as jobs, nodes, reservations or queues, which are identified within this list and referenced by all other LML elements. This is necessary to connect the data among all LML components. The *information* element allows for attaching additional data to each object. For example for each job its owner, the number of requested processors or any other job attribute can be saved. As LML is also applicable for data requests from the monitoring clients to *LML_da*, the *request* element can be used to transmit various request configurations. Thus, LML can be used as bidirectional communication language. The global data group forms the raw LML format, which works as data format for JuFo.

The graphical components contain derived data elements, which are generated with *LML_da*

by processing a raw LML file. These components easily can be converted into corresponding visualizations, because the data structures are designed close to graphical representations. The graphical components do not define how the data needs to be rendered, but they point to possibilities for displaying them. For example the *table* element consists of a set of columns and rows so that a corresponding visualization is easy to implement. However, the final rendering of each monitoring client is allowed to vary. A chart can be used to shape different types of diagrams, which are essential for statistical system monitoring. It allows to define charts about the system load, the number of active users or submitted jobs, for example. The *usagebar* holds information for a special diagram showing all currently running jobs in a horizontal bar. Each job is displayed as a colored rectangle, whose width accords with the job size. Finally, the *nodedisplay* element represents the most important graphical component. Its task is describing the physical structure of a supercomputer with the help of a hierarchy of compute resources. For instance a simple nodedisplay could define a cluster consisting of nodes, each of which contains a set of cores, in a two-level hierarchy. In the second part of the nodedisplay these compute resources are connected to currently running jobs. As a result, this graphical component describes the supercomputer's architecture and also maps its compute resources to the dispatched jobs. Users are then able to locate their jobs on the monitored system. A nodedisplay can be painted with nested rectangles, while each rectangle represents a processor, a node or any other element of the architecture's hierarchy. The rectangles are filled with unique colors in order to identify the job running on the specific compute resource.

The layout group of LML elements is a collection of layout information associated with each graphical component. This information contains optional hints for configuring the visualization of LML. For example the *tablelayout* allows for defining the percentaged width of each column within a table. Adding these layout tags instead of extending the graphical components with layout information, separates the plain status data from the configuration data for the monitoring clients. Each graphical component possesses its own layout element as configuration parameters differ for each component type. An entire documentation of the structure and design decisions in LML is provided by [10].

2.2.3 PTP

The Parallel Tools Platform is a set of Eclipse plug ins for supporting the development of parallel applications. It comprises syntax highlighting, context sensitive code completion for the programming languages *C* and *Fortran*, makefile support, synchronized projects, which means that the local source code is copied automatically to the remote supercomputers, and system monitoring. The latter represents one of the possible monitoring clients used for visualizing the results of JuFo. JSC is involved in the development of this system monitoring component provided by PTP. Its functionality is summarized in this section. The system monitoring of PTP uses LML as communication format. While Eclipse is usually running on a local client, PTP needs to directly connect to the supercomputer in order to gather status information, build the parallel application and submit jobs. This connection and all communication between PTP and the remote machine is established via SSH. The required authentication is handled by external Eclipse plug ins, which allow amongst others the usage of password or public key based authentication. Once the connection is established, the server application of LLview is copied to the remote system and reacts on LML requests sent from the PTP client. Corresponding LML responses

are generated and passed back to the system monitoring, which is then responsible for rendering the collected status information. As JuFo is written in *C++*, it needs to be compiled on the remote machine. Furthermore, it represents a stand alone module, which is not included into the PTP's server application. As a result, it needs to be installed on the remote machine by the system administrator, for example. PTP's server can then be configured to run the scheduler simulator as an additional step in its workflow.

The system monitoring is implemented as an Eclipse perspective, which is given as a set of views arranged in a specific layout. Each view of this perspective comprises the visualization of one graphical component sent within the LML input file. At present the tables, an info text box and the nodedisplay are implemented. However, independently from this thesis the charts, used for visualizing the results of the scheduler simulation as well as other statistical data, are part of PTP's future work. The various views are able to interact with each other by highlighting corresponding information throughout all components simultaneously. For instance, if the user clicks on a job in the nodedisplay, the corresponding row in the table of running jobs is also highlighted. In the same manner, the visualization of the scheduler simulator might be connected to the other components so that the user is supported as good as possible by linking the shown information. These enhancements of PTP's monitoring system are not included in the implementation of this thesis as it focuses rather on the core simulation than on its visualisation.

2.3 Target

The functionality of the simulation developed within this thesis is generally based on the presented prediction module SchedSim. However, a few disadvantages of this module can be extracted from the previous analysis: it lacks extensibility, because its design and the configuration parameters are specifically tailored to the supercomputers at the JSC. Additionally, the prediction's run time has reached the limits for an on-line scheduler simulation so that its efficiency has to be optimized. A possible method for accelerating the prediction module is its parallelization, whose practicability needs to be investigated.

This leads to the idea of redesigning the prediction module by trying to enhance the determinants abstraction, efficiency and extensibility. JuFo needs to provide a firm and generic code basis, which simplifies the integration of new configuration parameters and scheduling algorithms. Therefore, the batch systems' similarities have to be gathered into reasonable interfaces, which are sufficient for implementing as many scheduling system variations as possible. Despite the requested abstraction the interfaces need to allow the implementation of the functionality of LLview's prediction. In order to prove the usability of these abstraction, the main part of the given prediction module has to be implemented as reference implementation, which reduces the overhead of further developments. As a result, the simulation must be based on these abstract interfaces and must also provide practical example implementations in order to indicate how the interfaces are meant to be used. However, the design goal efficiency conflicts with abstraction and extensibility. Therefore, the simulation needs to balance these objectives in order to find an acceptable compromise.

Another design goal is to separate the scheduler prediction from the process of gathering the input data for the simulation. Without embedding the simulation program into the

monitoring environment outlined in section 2.2 not only the scheduling algorithms, but also the application for retrieving status information of the simulated supercomputer would have to be reimplemented for each new system. By relying on LLview and the monitoring clients JuFo is independent from the interfaces for gathering the supercomputer's status information. Thus, it can focus on simulating the scheduling system by processing the system independent status description language LML.

2.4 Limitations

First of all, despite the objective of abstraction the simulation cannot be applied to every supercomputer. For example, preemption and migration of jobs are not implemented as these concepts are of little relevance for large parallel systems. The simulation needs to make assumptions about the common basis of all possible kinds of batch systems, which should be simulated. It assumes the system to manage a set of compute nodes with associated resources, for example. There might be systems, which do not use the concept of nodes, and therefore cannot be simulated with this application. The more concrete assumptions are made the more efficient and simpler is the corresponding implementation, but at the same time this implementation becomes less generic. To conclude, the simulation cannot handle all possible scheduling concepts, because otherwise the abstraction would be a useless generalization.

Moreover, only a small set of possible scheduling algorithms can be covered by the basic implementation so that it is likely that the provided algorithms have to be extended or changed for other batch systems. But the simulation is designed to simplify these adaptions as much as possible. In addition, this simulation only deals with the global schedulers, which map parallel jobs to the resources provided by the supercomputer. The lower level schedulers are not included into the simulation.

As SchedSim evolved throughout the production of the simulated systems and has gained a huge range of configuration parameters, it is not possible to reimplement its full functionality within the given project period. However, the main algorithms certainly can be transferred into JuFo. In the same manner, the simulation does not represent a fully reliable emulation of a batch system, but is rather a simplified model, which considers the most important factors affecting the scheduler's behaviour. Keeping this model simple is also crucial in order to limit the run time of the simulation so that it can be applied as on-line prediction for monitoring purposes.

Chapter 3

Job scheduling

This chapter covers the analysis of frequently used global scheduling algorithms for supercomputers. A set of relevant strategies is presented in order to provide an overview of possible approaches to the scheduling problem. The second section deals with analysing actually implemented scheduling strategies for the batch systems of JUGENE and JUROPA. This analysis functions as basis for the design of JuFo and guarantees that it is applicable as scheduling predictor for these systems.

3.1 Analysis of job scheduling strategies

The scheduling strategies analysed in this section represent heuristic solutions for the scheduling problem defined in section 2.1. The problem is defined as generating job schedules, which optimise a given objective function. The resulting schedule represents a mapping of jobs to available resources and a corresponding time slot. Input values for the scheduler are jobs requesting resources and the currently available resources provided by the supercomputer. Jobs have to be placed into the schedule as atomic items. Once started by the scheduler, it is not allowed to suspend the job or change its allocated resources. Afterwards, the quality of the schedule can be estimated by given objective functions in order to compare the heuristic solutions.

The following scheduling algorithms only provide approximations for the optimal solutions. In order to find the optimal solution all possible schedules would have to be generated. Afterwards, the evaluation of the objective function for each schedule would allow the extraction of the best solution. As applying brute force is impossible even for a small number of jobs, the following heuristic solutions are analysed. These strategies can also be found in actually implemented scheduling systems, which are described in section 3.2.

3.1.1 First-Come-First-Served

The First-Come-First-Served (FCFS) algorithm is a very simple approach to the scheduling problem. The jobs are sorted by their queue dates with the earliest submitted job on top. They are started successively and no job is able to start before all higher ranked jobs were

started. FCFS tends to produce rather bad schedules concerning objective functions such as the average throughput, but guarantees fairness as the job positions within the waiting queue can only decrease (see [1]).

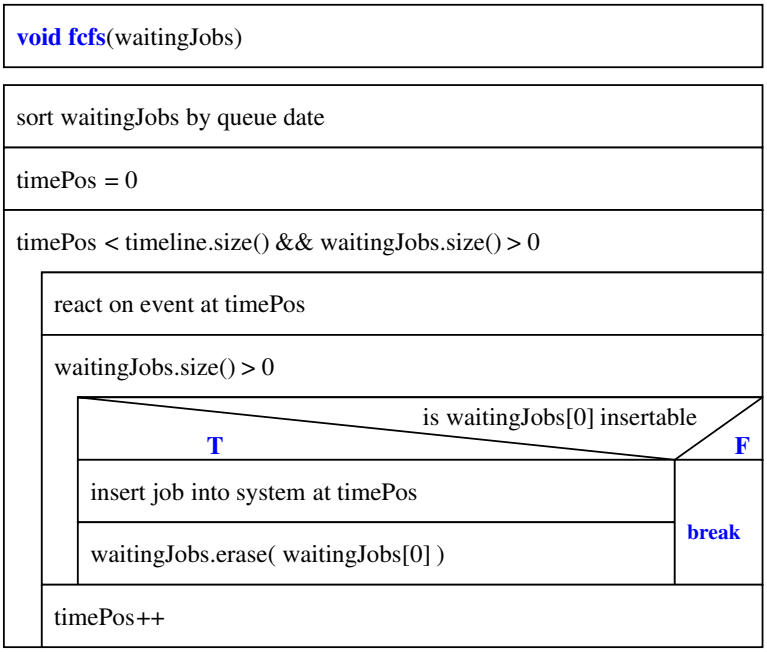

Figure 3.1: pseudo code implementation of FCFS

A possible implementation of FCFS is outlined in figure 3.1. The *Timeline* instance within this diagram consists of events such as the dispatch and completion of jobs, which are changing the system state. Its task is identical to the Timeline included in LLview's prediction implementation SchedSim described in section 2.2. The Timeline is traversed by the outer loop. In each iteration the algorithm reacts on the current event by either inserting a new scheduled job into the system or releasing the resources used by a completed job. The inner loop tries to insert as many jobs from the top of the waiting list as possible. If a job is insertable, it is placed into the current system state and removed from the waiting list.

In order to visualize the analysed scheduling algorithms, a simple example workload is used as input for each algorithm. The most important steps are then simulated with the help of this example. The input system state for this example is defined in table 3.1. Its content is similar to the data passed to JuFo.

All time values – such as start-, end-time and wall clock limit – in this example are given as relative dates in seconds regarding the start of the scheduling. This means, that the running jobs *j2* and *j3* started one hour before the simulation start time. For the FCFS-algorithm waiting jobs are sorted by their queue date, which is the date, when they were submitted. Therefore, the priority attribute listed for the waiting jobs is derived from their queue dates. The priority is calculated as the negated value of seconds between the queue date and the date 1st January 1970. This is the standard time stamp format used in C++ applications. By negating this time difference the job, which submitted the earliest, gets the highest priority. The waiting jobs table presents the jobs sorted by their priority.

resources the system comprises 5 processors, other resource types are neglected in this example

jobs two jobs have been started before the scheduling started, four other jobs are waiting to be allocated

running jobs

Job name	Totalcores	Start time	End time
j2	2	-3600	4
j3	1	-3600	9

waiting jobs

Job name	Totalcores	Priority	Wall clock limit
j5	3	-1333368300	2
j1	2	-1333371840	5
j4	1	-1333371900	7
j6	2	-1333373400	6

Table 3.1: example workload for the scheduling algorithms

The schedule generated by the FCFS algorithm is depicted in figure 3.2. Each job is displayed as a rectangle. Its width shows the time span of the job, the height represents the number of processors used by the job.

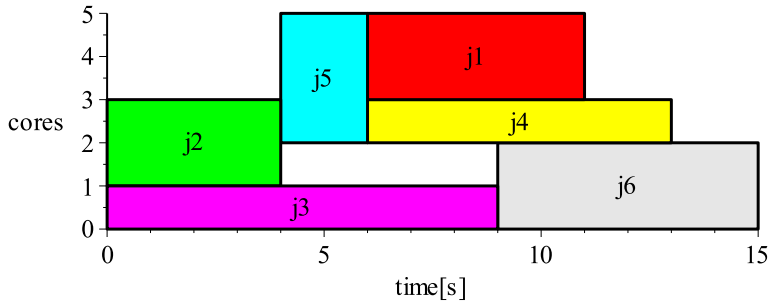

Figure 3.2: results of FCFS scheduling with 3.1 as input

The running jobs are placed at fixed positions determined by a previous run of the scheduler. The waiting jobs list is traversed in the given order and the jobs are placed successively into the schedule as soon as possible. As a result, the job *j5* is started before *j1* and so forth. In this example it is obvious, that the schedule could have been improved by starting *j4* at second 0, for instance. This would have been allowed, since one processor is available throughout this job's duration. However, the FCFS strategy prohibited its start before the higher prioritised *j5*. That causes a relative high number of idling resources.

3.1.2 List-Scheduling

The List-Scheduling algorithm sorts the waiting jobs by arbitrarily configurable priorities calculated from job attributes. A job requests a set of resources such as processors and memory and possesses additional attributes like its queue date, owner, user group or submit

queue. Based on these attributes job priorities can be calculated with simple mathematical terms. For instance the priority could be calculated by $CPU * MEMORY - QUEUEDATE$, whereat CPU means the number of requested processors, $MEMORY$ the quantity of memory in megabytes and $QUEUEDATE$ the queue date of the job in seconds. This term is evaluated for each job and represents its priority. List-Scheduling then traverses the sorted list and starts all jobs, for which sufficient resources are available (see [1]).

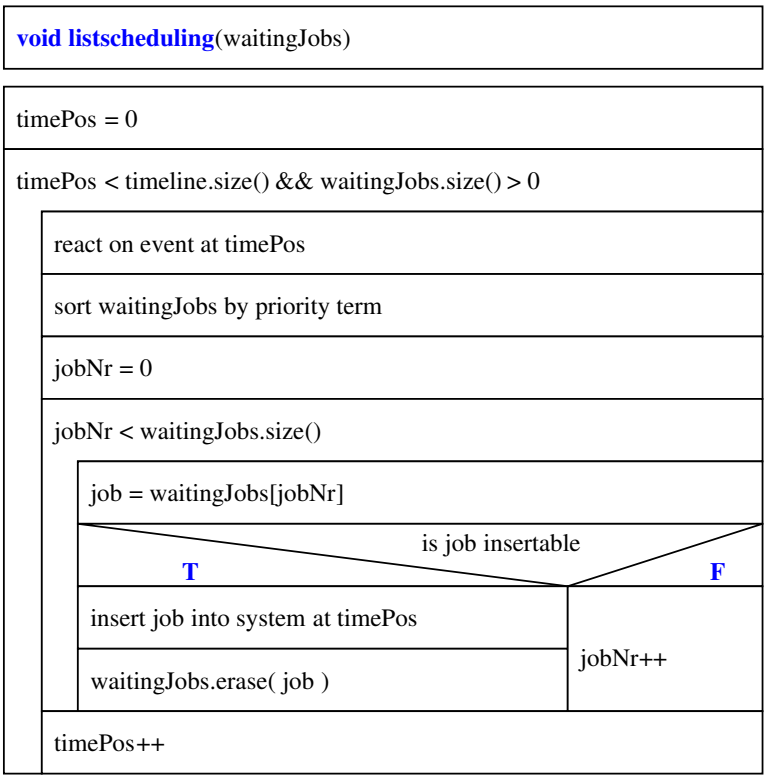

Figure 3.3: pseudo code implementation of List-Scheduling

The resulting algorithm is depicted in figure 3.3. Differences to FCFS are, that the jobs queue has to be resorted in each iteration as the priorities could be time dependant or new jobs could have been queued with higher priorities. Furthermore, the inner loop is not stopped, if the first job cannot be inserted, but tries to insert lower prioritised jobs. As a result, List-Scheduling is more complex than FCFS, but it is assumed to produce schedules with higher throughput as it tries to allocate idling resources to any fitting job. However, the fairness provided by FCFS is not guaranteed with List-Scheduling, because new submitted jobs are able to delay those with an earlier queue date.

With the example input of table 3.1 List-Scheduling would produce the schedule shown in figure 3.4. The priority term is equal to the term used for sorting in the FCFS scheduling. Thus, jobs are simply sorted by their queue date. However, the jobs are started as soon as the requested resources are available. This causes *j1* to be started at time zero, because it is the highest prioritised job, which fits into the idling two processors at the start of scheduling. Even *j4* is dispatched before the actually highest prioritised waiting job *j5*, because it only requests a single processor, which is available as soon as *j2* is completed.

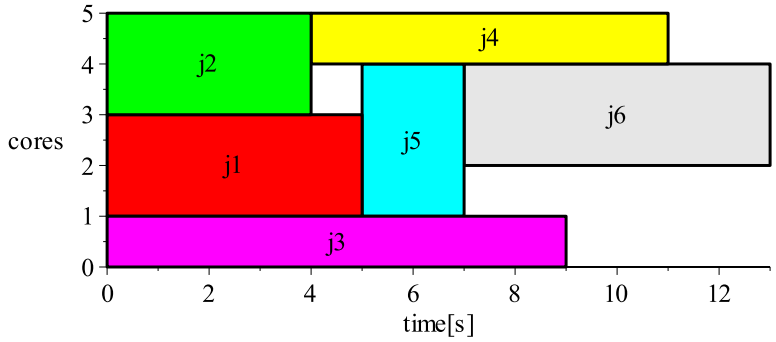

Figure 3.4: results of List-Scheduling with 3.1 as input

3.1.3 Backfilling

Backfilling represents an enhancement of FCFS. Its development is described by Lifka [11]. It tries to use idling resources in the schedules generated by FCFS without delaying higher prioritised jobs. However, the jobs are sorted by a generic priority term like described for the List-Scheduling algorithm. The list is traversed and all matching jobs are inserted. If a job requires more resources than the system is currently offering, the first time slot with sufficient resources is reserved for this job. Thus, lower prioritised jobs are not able to delay the dispatch of this job. But they can be inserted before the reservation in order to use possibly idling resources, which would have been unused in a FCFS schedule.

A possible implementation of backfilling is shown in figure 3.5. It extends the List-Scheduling implementation by the creation and handling of reservations. A second parameter is passed to the backfilling algorithm, which determines the allowed number of reservations. If a job cannot be started within the inner loop, the described time slot is searched and reserved. The job is moved from the waiting to the *reservedJobs* list, so that it is not used for backfilling afterwards. A corresponding event is added to the Timeline. In subsequent iterations of the outer loop the algorithm checks, if a previously reserved job is started. In this case it is removed from the *reservedJobs* list, which allows the creation of another reservation in the inner loop again.

The number of created reservations for the highest prioritised jobs, which cannot be started, leads to the distinction between *easy* and *conservative* backfilling (see [1]).

Easy backfilling

Easy backfilling creates a reservation only for the first job, which cannot be started. This improves the efficiency, but permits the delay of higher prioritised jobs by backfilling inferior jobs. For example the first and second jobs in the waiting list cannot be started due to the lack of resources. Easy backfilling reserves a time slot for the first job and tries to backfill available resources before this reservation. The third job is used for backfilling and might then delay the second job's start. This would be impossible when using conservative backfilling.

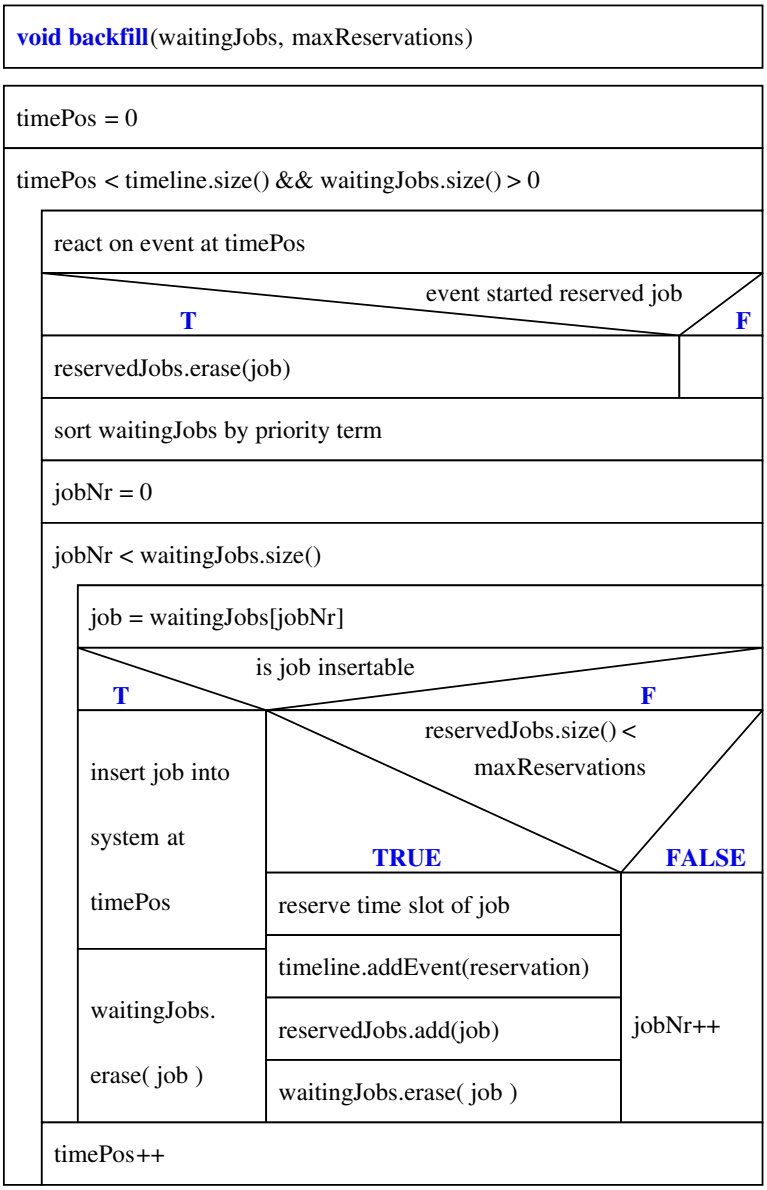

Figure 3.5: pseudo code implementation of generic backfilling

The implementation of easy backfilling would set the *maxReservations* parameter in figure 3.5 to *one*.

Conservative backfilling

Conservative backfilling allows the creation of any number of reservations. As soon as the algorithm detects a job, which cannot be started, the first matching time slot is reserved. Lower prioritised jobs are used for backfilling, but are not allowed to delay the reserved resources. If a lower prioritised job cannot be started with backfilling, the algorithm also reserves a corresponding time slot for this job. Thus, conservative backfilling avoids any delay of a higher ranked job, but leads to a more complex implementation. It is

implemented by setting the *maxReservations* parameter to a higher value than the number of jobs, which are to be inserted, in order to allow the creation of a reservation for each job.

Example

The results of the Easy backfilling algorithm processed on the above example workload are depicted in figure 3.6.

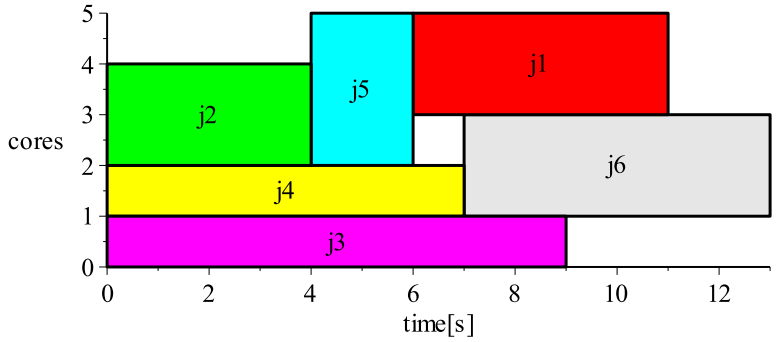

Figure 3.6: results of Easy Backfilling with 3.1 as input

The used type of backfilling allows for creating only one reservation for the highest prioritised job, which cannot be inserted at a given position in the timeline. At second 0 it tries to insert *j5* into the available resources of two processors. Since this job requires three processors, the first future time slot with sufficient resources is reserved. In this example the requested resources are found at second 4 as the completion of *j2* releases two additional processors. After placing the reservation the other jobs are used for backfilling the remaining processors. The backfilled jobs do not need to be finished before the reservation. However, they are not allowed to use any reserved resources throughout their duration. Therefore, searching suitable jobs, which are allowed to be started at a given time, is more difficult than placing jobs in the previous scheduling algorithms. While FCFS and List-Scheduling only have to check at the current time position, whether a job can be inserted into the system, Backfilling also needs to make sure that the inserted job does not collide with any future reservation. For that reason *j1* cannot be inserted at second 0 as it would consume two processors throughout its duration of 5 seconds. This is not allowed, because *j5* would not be able to use its reserved three processors from second 4 to 5. Although sufficient processors are available for *j1* at second 0, its insertion is denied because of the future reservations. The following job *j4* only requires one processor so that it does not interfere the reserved resources and can be used for backfilling the available processor in the time span 0 to 7. As soon as *j5* is actually started, it is removed from the list of reserved jobs, which allows for creating a new reservation for *j1*. In doing so, the scheduler detects the first position, at which the job can be inserted into the system state throughout its entire duration. The same method is applied for the last job *j6*, which can be inserted after the completion of *j4*.

3.2 Practical examples

Although the job scheduler simulator aims to be as configurable as possible, it is important to stay close to real supercomputers and their demands. Implementing configurability is only useful, if a wide range of relevant aspects defining the job scheduler's behaviour can be covered by the given configuration parameters. According to this, the scheduling systems *Loadleveler* and *Moab* are analysed in detail within this section. This allows the extraction of a common basis for scheduling systems. Moreover, the analysed scheduling systems will be used for testing the simulation. As real workload and schedule data can be gathered from the supercomputers JUROPA and JUGENE, which are using the analysed schedulers, these tests can estimate the accuracy and quality of the simulation.

3.2.1 Loadleveler

Loadleveler is used as job management system for the supercomputer JUGENE. The following description of this scheduling system is based on its documentation in [12]. Loadleveler is able to handle job allocations for a set of controlled machines. Its main components are depicted in figure 3.7.

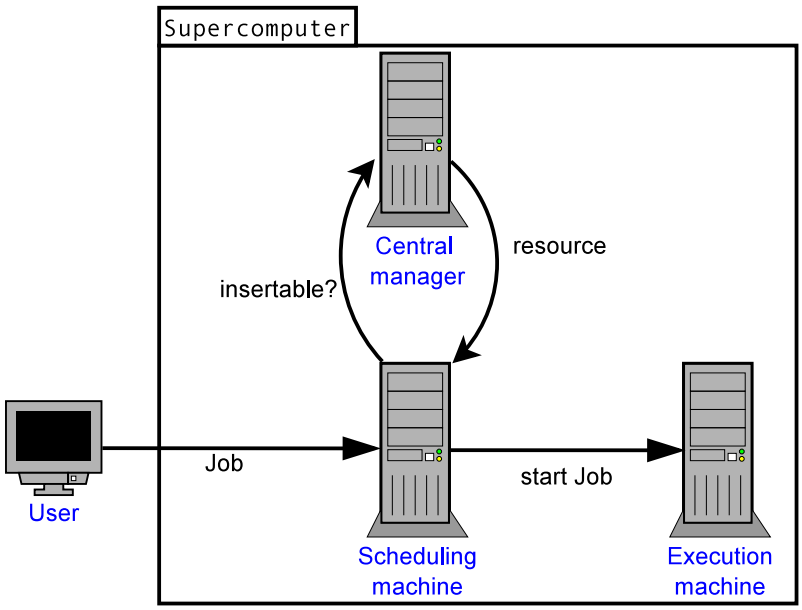

Figure 3.7: components of Loadleveler (compare [12, p. 16])

The supercomputer is in this context abstracted, so that it consists of the central manager, the scheduling and the execution machine. All three components could be placed on a single physical system, although they are separated in this figure. The standard use-case for Loadleveler is shown given by a user, who submits a job to the scheduling machine. This machine is responsible for executing the scheduling algorithm and therefore sorts all submitted jobs by their priority. The central manager monitors the supercomputer resources – such as processors, memory and disk space – and responses to resource requests of the scheduling machine. Resource requests are sent in order to check, whether sufficient resources are available for a submitted job. If the central manager finds fitting resources,

the scheduling machine starts the job on the execution machine. The scheduler simulator has to implement the central manager's and the scheduling machine's algorithms only, as the execution machine does not affect predicted schedules.

Scheduling machine

Loadleveler supports three types of schedulers: `LL_DEFAULT`, `BACKFILL` and invoking an external scheduler. The `BACKFILL` scheduler is the only relevant type for this analysis, because `LL_DEFAULT` is merely applicable to serial jobs and external schedulers cannot be covered completely by JuFo.

The `BACKFILL` scheduler implements a compromise between easy and conservative backfilling. This means, currently waiting jobs are sorted by their priority. The sorted list is traversed starting with the highest priority. For each job the central manager is asked, if there are sufficient resources available. If this is the case, the job is started by the scheduling machine. However, the first jobs, which cannot be started, are marked as *top dogs*. Via the `MAX_TOP_DOGS` attribute the number of jobs treated as top dogs is set. A top dog is placed in the schedule at the first time slot, in which sufficient resources are available. As a result, high prioritised jobs are not delayed by other jobs with less resource requirements.

After the top dog placement the scheduling machine tries to backfill the system with jobs of lower priority by using idle resources available before the top dog executions. These jobs will only be placed, if they do not interrupt the top dog's resource requirements. For this algorithm the scheduling system needs information about the execution times of jobs used for backfilling. This information is provided by the *wall clock limit* specified in the job submission data. As this limit might be imprecise, it is still possible that top dogs are delayed in comparison to a schedule, which would have been generated based on exact execution times (see [12, p. 110-113]). E.g. the running job A defines a wall clock limit of one hour. The top dog requires resources, which are currently used by job A, so that the top dog reservation is placed after the completion of job A. This allows to backfill the remaining resources by another job B, which runs for only 30 minutes. Job A finishes early after 10 minutes. As a result, the top dog is now blocked by the lower prioritised Job B, which would not have been backfilled, if the exact execution time of job A was known.

The job priority is calculated by combining the system and user priority. While the system priority is calculated as a simple function depending for instance on the job class, user group and queue date, the user priority is defined by the user individually as a scalar value in order to sort the jobs of the user by an additional ranking (see [12, p. 230-232]).

Another important aspect, which the scheduling machine has to handle, are reservations. Authorized users are able to reserve compute resources for a given time slot. This is useful for reducing wait times for important tasks and maintenance. Within a reservation only jobs bound to the reservation are submitted. For the scheduling system a reservation is similar to a top dog with highest priority. Other jobs cannot delay a reservation, if they are submitted after the reservation (see [12, p. 25-27]).

The backfill scheduler also supports the preemption of jobs, which is the process of suspending low prioritised jobs to release needed resources for a top dog. As preemption is

not enabled on JUGENE and rarely on other larger massively parallel systems, this option is not further investigated.

Central manager

The central manager determines based on a set of rules, whether a submitted job can be started according to currently available resources. One simple rule is for instance, that there have to be as many nodes available as requested by the job. Any resource requested by a job, which can be used by only one job simultaneously, is called *consumable resource*. Examples are the number of needed processors, memory, disk space and software licenses (see [12, p. 22]). The central manager compares available resources with those requested by a job. It manages the resource changes caused by job starts and completions.

Further rules are given by the concept of job classes. A job class represents a queue, to which jobs are submitted. Each queue might have different scheduling rules. In particular, administrators define for each class the maximum number of concurrently started jobs. This could allow for example to avoid the start of too many jobs with low resource requirements by defining the classes *Small* and *Large*. The administrator could set the maximum starting numbers for the first class to a low value and to a high value for the second class. Therefore, the central manager has to keep track of the number of jobs started for each class and needs to check the constraints for each job request. Next to the classes, setting the `MAX_STARTERS` attribute in the configuration of Loadleveler limits the total number of concurrently started jobs on the entire supercomputer (see [12, p. 55-56]).

Another important aspect when mapping jobs to the compute resources is allocating the communication network. According to Sosa et al. [13] a Blue Gene/P System administers five different networks for the various communication tasks. It provides a three-dimensional torus network for point-to-point operations, a global collective network for communication involving all processors, a global interrupt network, one network for external I/O operations and the control network for monitoring and diagnostic purposes. Since only the torus network is used exclusively by the parallel applications, this network also represents a consumable resource, which limits the number of simultaneously running jobs on the system.

A torus network is defined as a set of point-to-point connections of nodes, which are arranged in cycles for each dimension. As a result, a 1D torus is simply a ring of nodes, while in a 2D torus each node is part of two network cycles. The higher the torus dimension is the more point-to-point connections are installed, which increases the total bandwidth.

Jobs on a Blue Gene/P System request their resources as partitions (see [14]). A partition is a set of midplanes, each of which is a group of 512 compute nodes. Depending on the user request the midplanes can be connected either in a 3D mesh or a torus network. It is also possible to request a partition smaller than a midplane, but this only allows to use a mesh topology. As each sub mesh or torus are dedicated explicitly to the corresponding application, the resource manager has to consider the available partitions before allocating resources to a job. Besides the network topology the user is also able to request the partition's shape by passing the three dimensions of midplanes, which build the partition. Thus, before a job is dispatched the resource manager has to search for a sub partition in the desired shape within the set of all available compute nodes.

3.2.2 Moab

Moab represents another scheduling system used by the JSC for the supercomputer JUROPA. The technical documentation of Moab introduces it as "a highly advanced scheduling and management system designed for clusters, grids, and on-demand/utility computing systems" [15]. The following analysis of Moab is mainly based on this documentation.

Moab is combined with resource managers like Torque, which observe available resources provided by the supercomputers and pass current status information back to Moab. Moab's task is comparable with the scheduling system, which is described as a part of Loadleveler. It prioritises waiting jobs, asks the resource manager, whether a given job can be started according to available resources, and allocates jobs to the resources by sending corresponding requests to the resource manager. The process of scheduling is for Moab divided into the following successive steps: refresh reservations, schedule reserved jobs, schedule priority jobs, backfill jobs, update statistics, update state information and handle user requests (see [15, p. 47]). From these steps the main part of the simulation for Moab can be derived. The first step is inserting active reservations, which must not be interrupted by any other job submission as reservations possess the highest priority for the scheduler. Afterwards the current job priorities are calculated based on many job attributes like the group priority, the number of requested processors or needed memory. Jobs within a reservation are scheduled before normal jobs. The priority of each job determines the scheduling order. Like Loadleveler Moab supports backfilling of idle resources, which can be used by lower prioritised jobs without delaying the start of superior jobs. Another step is updating the status of the compute resources and jobs by requesting the resource manager. The state of available compute nodes as well as the current state of jobs is updated, which for example allows to detect a job completion. Finally, the scheduling system reacts on user requests. Examples for that are users altering reservations or sending new job submissions.

Moab also uses the generic concept of consumable resources, which is introduced in section 3.2.1. The supercomputers provide resources of different types. Job submissions require certain resources. The resource manager keeps track of the available resources and determines, whether current job requirements can be met. For Moab the main consumable resources are: processors, memory (real memory), swap (virtual memory) and disk space (see [15, p. 35]). However, any type of resource might be connected to the system and requested by jobs.

Scheduling algorithm

Moab's scheduling algorithm is comparable with the one used in Loadleveler. It also implements a combination of easy and conservative backfilling (see [15, p. 540-546]). Time slots for the highest prioritised jobs, which cannot be started, are reserved in order to avoid the delay of these jobs. The `RESERVATIONDEPTH` configuration parameter is equivalent to the `MAX_TOP_DOGS` parameter described in section 3.2.1. However, Moab's documentation provides more details about the implemented backfilling strategies. Moab supports among others the backfill policies *FIRSTFIT*, *BESTFIT* and *GREEDY*. After reserving the time slots for the highest priority jobs, *backfill windows* are extracted. A backfill window is defined by a set of concurrently idling nodes and the time span, in which they can be used

for backfilling. For example nodes A, B and C are idling from 1 pm to 2 pm. This defines a backfill window of three nodes covering a time span of one hour. Afterwards node B and C might be reserved for a job, while node A is still idling. Thus, a second backfill window is given by node A with an unlimited time span. As a result, the backfill windows can overlap. The backfill algorithm implemented by Moab iterates over all found backfill windows starting with the window covering the highest number of nodes. For each backfill window fitting jobs are filtered. The *FIRSTFIT* policy implementation simply starts the highest job in the filtered job list. The *BESTFIT* algorithm calculates the *degree of fit* in order to select the most appropriate job. The degree of fit is a function, which takes a job as variable and returns a scalar value. Possible functions are the number of used processors, the product of processors and time span or simply the job time span. The *GREEDY* implementation calculates the sums of the degree of fit of all possible job combinations for the current backfill window and starts all jobs of the maximizing combination.

Resource manager

Moab is capable of interacting with a wide range of resource managers. A resource manager is represented as an encapsulated object, which is responsible for monitoring currently running jobs and nodes. Jobs are consuming resources, while nodes are providing them. Moab communicates with resource managers via well defined interfaces. This interface consists of functions for querying and modifying the scheduling objects *jobs*, *nodes* and *queues* (see [15, p. 705-706]). Moreover, it allows to start and cancel jobs on a given set of nodes. Thus, the following four functions compose the basis for a fully usable resource manager integrated into Moab (see [15, p. 739-740]):

GETJOBINFO requests job information and attributes

GETNODEINFO requests the status of the nodes

STARTJOB starts a job on a set of resources

CANCELJOB cancels a job immediately

A program, which provides this small set of functions, can be configured to be one of Moab's resource managers. The communication interface is extended by additional functions providing more detailed information about the managed resources. This allows the optimization of the generated schedules regarding given objective functions. Based on this interface concept Moab facilitates among others the integration of Loadleveler and Torque as resource managers. As Torque is the currently used resource manager for JUROPA its functionality is summarized in the following section.

Torque

Torque is an open-source resource manager. It manages allocated resources and grants access to them for submitted jobs. Torque monitors the resource attributes categorized into configuration, utilization and node states (see [16, p. 107]). While the configuration category consists of attributes concerning totally available resources such as memory and

processors, the utilization deals with the number of currently used resources. The node state information saves mainly the current status of each node. For instance valid node states are *busy*, *down* or *free*, which mean that the corresponding node is currently occupied by other jobs, detected a system failure or for the third case is waiting for additional jobs.

Examples for resources, which can be requested by a job from Torque, are the maximum value of needed memory, the number and types for nodes, on which the job should be started, and the wall clock limit. Via a resource request of the type *other* passed to Torque system specific resources can be requested (see [16, p. 39-42]). This allows a highly flexible usage of Torque as a resource manager for any kind of shareable resources.

Simulation mode

In addition to running Moab in the default mode, which needs full access to the resource manager and the administrated supercomputer, a simulation mode is provided. This special mode only requires the Moab server to be installed without a running connection to any resource manager and is capable of simulating a real batch system. It allows – similar to JuFo or SchedSim– for testing parameter changes of a real system. However, it is developed especially for Moab and cannot be applied to other scheduling systems. Possible test scenarios for this simulation include adding or removing hardware, increasing the workload by simulating additional users, adapting the scheduling policies and algorithms or changing any configuration of the batch system (see [15, p. 839]). These tests would be of high risk, when executed at supercomputers in production, as they could affect the system's efficiency and stability. Thus, a simulation of the full system in this mode is needed to test crucial changes of the system and in order to predict their impact in advance.

The simulation runs Moab's server encapsulated from the real system. It requires the following three configuration files, which contain all information usually requested from the connected resource manager:

Resource Trace File stores available compute resources and their state changes during the simulation

Workload Trace File provides submitted jobs, which should be executed on the simulated system

moab.cfg contains scheduling policies and the system configuration in the format, which is also used in a production system

While the resource trace file comprises a list of compute node descriptions, the workload trace file specifies events regarding the simulated jobs. Hence, this simulation is event driven, which not only allows for simulating job completions or failures, but also changing the node states from available to drained, for example. Using these events enables to test even unexpected issues such as maintenance, hardware or application failures during the simulated time span. The global configuration file *moab.cfg* can be taken from the setup of a real supercomputer and might be extended afterwards. Alternatively, this file can be written from scratch in order to develop a configuration specifically designed for the

simulation run (see [15, p. 840-841]). As the configuration data format accords with the format used for the normal mode of Moab, successfully tested parameter changes can be integrated easily into the production system.

The simulation is executed interactively via a command line interface. It is started with the three configuration files specified and is usually paused before the first run of the scheduling system. The scheduler is simulated by running it repeatedly after an adjustable interval in the simulation time domain. Thus, it is executed in equidistant time intervals independently from the events, which might be specified in the trace files. The simulation can be paused after each run of the scheduler, which allows for detailed requests of the current system state. These requests are sent to Moab via the normal commands, which would be used on a real system. They provide information about the jobs and configured nodes. Moreover, reasons for the scheduler to not dispatch a queued job can be retrieved and workload statistics can be collected. In addition, it is even possible to change any configuration parameter of the simulated scheduling system interactively. To conclude, this mode provides an entirely configurable simulation of Moab with the possibility to suspend the system at any time in order to check the current state via the same command line interface and applications used in real production.

Chapter 4

Simulation's design

The previous chapter summarises a set of scheduling algorithms, which are often implemented on actual supercomputers. Furthermore, the job schedulers of Loadleveler and Moab are analysed in order to extract their similarities. Based on this theoretical background, the major task of this thesis is to develop a highly configurable job scheduler simulation program named JuFo. The concept of this simulation program and the process of its development is outlined in this chapter. The first section introduces the main idea of the simulation program and presents the most important classes by grouping them into packages and by documenting their interactions. Afterwards, each package is analysed in detail with the help of class diagrams, which ensures the simulation's extensibility. With the classes and data structures described the most relevant algorithms are documented in section 4.3. Finally, the simulation's complexity is examined, the use of LML within JuFo and the possibilities for configuring the simulation parameters are described.

4.1 Overview

The problem of simulating job schedulers can be divided into a set of interacting packages. They provide the basis for JuFo. The major packages of the developed simulation program are depicted in figure 4.1. This figure divides the simulator into eight packages and presents how the packages are connected to each other. Each package lists its most important classes, which are analysed in detail in the following section about *Data structures*.

The task of the simulation can be summarised as parsing the scheduling objects defined in the input LML file, generating a predicted schedule with a given scheduling algorithm and extending the LML file by attributes, which allow for reconstructing the schedule. The *Simulation* package represents the core component. It connects the different packages in a main workflow and implements the scheduling algorithm. The scheduling algorithms are realized by extending the abstract *Simulation* class, which provides a framework for the simulation workflow. The requested algorithm is defined in the LML input file. The *Configuration* class triggers the parsing of this file, extracts relevant parameters, chooses the suitable scheduling algorithm, runs the simulation and generates the output LML. The class structure of SchedSim consisting of the Simulation, SystemState and the Timeline, which are described in section 2.2, is also reflected by the packages of JuFo. However,

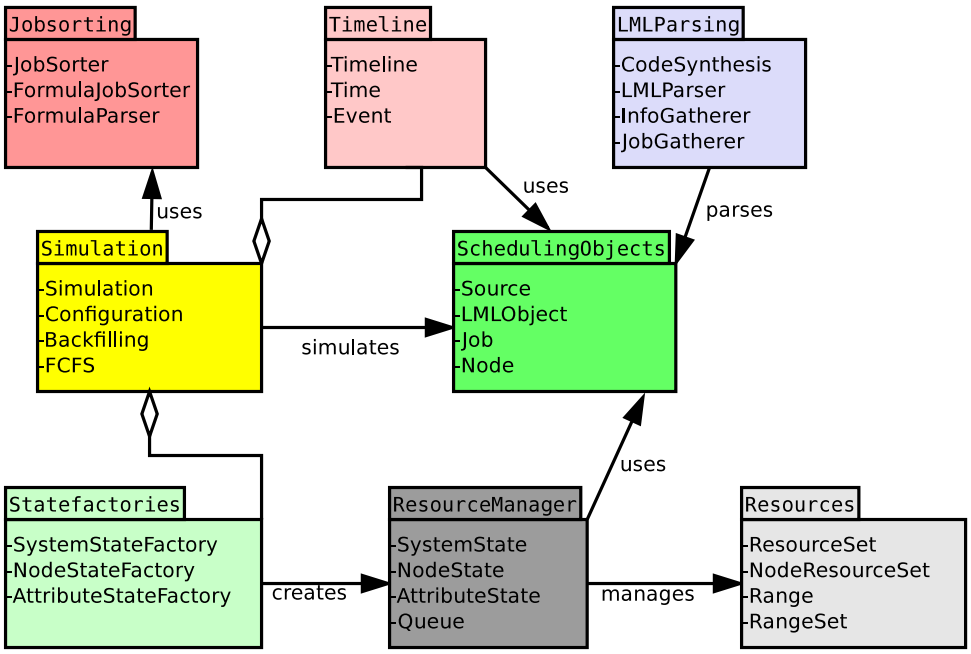

Figure 4.1: package overview of JuFo

each of these former implicitly defined classes is implemented as package in this context so that the range of tasks is subdivided into several classes. This decreases the number of functions covered by each class and simplifies to reconstruct their scopes.

Considering the Timeline design of LLview's prediction module SchedSim and the modelling of jobs and nodes with events in Moab's simulation mode, JuFo also processes events for each activity occurring throughout the simulated time span. Besides the *Timeline* class as a list of events, the *Timeline* package comprises the *Event* class, which collects the type, time and reason of an event, and the *Time* class for abstracting the time format used by the simulation. In addition to a Timeline instance, the simulation requires a *Statefactory*, which creates the *ResourceManager*. While the Simulation class only uses the interface of the abstract *SystemStateFactory*, the Configuration determines the proper factory and passes it to the Simulation. As a result, the Simulation does not know the actual type of the factory and the produced ResourceManager instances, but simply relies on their abstract interfaces. This concept is known as the *Factory Method* design pattern, in which the subclasses of the abstract factory interface decide which concrete product –in this case resource managers– is created (see [17, p.107]). Relying only on the abstract interfaces allows for exchanging the major parts in the simulation framework without changing its implementation. Thus, arbitrary combinations of scheduling algorithms and resource managers are possible.

A ResourceManager simulates resource managers of real supercomputers such as Loadleveler's central manager or Torque. It is responsible for managing available compute resources, for deciding, whether a job can be started at the current state, and for actually placing the jobs into the simulated system. Separating the creation of a resource manager from its use ensures, that the implementation of the different scheduling algorithms in the Simulation package is independent from the type of the resource manager as constructor calls are outsourced into the factory classes. Moreover, the process of converting the parsed

compute resources into the initial state of the resource manager is also moved to the State-factories package. When requesting, if a job can be inserted into the current system state, the resource manager returns a *ResourceSet*. It contains the available resources, on which the job might be dispatched. The *Range* and *RangeSet* classes implement an optimised version of the LML nodedisplay. They allow for storing the exact position of a job on the supercomputer by specifying the used nodes and cores. These classes are used in the *NodeResourceSet*, which extends the ResourceSet class and represents a subset of all nodes provided by the supercomputer. With the help of resource sets it is possible to separate the function for checking, whether a job can be inserted, from the actual insertion of the job. While the former function searches for suitable compute resources and returns them in a ResourceSet, the latter only marks the found resources as occupied by the inserted job.

Since JuFo processes LML data, the *LMLParsing* package is needed to import the scheduling objects such as jobs, nodes and reservations into usable object hierarchies. Several attributes of these objects are adapted or added by the simulation and the results are then converted back into LML by the LMLParsing package. Besides the parsing of LML into objects, the main task of this package is to search for all information tags associated with each object. For each of these objects an *LMLObject* instance is created, which is amongst others extended by the *Job* and *Node* classes. These extensions simplify their utilisation by adding functions for retrieving specific attributes for each object type. E.g. a function is added to the Job class for requesting the number of needed processors. Most of the packages access the generated scheduling objects since they represent the data model for the simulation. The Timeline uses jobs as origins of events, the ResourceManager has to monitor the node states and the Simulation extracts its configuration from a set of LMLObject instances.

Finally, the *Jobsorting* package comprises abstracted classes for sorting the lists of simulated jobs by defined priority criteria. The package is mostly used for sorting the waiting jobs based on priorities defined by the supercomputer's scheduler. Again an abstract interface for sorting a list of jobs is given with the *JobSorter* class, which is implemented by corresponding class extensions such as the *FormulaJobSorter*. However, the scheduling algorithms within the Simulation package make only use of the abstract interface, because this allows to easily switch the actual implementation.

To conclude, the simulation program is based on a set of abstract interfaces, which are designed to allow the implementation of as many different scheduling systems as possible. These interfaces are derived from the common basis of functionality provided by batch systems of actual supercomputer, which are analysed in the previous chapter. Moreover, each abstract class is implemented by at least one reference implementation so that a fully functional simulation can be created by putting together the different implementations. In order to allow changing the implementation behind each interface, the packages have to be strictly encapsulated. E.g. the Jobsorting package must not know or depend on the chosen ResourceManager as this would eliminate the possibility of adding new implementations of the ResourceManager without adapting the Jobsorting classes. As a result, a package is only allowed to call the abstract interfaces of the other packages. By using polymorphism any implementation of the abstract interfaces can be passed to each package. As a result, the design of JuFo especially complies with the following two concepts: "Program to an interface, not an implementation" [17, p.18] and "Favor object composition over class

inheritance" [17, p.20]. The first is applied to the simulator by introducing abstract classes working as interfaces for those parts of the simulation, which are expected to be extended and adapted frequently. The second corresponds with the idea of including objects, which implement these interfaces, into other classes such as the Simulation class in order to use their functionality without depending on their concrete implementation.

4.2 Data structures

After grouping the simulation framework into a set of packages, each of these packages is explained in detail within this section. A class diagram documents each package by depicting the interfaces of its classes and by outlining how the classes interact with each other and with the classes of other packages. At the same time the extension points of JuFo are highlighted in order to support the adaption of the simulation program to new supercomputers. I.e. the meaning and constraints of the abstract interfaces, which are meant to be implemented by possible extensions, are reconstructed with the help of the example implementations.

4.2.1 Simulation

This package builds the main framework for the simulation. The class diagram in figure 4.2 shows the *Configuration* class, whose task is putting together the simulation packages, and the abstract *Simulation* class, which is implemented by the scheduling algorithms FCFS, backfilling and List-Scheduling.

The Configuration is created by passing an LML file, which is converted into an object hierarchy by the *LMLParsing* package. The parsed objects are stored in the *lml* attribute of Configuration. Besides the scheduling objects such as jobs, reservations and nodes the LML file contains all parameters affecting the simulation's behaviour. The parameters specify amongst others the scheduling algorithm, the type of the resource manager, the resources, which have to be managed, and the priority term for sorting waiting jobs before their insertion. These parameters are interpreted by the Configuration class, which subsequently chooses the corresponding implementation classes, calls their constructors and passes the instances to the constructor of the *Simulation* class. E.g. the LML file could specify that the backfilling algorithm should be simulated along with a NodeState as resource manager and the waiting jobs should be sorted by their queue date. This causes the Configuration to create a *FormulaJobSorter* with the priority term $-queuedate$. Moreover, a *NodeStateFactory* is generated by passing the parsed compute nodes to it and both instances are forwarded to the constructor of the *BackfillingSimulation*. As a result, the Configuration class functions as concrete factory for a single Simulation instance.

The frame of all simulations is provided by the abstract *Simulation* class, which runs on a Timeline, the passed JobSorter and the SystemStateFactory for creating the initial system state assigned to the *systemState* attribute. The *simulate* function triggers the insertion of already dispatched jobs and reservations by calling *insertRunningJobs* and *insertReservations*. Afterwards, the most important function *insertWaitingJobs* is called, which is responsible for predicting the dispatch times for the currently queued jobs. Since this

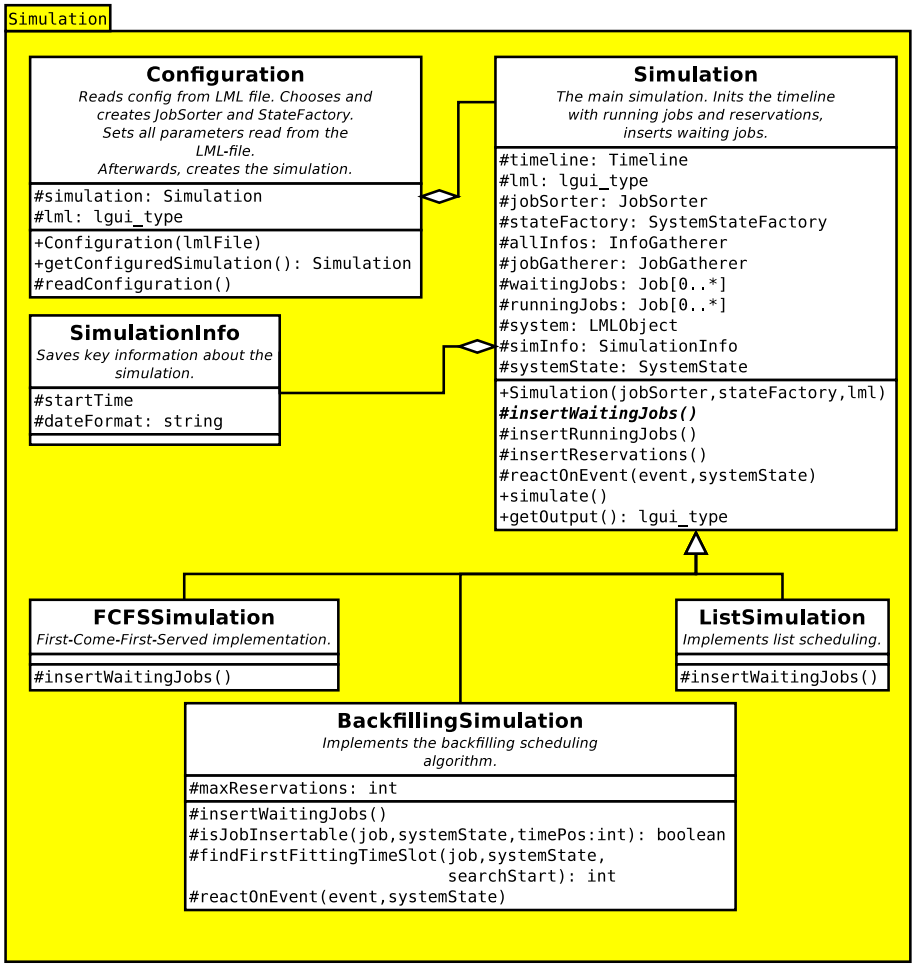

Figure 4.2: class diagram for Simulation package

simulation step depends on the particular scheduling algorithm, the function is declared to be abstract and has to be implemented by the Simulation subclasses. Thus, in order to add another scheduling algorithm either the *Simulation* class itself or any of its subclasses has to be extended. In doing so, basically the *insertWaitingJobs* function has to be overridden just like for the three implemented scheduling algorithms. The *simulate* function implements the *Template Method* design pattern by calling abstract functions such as *insertWaitingJobs*, whose behaviour is defined by the subclasses of Simulation (see [17, p.325]).

However, the extensions are allowed to adjust any simulation step to their needs by also overriding other functions of the *Simulation* class. E.g. the *Backfilling* simulation optimises the *reactOnEvent* function, which applies the changes caused by an event to the passed *systemState*. While the Simulation class simply calls the resource manager's insertion function without any additional information, the Backfilling implementation passes a resource set, which is searched in a previous check for free resources. As the Simulation's algorithm frame is expected to change rarely and its subclasses should mainly adapt the function *insertWaitingJobs*, class inheritance is here preferred to object delegation. This simplifies the adaption of the Simulation class, because subclassing makes the Simulation's internals visible to the implementations of the scheduling algorithms (see [17, p.19]).

A simulation administers one list of running jobs and another for waiting jobs. The first list contains all jobs, which have actually been started in the systemState. Therefore, it might also contain completed jobs, which is reasonable, because all started jobs can be neglected in further simulation steps. The simulation intends to traverse the Timeline in order to find sufficient resources for all waiting jobs so that they can be started and consequently can be moved to the list of running jobs. However, it is possible that not all waiting jobs can be dispatched as the simulated time span or the number of inserted jobs might be restricted in order to limit the run-time of the simulation. As soon as a job is explicitly inserted into the system state, its dispatch and completion times can be saved as job attributes. A job, which lacks these attributes after the simulation finishes, was not inserted. All other jobs then possess an assigned time span and a resource set, on which they can be started. These jobs form the schedule generated by the simulation. The result of the simulation is written into the original object hierarchy parsed from the LML file. It can be accessed with the *getOutput* function.

The classes of the attributes *allInfos* and *jobGatherer* are part of the LMLParsing package. They simplify the handling of the scheduling objects. While the jobGatherer provides functions for retrieving job instances by their IDs, the allInfos instance comprises references to every LML object. One of these LML objects with the type *system* saves the start time of the simulation as well as the date format used for the job queue and dispatch dates. This information is extracted within the Simulation's constructor and passed to each job instance. Thus, the jobs are able to calculate time differences between their queue dates and the current system date so that all time values in the Timeline are saved as time values relative to the start time of the simulation. In order to avoid that each job has to parse this global information about the supercomputer, a *SimulationInfo* instance collects the information and is passed to the jobs.

To sum up, the Configuration produces the Simulation with its needed components for sorting jobs and creating a resource manager. While the Simulation provides an abstract framework, which executes those steps, which are mostly independent from the scheduling algorithm, the subclasses actually implement the behaviour of the scheduling algorithm by focusing on the function for inserting waiting jobs.

4.2.2 Timeline

Since the simulation is event driven, a data structure is required for efficiently handling the occurring events. The *Timeline* package comprises three simple classes for this task, which are displayed in figure 4.3.

An *Event* instance represents a point in the *Timeline*, at which the simulation state changes. Each event is associated with its origin, which causes the event, and a *Time* instance for identifying the event date. The origin is saved in the *source* attribute, while a *Source* is a lightweight superclass for all scheduling objects. Additionally, the event type is stored along with each event. Possible event types are the start and completion of jobs or reservations. Two dummy events are always added to the Timeline for marking the simulation start and end. Other event types such as broken compute nodes or unexpected job failures could also be implemented. Events can be compared to each other in order to sort them chronologically.

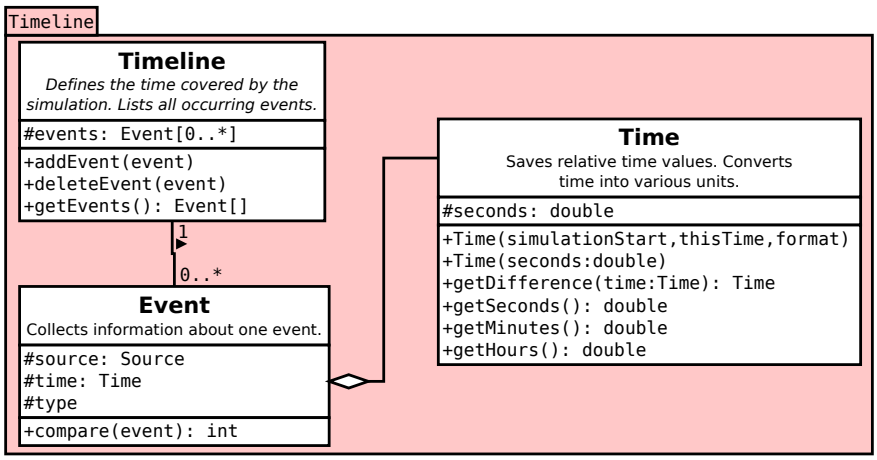

Figure 4.3: class diagram for Timeline package

The Timeline is modelled as sorted list of events. It allows for adding, deleting and retrieving the events. A new event is inserted before the first event, which is evaluated to be bigger than the new event. The main comparison criterion is the time value. However, events with identical time values are sorted according to their event types. E.g. a job completion should always be executed before an insertion so that the resources are released at first and consumed afterwards. Figure 4.4 shows an example Timeline holding all events for two jobs and one reservation. The resulting Timeline contains eight events starting with the simulation start and ending with the simulation end event. Thus, a Timeline allows for reconstructing all actions, which take place throughout the simulated time period. With each event pointing to the corresponding origin a chronological log can be generated by traversing the sorted list of events.

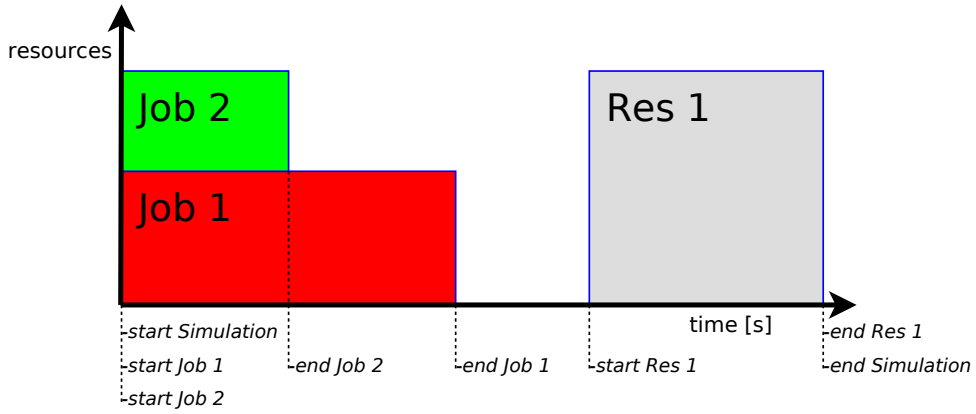

Figure 4.4: example Timeline with possible events

The *Time* class is build into this package in order to add an abstraction layer for time calculations. Internally it uses seconds for all operations such as time comparisons or

subtractions. However, a *Time* value can be set by passing any time unit like seconds, hours or days. Additionally, it is able to parse an absolute date with a given time format and then it calculates the time difference to the simulation start. As a result, all time values in the Timeline are relative to the simulation start, which itself gets the value zero. Except for adding new event types, the Timeline package is not designed to be extended as it only represents a tool for managing and sorting the events of the simulation. The Time class is not only used within this package, but for all time attributes in JuFo.

4.2.3 ResourceManager

The *ResourceManager* package deals with the simulation of the resource managers described in section 3.2. It is especially based on the simple interface required by Moab, which expects a resource manager to only provide functions for retrieving information about the jobs and nodes and for starting and cancelling jobs. Considering the abstract scheduling problem in section 2.1 this package is responsible for monitoring the set of available resources R. It implements the function f for mapping resource requests to actual resources and at the same time ensures the validity of the schedule by assigning only disjunct resources to the jobs like demanded in equation (2.1). The classes implementing this functionality are outlined in figure 4.5.

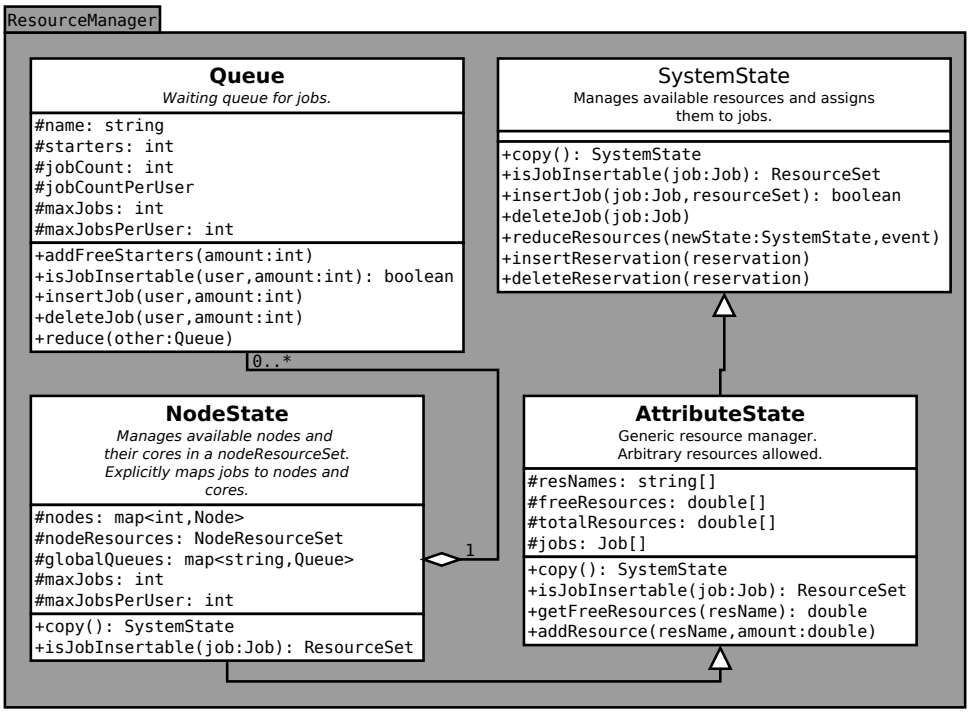

Figure 4.5: class diagram for the ResourceManager package

The *SystemState* is the abstract interface, which should be used by the other packages when interacting with a resource manager. It provides functions for inserting and deleting jobs as well as reservations. Before a job is inserted, the function *isJobInsertable* searches for a currently available set of resources, which are sufficient for the job requirements. If the job cannot be started at the moment, an empty resource set is returned. Otherwise,

the resource set can be passed to the *insertJob* function, which marks these resources to be assigned to the inserted job. In terms of the abstract scheduling problem the check for available resources implements the function call $f(r_j,t)$, which returns a subset of R. The actual insertion has to reduce the available resources by executing $R' = R \setminus f(r_j,t)$, while the deletion of a job reverses this operation with $R = R' \cup f(r_j,t)$. E.g. the set R could hold five available processors $\{1,\ldots,5\}$, a job request for three processors could return $f(3,t) = \{1,2,3\}$. By inserting this job the remaining resources would be $R' = \{4,5\}$. Note that this is only a simple example of a SystemState since the resource managers do not only have to manage processors, but a variety of consumable resources.

Besides simply simulating a resource manager, the SystemState can be used to identify the minimal set of resources available throughout a time span. This is needed for all scheduling algorithms, which use the concept of reservations. A reservation blocks resources for a specific time span. Thus, even if a job can be inserted into the current SystemState, it has to be checked, whether the resources assigned to the job collide with future reservations. The first approach to this problem is to simulate all future events and check for these resource collisions. A more efficient idea is to identify the resources, which are available throughout the entire job duration. Then the job is inserted into these resources only once, while the former approach would have to start from scratch each time a resource collision is detected in the future. In order to find the minimal set of resources a deep copy of the SystemState is generated by calling its *copy* function. By iterating over all future events the original state executes all changes defined by the events, while for the copied state only the *reduceResources* function is called for each event. This function removes all resources, which are marked as unavailable due to the current event. As a result, the copied state only keeps those resources, which are available throughout all simulated events. A similar algorithm is used in the *handleTopDogs* function described in section 2.2.1.

Two SystemState implementations are provided by the classes *AttributeState* and *NodeState*. They have to implement all functions defined in the SystemState, which is only indicated by adding the *isJobInsertable* function to the subclasses in order to keep the class diagram simple. The AttributeState administers a set of resource types, each of which is associated with a number of currently available resources of this type. E.g. an AttributeState could contain the resource types *processors* and *memory* and in its initial state the system could have 10 processors and 5 GB memory, which can be split among the jobs running on the system. Another AttributeState could manage the resources *rooms* and *chairs*, which are used for meetings. Hence, this concept allows for simulating arbitrary consumable resources, which are requested by the jobs. Each resource is identified by a name, which is used as attribute name in the job definitions of the LML file. The attribute values of the jobs specify the number of requested resources of each type. As a result, searching available resources for a job means simply to parse the number of requested resources and to check, whether the system currently offers enough resources for each type. While the insertion of a job decreases the available resource numbers, a deletion increases them. In spite of the generality and simplicity of this approach, it does not always produce valid schedules. Its disadvantage is in the fact, that a job is assigned to a number of resources without specifying, which of the resources are assigned. This problem is visualized with the help of figure 4.6.

The example uses an AttributeState, which only manages processors. Two jobs are already inserted into the schedule with the first job running on processor one and the second on

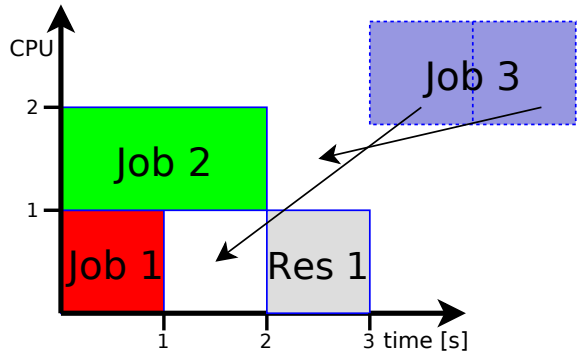

Figure 4.6: schedule example with two jobs and a reservation

processor two. A reservation is included for one processor starting after two seconds. A third job is simulated, which requests one processor for two seconds and which is not allowed to run within the reservation. The simulation has to search the first coherent time span of two seconds, in which one processor is available. This time span is found starting at second one, although the first half offers processor 1 and the second half processor 2. The AttributeState is not able to identify, which of the processors is available, but only determines the number of available processors. As job migration is not allowed, the produced schedule is invalid. This problem is solved by the *NodeState*.

The *NodeState* explicitly manages the states of the compute nodes in its *nodes* attribute. It identifies the nodes by their IDs and assigns subsets of them to the running jobs. Moreover, it is even aware of which cores are allocated to each job, because otherwise the described lack of the AttributeState would just be shifted to the core level. The NodeState extends the AttributeState and uses its functionality for managing the number of available processors globally. If the super class then returns that a job cannot be inserted, because there are not enough free processors, the more complicated tests of the NodeState can be skipped as they cannot be successful. Unfortunately, this implementation looses generality as it expects the simulated system to possess compute nodes so that both implementations are valuable for possible extensions. In addition, identifying the nodes assigned to the jobs highly increases the complexity of the implementation. While the AttributeState models the system with a single numerical value, the NodeState has to manage a list of available nodes with each node containing a list of cores. Moreover, copying a NodeState requires to copy the entire list of nodes, while the AttributeState only has to copy the short list of resource names and numbers.

By using the NodeState each job is assumed to request a number of nodes and a number of cores within each node. Searching a corresponding resource set means to traverse the list of available nodes and to extract the nodes with enough free cores. If a resource set is found, it is saved as a job attribute so that these nodes can be released at the job completion. Due to the increasing number of compute nodes in today's supercomputers –JUROPA comprises more than 3000 nodes– the traversal of the node list for each job insertion can become very time-consuming. A more efficient data structure for mapping jobs to compute resources is given by the nodedisplay included in LML. Therefore, the *Resources* package implements an optimised and more abstract version of the nodedisplay,

which is built into the NodeState with the *nodeResources* attribute. This object allows for efficiently retrieving the set of nodes, which provide the requested number of cores. The explicit list of all nodes is still required for more detailed checks such as the queue constraints, which can be defined individually for each node.

Each job is associated with a queue, which specifies its own scheduling rules. A queue defines in its *starters* attribute how many processors can be used by the jobs in the particular queue. It limits the number of simultaneously running jobs of the same queue with the *maxJobs* value as well as the number of jobs with identical owner defined in *maxJobsPerUser*. The NodeState parses the queue configuration from the node objects. Each node has to manage its own set of allowed queues, while the NodeState accumulates all node queues in a list of global queues. The global queues are used for optimisation similar to the use of the AttributeState's functionality. If the global queue returns that there are not enough available processors of this job queue on the entire system, the expensive iteration over all nodes can be skipped. The functions of the *Queue* are used similar to the corresponding functions of the NodeState. By inserting a job into a queue, the *starters* attribute is decreased by the number of consumed processors, while *jobCount* and the owner specific *jobCountPerUser* value are increased by one. The *reduce* function of the Queue has to be called for all global queues within the NodeState's *reduceResources* function. To each queue the corresponding modified queue is passed, so that it can potentially decrease the number of available starters or increase the number of started jobs. Thus, the queues also contain the minimal set of resources throughout a simulated time span.

4.2.4 Statefactories

The Simulation class makes use of the resource managers, but it is only allowed to access the abstract interface given by the *SystemState* class. Hence, it does not know the class, which actually implements the SystemState. However, the Simulation should be able to create the resource manager at run-time by calling its constructor. Without being aware of the class the creation of the resource managers has to be implemented by an abstract factory. This design pattern separates the use of an object from its creation. As a result the *Statefactories* package is developed. Its classes are shown in figure 4.7.

The design of this package is simple: the interface *SystemStateFactory* is used by the Simulation and it is implemented by the classes *NodeStateFactory* and *AttributeStateFactory*. They provide a single public function for the creation of a resource manager with the corresponding type. The resource manager configuration is extracted from the parsed LML object hierarchy, which is passed to the factory constructors. The AttributeStateFactory just adds the defined cores of all nodes and generates an AttributeState with the resource type *processors* and an initial number of the calculated sum of cores. In possible extensions other consumable resources such as memory or network might be added to the managed resources. This demonstrates another advantage of the factories: in order to add new resource types a new factory can be implemented without the need for adjusting the implementation of the Simulation or of the AttributeState. The configuration only has to switch the factory, which is passed to the Simulation.

The NodeStateFactory takes an *infoGatherer* as parameter, which comprises all nodes listed in the input LML file. Moreover, it parses the queue limits from the scheduler

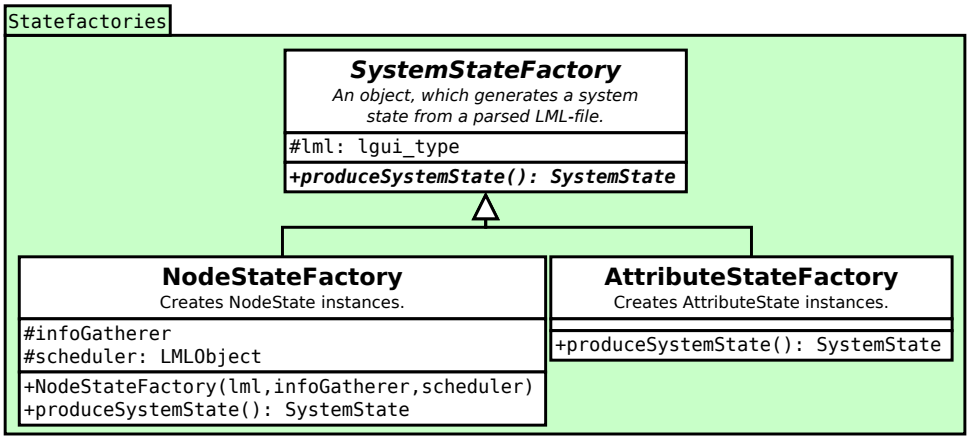

Figure 4.7: class diagram for the Statefactories package

configuration and then calls the appropriate NodeState constructor. If a new resource manager is implemented, it is required to also add a corresponding factory implementation so that the Configuration class is enabled to pass the factory to the Simulation.

4.2.5 Resources

In order to connect the SystemState's functions *isJobInsertable* and *insertJob* a *ResourceSet* needs to be defined. A ResourceSet represents a subset of the available resources managed by a SystemState. Since the resource administration differs for each implementation of the SystemState, each resource manager requires its own type of ResourceSet. While the class ResourceSet is used within the AttributeState, the NodeState deals with instances of the *NodeResourceSet*. Moreover, the classes *Range* and *RangeSet* form the optimised representation of the nodedisplay. They are integrated into the NodeResourceSet so that the compute nodes can be managed more efficiently. The described classes are depicted in figure 4.8.

Each SystemState implementation is aware of the used type of ResourceSet. I.e. the AttributeState knows that all functions are using ResourceSet instances and the NodeState exclusively uses NodeResourceSet instances. This is reasonable, because the resource sets must represent subsets of the actually managed resources so that the NodeState cannot use resource sets produced by an AttributeState. However, the interface of SystemState is implemented by both NodeState and AttributeState. Thus, although the NodeState only creates and uses the NodeResourceSet, its interface defines to use the superclass ResourceSet. The common interface of the resource managers requires the resource sets to also use a common interface. Since the implementation of the resource set for the AttributeState is very simple and at the same time generic, the abstract interface for the resource sets, which is used by the other packages, is directly given by the ResourceSet class. If a new resource manager is implemented, it can either use one of the existing resource sets or has to implement a subclass of the ResourceSet in order to accord with the interface of the SystemState. Internally the resource managers can cast the passed resource sets to their particular type, because they can expect that these resource sets are

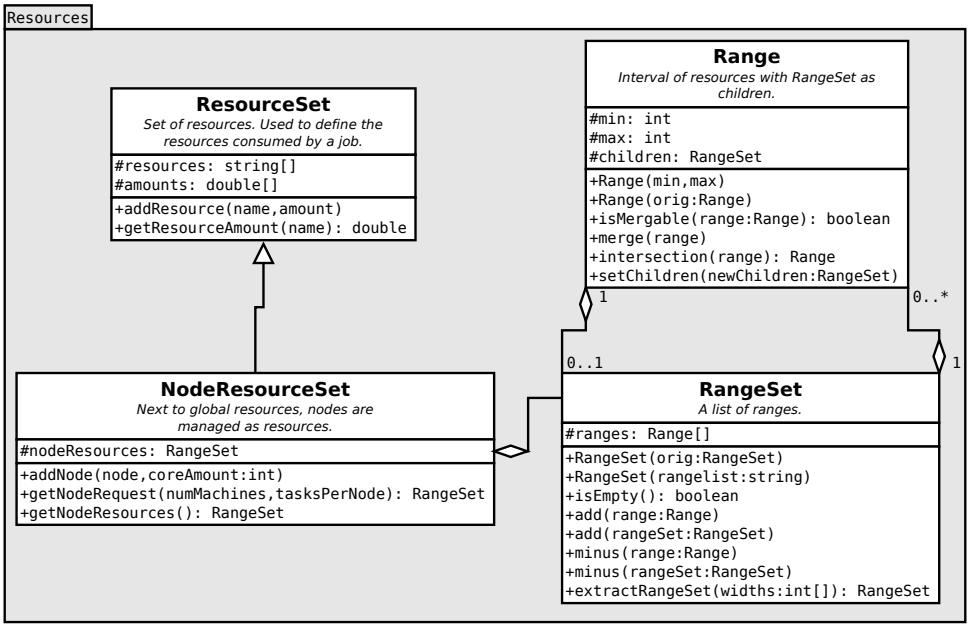

Figure 4.8: class diagram for the Resources package

the results of calling their own function *isJobInsertable*.

The structure of the ResourceSet is comparable to the structure of AttributeState. It stores a list of resource names along with the corresponding numbers of resources requested by a job. One can only add resources to the ResourceSet, because it is merely meant to construct a subset of resources, while the deletion of resources is left to the resource managers' functionality.

The NodeResourceSet extends the ResourceSet by explicitly identifying the nodes and cores of the resource subset. It provides a function for adding a given number of cores within a specific node to its resources. By calling this function for the entire list of nodes, the resource set would contain all configured resources of the simulated supercomputer. The function *getNodeRequest* extracts an available subset of nodes, while each of the nodes provides a number of *tasksPerNode* free cores. This allows for using the NodeResourceSet not only for the subsets of resources used by each job, but also for managing the totally available resources administered by the NodeState.

Instead of collecting the nodes in a simple but inefficient list of IDs, they are modelled as recursive integer intervals with the help of the classes *Range* and *RangeSet*. A Range represents an integer interval from the *min* value till its *max* value inclusively. It can store both node and core ranges. E.g. when using it as node range the range $[1,3]$ includes the nodes with the IDs 1, 2 and 3. Thus, instead of storing a list of 3000 node IDs for JUROPA, the range $[1,3000]$ has the same meaning. If a job requests five of these nodes, its resource set would be $[1,5]$, while the remaining resources can be saved with $[6,3000]$. When the nodes in a resource set do not possess coherent IDs, multiple ranges are needed. E.g. a job could be assigned to the processors 1, 2, 5 and 6. This can be mapped to the range list $[1,2],[5,6]$. A list of ranges is saved in a *RangeSet* instance. As each node comprises a list of processors, the concept of ranges is used recursively: every range saves a RangeSet, which represent its child elements. E.g. for modelling the nodes with IDs 1,

2 and 3, each of which has 2 cores, one could create the node range $[1,3]$ and set its child range set to $[1,2]$. If these compute resources would be modelled as recursive ID lists, the resulting tree would have 6 leaves, while the equivalent range data structure consists of only two nested range objects. This example is depicted in figure 4.9.

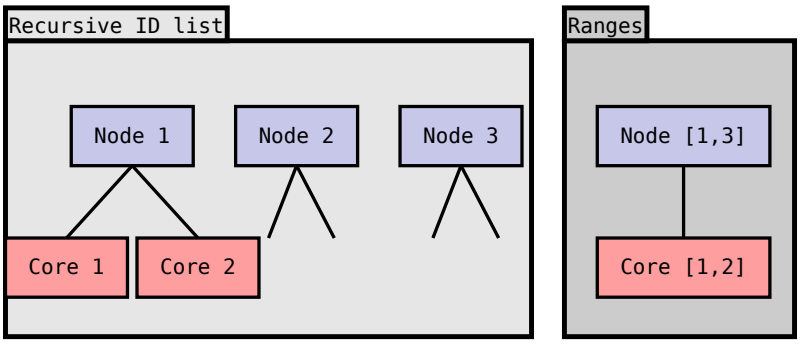

Figure 4.9: comparison of the data structures *recursive list* and *nested ranges*

This idea of replacing ID lists by ranges is borrowed from the data structure used for the nodedisplay in LML. The definition of the nodedisplay and methods for minimising its memory footprint are documented in [10, p.53 – 61]. However, with the Range classes the pure data structure is extended by set operations working on the nested ranges. Like explained in section 4.2.3 the SystemState manages the resources R and executes set operations on it such as calculating differences or unions. As the set R is modelled by a RangeSet, these set operations are implemented with the functions *add* and *minus*. They allow for example to calculate $[1,3] \cup [4,10] = [1,10]$, which complies with a call of the *add* function on the range $[1,3]$ with the passed range $[4,10]$. The Range's *merge* function joins two coherent ranges, if they possess identical child elements. Moreover, the RangeSet provides the *extractRangeSet* function, which returns an available subset of this range set according to a job request. E.g. a job could request two nodes each with 10 cores. By passing the integer array $\{2,10\}$ a corresponding subset is extracted, which fulfils these requirements.

Without the recursive component of the ranges the set operations such as *union* and *difference* can be reduced to the default mathematical set operations. Each range can be mapped to an integer set containing all IDs covered by the range. Then the well defined default set operations are applied to these sets. For the above example calculating $[1,3] \cup [4,10]$ is equivalent to $\{1,2,3\} \cup \{4,\ldots,10\}$. However, this only defines, which results the functions should produce, but does not describe the actual implementation. Based on the ranges it would be very inefficient to convert the ranges into integer sets, then execute the default set operations and afterwards convert the sets back to the ranges data structure. How these operations are implemented for the ranges is documented in section 4.3.

As soon as a range possesses child ranges, the operations become more difficult. The union of two recursive range sets is achieved by the following steps:

1. calculate the intersections of both ranges on the node level

2. split node ranges into intersecting and disjunct parts

3. insert the disjunct parts entirely
4. execute the union recursively on the core level for all intersecting nodes

For calculating the difference of two recursive range sets the last two steps of the union algorithm are replaced by the following steps:

3. execute the difference recursively on the core level for all intersecting nodes
4. remove node ranges, if all core elements are deleted in the previous step

Figure 4.10 shows examples for each of these operations. The upper operation demonstrates the union of two ranges. Their node intersection is $[3,4]$, for which the union has to be executed recursively. The node range $[1,2]$ is not modified, while the disjunct node part $[5,5]$ is entirely inserted as defined in the third step of the algorithm. The example for the difference operation corresponds with the insertion of a job, which requires two nodes each with four cores. The disjunct node ranges $[1,2]$ and $[5,5]$ stay unchanged and the core ranges of the nodes $[3,4]$ are reduced by the core IDs $[1,4]$.

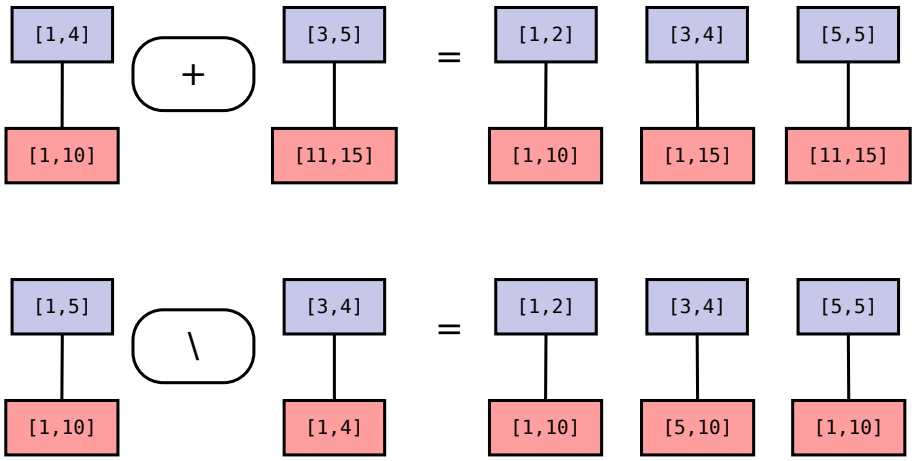

Figure 4.10: examples for recursive range set operations

In order to keep the description simple, the operations are defined for a two level range set with the levels *nodes* and *cores*. However, these operations can be executed in the same manner on arbitrary hierarchies.

4.2.6 LMLParsing

JuFo processes LML files and therefore has to deal with XML. The classes shown in figure 4.11 provide support for parsing LML into object hierarchies, for adapting the parsed objects and for generating the modified output files.

The package makes use of *CodeSynthesis XSD*, which allows for mapping the XML Schema given by LML to a corresponding class hierarchy (see [18]). According to the CodeSynthesis documentation this technique is called *XML Data Binding*. While this project generates

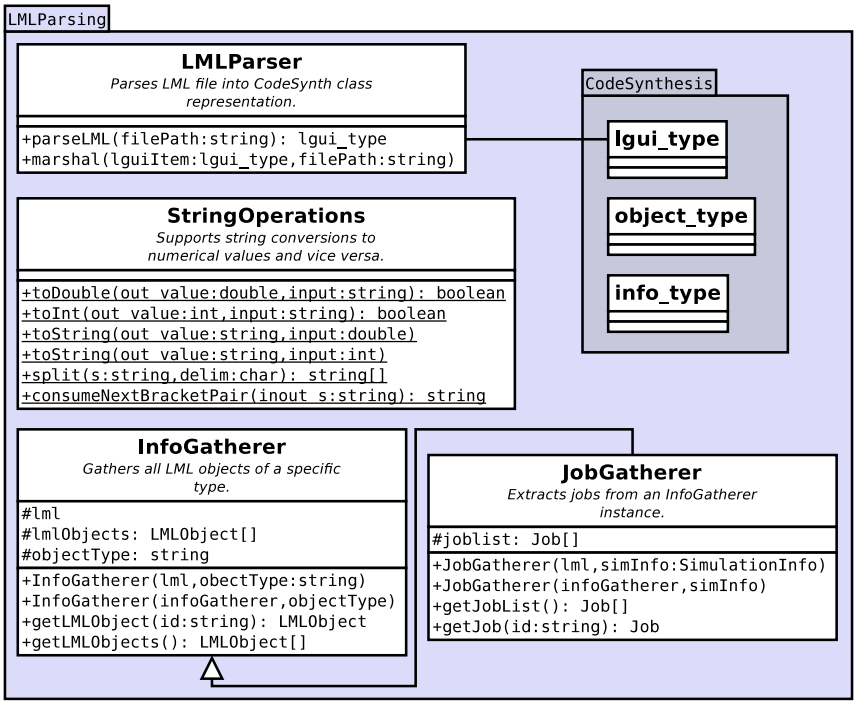

Figure 4.11: class diagram for the LMLParsing package

C++ classes from the XML Schema, the same concept is used by the Java Architecture for XML Binding (JAXB) in order to derive a Java class hierarchy (see [19]). The XML Schema defines the structure of valid XML files. Based on the Schema, classes are derived, which match with the semantic of the XML elements. The XML Binding libraries also include functions for parsing XML files into instances of the generated classes and for serialising these instances into XML files again. Thus, the handling of XML files is encapsulated by these libraries. They produce easy to use object hierarchies, which can be read and modified just like all other objects. Afterwards, the changes can be serialised automatically to the XML format.

The most important classes of the class hierarchy generated by CodeSynthesis XSD are outlined in the CodeSynthesis sub package with the types *lgui_type*, *object_type* and *info_type*. Their names accord with the names of the LML tags. The root element in an LML file is called *lgui*, the scheduling objects are identified by *object* tags and information associated with each object is listed in *info* tags. A simple interface for parsing and serialising LML files is implemented by the *LMLParser* class. *StringOperations* collects functions in class scope for supporting string related operations such as converting string values to numerical values and vice versa.

Finally, the classes *InfoGatherer* and *JobGatherer* use the parsed object hierarchy passed to their constructors in order to generate the scheduling objects, which form the data model for the simulation program. These objects simplify the access to their attributes. E.g. the InfoGatherer composes each object tag along with all corresponding info tags in an *LMLObject* so that all attributes given for an object can be retrieved from this generated instance. The scheduling objects are created in the constructors of these classes and placed into lists, which either can be returned entirely or allow for requesting a specific object by

passing its ID. The upper constructor of the *InfoGatherer* generates the scheduling objects by directly processing the CodeSynthesis object hierarchy. The second constructor is able to copy another InfoGatherer or to filter scheduling objects of a specific type. Thus, the first constructor simply parses all objects of the LML file, while the second could be used to extract the jobs from the LMLObject instances generated by another InfoGatherer. A JobGatherer works similar to the InfoGatherer, but creates instances of the Job class, which requires to convert the mostly used attributes of the jobs into numerical values in order to optimise the access to them. An LMLObject stores all attributes as strings. If this class was used for jobs, each time an attribute is requested as integer value a string conversion would have to be called. As a result, the string conversions are executed once in the Job's constructor and saved as corresponding attributes of each job.

4.2.7 SchedulingObjects

This package contains classes for the basic data model of JuFo. The *Source* class is the super class for all kinds of event origins. An event can be caused by the dispatch of a job or the crash of a compute node, for example. It has the optional attributes *start* and *end* for specifying the time span of the event. The *LMLObject* is a Source parsed from the LML file. It is produced by the InfoGatherer and saves all information about a single object. It provides functions for getting an attribute value as string or numerical value, which is searched in the list of all info tags related to the object.

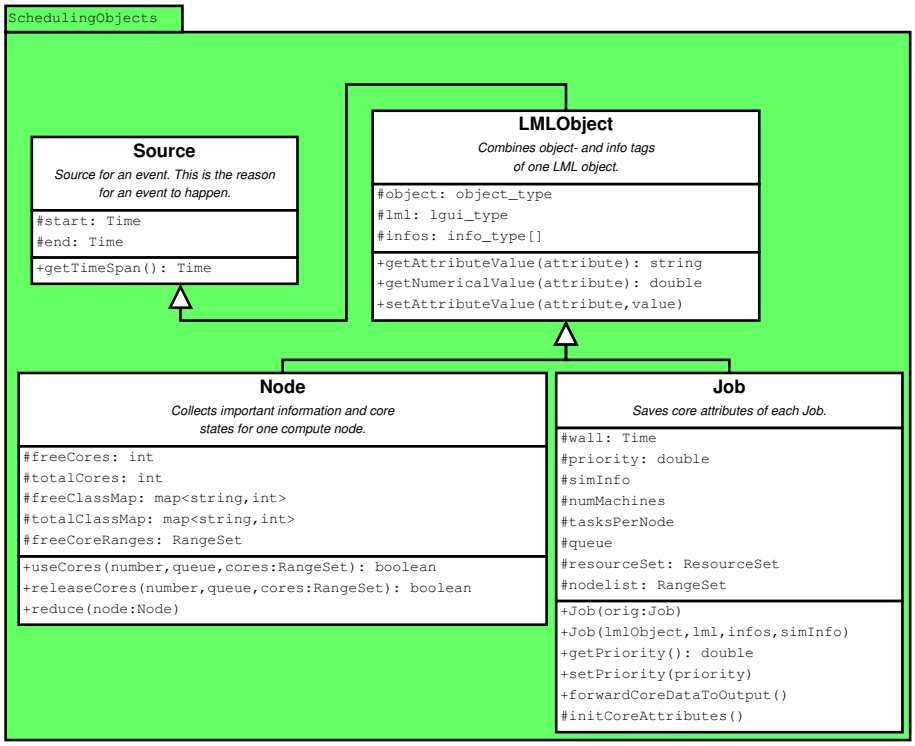

Figure 4.12: class diagram for the SchedulingObjects package

The classes *Node* and *Job* are specialisations of the LMLObject and extend it by attributes and functions, which are needed frequently for each type. The Job adds amongst others attributes for the wall clock limit, the priority and the requested nodes and processors.

These attributes are extracted once in the Job's constructor so that it can work as fast attribute cache for all data, which is used often throughout the simulation. After the simulation finishes, the cached attributes are written back to the CodeSynthesis object hierarchy, which is serialized afterwards.

The Node class is also introduced for optimised access to important attributes. Moreover, it manages the available cores in the range set *freeCoreRanges* and the job queues, which can be defined individually for each node. In doing so the totally allowed processors used by each queue are saved in *totalClassMap*, while the *freeClassMap* holds the currently available processors, which is altered throughout the simulation. If parts of a node are allocated to a job, the function *useCores* has to be called. A job is removed from a node by calling *releaseCores*. The *reduce* function has the same task like the corresponding function of the Queue class in the ResourceManager package. It is needed to get the set of resources, which is available throughout an entire time span.

4.2.8 JobSorting

The *JobSorting* package implements the sorting strategy for a job list. In order to separate the sorting algorithm from the core scheduling algorithm this package is introduced. It allows to exchange the sorting implementation without exchanging the scheduling algorithm itself. The *JobSorter* class defines the abstract interface, which is used by the Simulation and which has to be implemented by the concrete sorting strategies. This package is based on the *Strategy* design pattern, which encapsulates independent algorithms into exchangeable instances implementing a common abstract interface (see [17, p.315]).

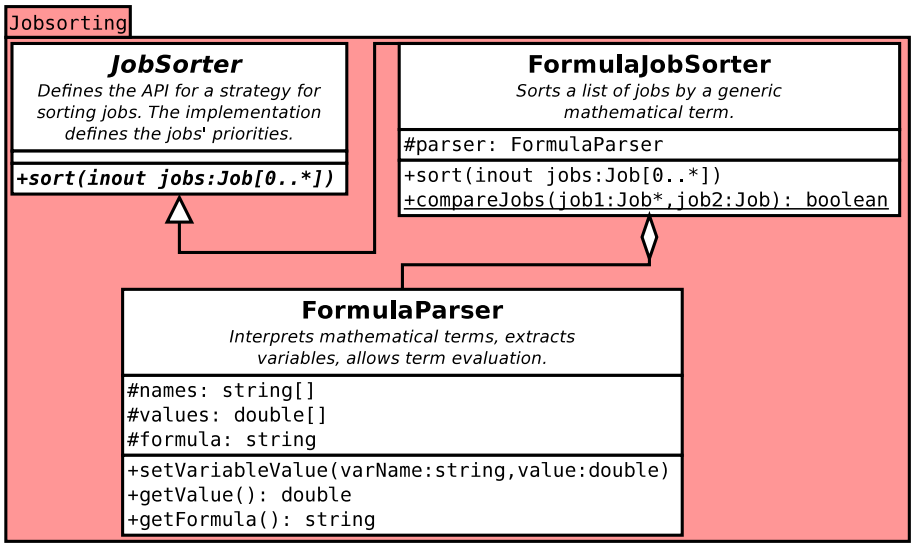

Figure 4.13: class diagram for the JobSorting package

The *FormulaJobSorter* is the only implementation of the *JobSorter* interface so far. It calculates the job priority with the help of a user defined mathematical term. The term contains numerical job attributes as variables and connects them with standard mathematical operators. E.g. the term could be defined as $totalcores * 1000 - queuedate$. Jobs with a high number of requested processors and an early queue date would get high prior-

ity values. The term is evaluated by parsing the job attributes with the names according to the variables of this term. Then the parsed values are inserted into the term and it is evaluated numerically. The result is a scalar, which represents the job priority. Afterwards, the jobs are sorted by their priorities in descending order. The *compareJobs* function returns, whether the first job has to be ranked higher than the second. The *FormulaJobSorter* aggregates the *FormulaParser*, which provides functions for evaluating generic mathematical terms. It parses the formula passed as a string and extracts the variables of the term. Each variable value can be assigned by the *setVariableValue* function. Evaluating the term is executed by calling *getValue*. For each job the FormulaParser has to be filled with the job attributes in order to calculate the job priority. If another sorting algorithm is required, the *JobSorter* interface has to be implemented, the Configuration class has to be aware of the algorithm and has to pass the corresponding instance to the Simulation's constructor.

4.3 Algorithms

While the previous section presents the classes and data structures involved in the simulation, this section focuses on the most important algorithms defining how the various packages interact with each other. At first an overview of the messages exchanged by the major components is illustrated. Afterwards, special features of the implemented backfilling algorithm are presented along with the algorithms for the resource managers and for the range set operations.

4.3.1 Interaction Overview

Although the packages and their classes are documented in detail in the previous section, an explanation of the calling sequences and of the algorithmic internals is still pending. Figure 4.14 depicts a possible sequence of function calls throughout a simulation run on the basis of its most important classes. It does not show every involved class, but helps to roughly understand how they are linked with each other. The Configuration class is designed to be a simple entry point for creating and using a Simulation instance. It is instantiated by passing an LML file. The simulation parameters as well as the scheduling objects, which form the data model for the simulation, are parsed by the Configuration. The configured SystemStateFactory and JobSorter strategy are created and passed to the proper Simulation's constructor. Note that the diagram only uses the abstract interfaces Simulation, StateFactory, SystemState and JobSorter in order to indicate that any concrete implementations of these interfaces can be inserted as the actual instances. The shown simulation's interactions are unaffected by exchanging these implementations except that the scheduling algorithm might adapt the loops in the lower part of the diagram.

The Simulation's constructor calls the StateFactory's function for the production of a SystemState instance, which manages the simulated resources. Additionally, dummy events are added to the Timeline and the job lists are sorted at least once independently from the scheduling algorithm, which implements the *insertWaitingJobs* function.

The created Simulation instance is returned to the *Client*, who triggers the main part of

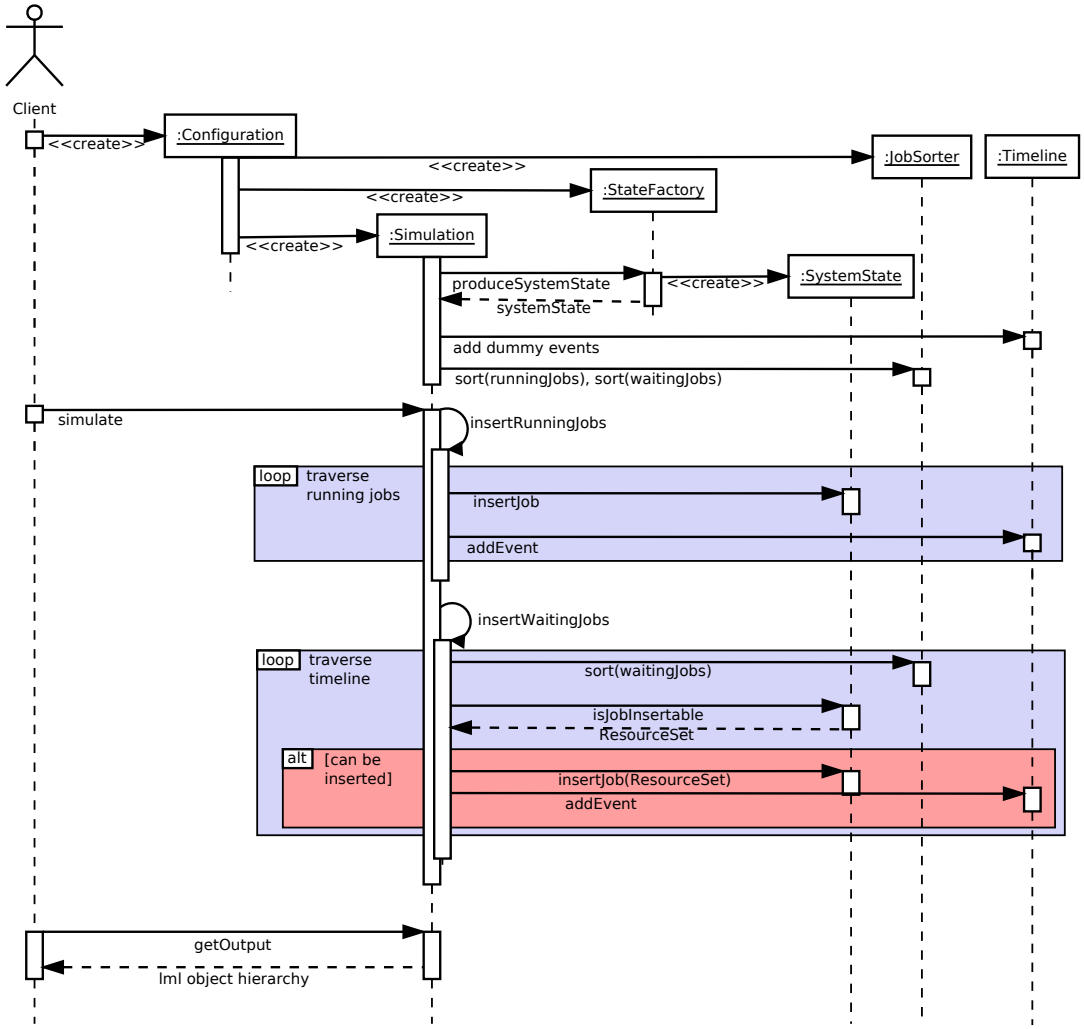

Figure 4.14: overview of interactions in JuFo

the simulation by calling the *simulate* function. Then the Simulation inserts the running jobs in order to initialise the SystemState with its start situation. As the running jobs are required to fit into the available resources, they only need to be inserted without checking for a suitable resource subset. Afterwards, the actual prediction is performed by the *insertWaitingJobs* function. The diagram illustrates the scenario given for the FCFS algorithm. The event list of the Timeline is traversed. In each iteration the waiting jobs are sorted as their attributes and therefore their priorities could change due to the effects of previous events. A second step is searching available resources for the waiting jobs by sending corresponding requests to the SystemState. If a subset of resources is found for a job, it is inserted into the system similar to the insertion of the running jobs. The sequence of function calls within *insertWaitingJobs* depends on the particular scheduling algorithm. However, the possibilities of interaction between Simulation and SystemState are similar for each implementation, as the access to the resource managers is limited to the abstract interface of the SystemState.

After the *simulate* function is finished, the results are written into the object hierarchy

generated by CodeSynthesis. The Client requests these results by calling the *getOutput* function of Simulation so that they can be analysed according to the schedule quality or serialised to the output LML file.

4.3.2 Extension of the backfilling algorithm

Since the scheduling algorithms implemented in the simulation program are analysed in detail in section 3.1, this section only shortly discusses a special part of the backfilling algorithm, which is responsible for extracting the minimal set of resources available throughout the run-time of a job in order to check for resource collisions with future reservations. It is left out in the scheduling algorithm analysis, because it is easier to understand based on the knowledge of the resource manager design. In general, the scheduling algorithms are implemented as described in section 3.1. However, the parts related to the resource managers are suppressed in this analysis in order to focus on the scheduling. Thus, the backfilling algorithm is strongly abbreviated concerning the check, whether a job can be inserted into the system. For this check suitable resources have to be searched not only at the current position in timeline, but for the entire job run time. This algorithm is derived from the corresponding part of the *handleTopDogs* function described in section 2.2.1. The implementation of this algorithm for JuFo is outlined in figure 4.15. It represents a good example to comprehend how the simulation accesses the SystemState as most of its functions are used within this algorithm.

At first, it is tested, whether the job can be inserted into the current position with a call of *isJobInsertable*. If so, the passed resource manager is copied twice. The *minState* is used to hold the minimal set of resources, which is available throughout the entire forward simulation performed in this function. The *tmpState* functions as temporary replacement of *currentState* in order to avoid, that the actual system state of the simulation is changed. The loop traverses the timeline starting at the position after *timePos*. In each iteration the time span, which is simulated so far, is calculated. If it exceeds the wall clock limit, the job can be inserted without any collisions, as it could be inserted into the resources provided by *minState* at any time within its duration. Otherwise, the current event takes place within the job duration. The effects of the event such as consumption of resources by an inserted job are applied to *tmpState* by calling *reactOnEvent*. If this operation removes resources from the *tmpState*, they are also removed from *minState* with a call of *reduceResources*. Afterwards, it is checked, whether the resources stored in *minState* are still sufficient for dispatching the job. As soon as this check fails at any position in the timeline, the function returns that the job cannot be inserted at the given start position. However, if the job is inserted successfully into all simulated system states, the last set of resources, which is found by the *isJobInsertable* function, is saved as a job attribute. It is ensured, that this resource set does not collide with any future reservations. As a result, this resource set can be used to actually insert the job into the system.

On the one hand, this function is used to check, whether a job can be inserted at a certain position in the timeline. On the other hand, creating a reservation for a top dog also makes use of its functionality: the timeline is traversed in a forward simulation until the *isJobInsertable* function returns true for the first time. Then the found time span is reserved for this job.

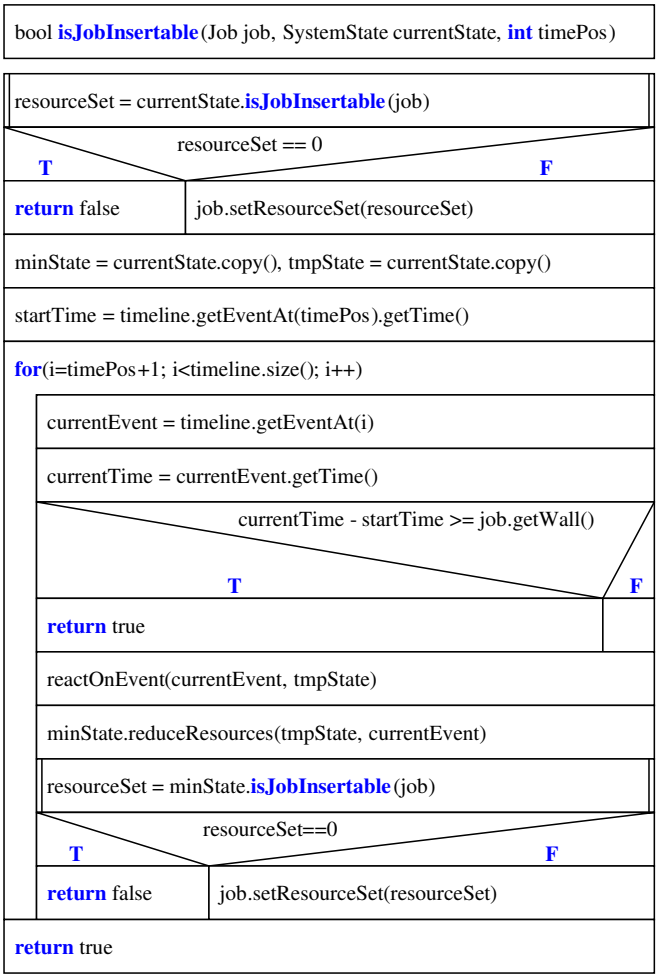

Figure 4.15: algorithm for searching a resource set for a job while avoiding future collisions

4.3.3 Resource manager algorithms

JuFo provides two implementations for the *SystemState* interface. While the *AttributeState* is a simple and generic resource manager for any kind of compute resources, the *NodeState* is explicitly aware of the resources allocated to each job. Due to the simplicity of the AttributeState's implementation this section focuses on the details of the NodeState algorithms.

Like explained in section 4.2.3 a SystemState comprises functions for searching resources according to a job request and for actually inserting and removing a job. The first function named *isJobInsertable* is the most interesting as the others depend on the results of this function and only implement simple modifications to the SystemState's attributes. As a result, *isJobInsertable* is explained comparatively detailed, while the other algorithms are merely summarized.

AttributeState algorithms

The AttributeState resource manager stores a set of resource types in the array *resNames* and for each resource a value expressing the quantity of the particular resource type within its attribute *freeRes*. In this context a resource set for a job request represents a list of used quantities for each resource type. These resource sets are generated by the function *isJobInsertable*. Its implementation is depicted in figure 4.16.

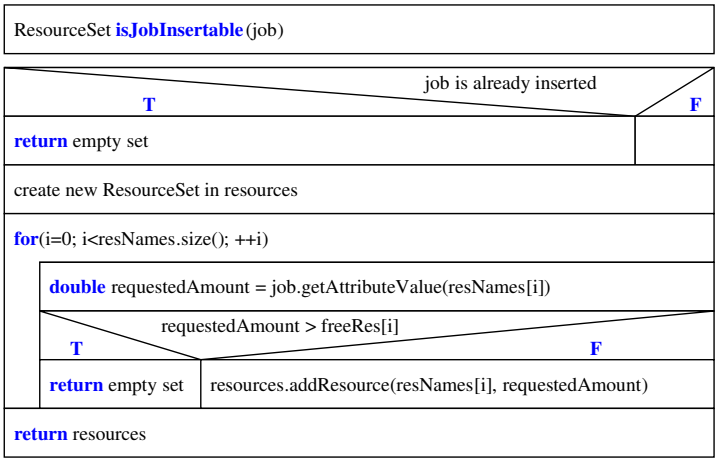

Figure 4.16: *isJobInsertable* implementation of class AttributeState

At first, it checks, whether the passed job was inserted before, in order to avoid multiple insertions of identical jobs. Afterwards, the loop tests for each resource type, if sufficient resources are available according to the job requests. The quantity of each resource type requested by the job is parsed from the LML input file. If the job requests more resources of a specific type than the AttributeState offers, an empty resource set is returned. Otherwise, the list of requested resources is handed back, which is later used by the functions for the actual insertion and deletion of the job.

Inserting a job into an AttributeState means reducing the values of *freeRes* by the requested quantities. By removing a job from the system, the consumed resources are released again.

NodeState algorithms

The NodeState extends the AttributeState's functionality by explicitly identifying the nodes and processors allocated to each job. It also allows for checking constraints defined by the queues of each node. Since the queues are an important tool for configuring the scheduler behaviour, they highly influence the produced schedule so that the resource managers should consider queue constraints before inserting a job. By including these extensions the *isJobInsertable* implementation shown in figure 4.17 becomes more complex and time consuming than the corresponding AttributeState's function.

Initially, the function makes use of its super class AttributeState in order to efficiently check, whether the system globally possesses sufficient processors for the passed job. Then

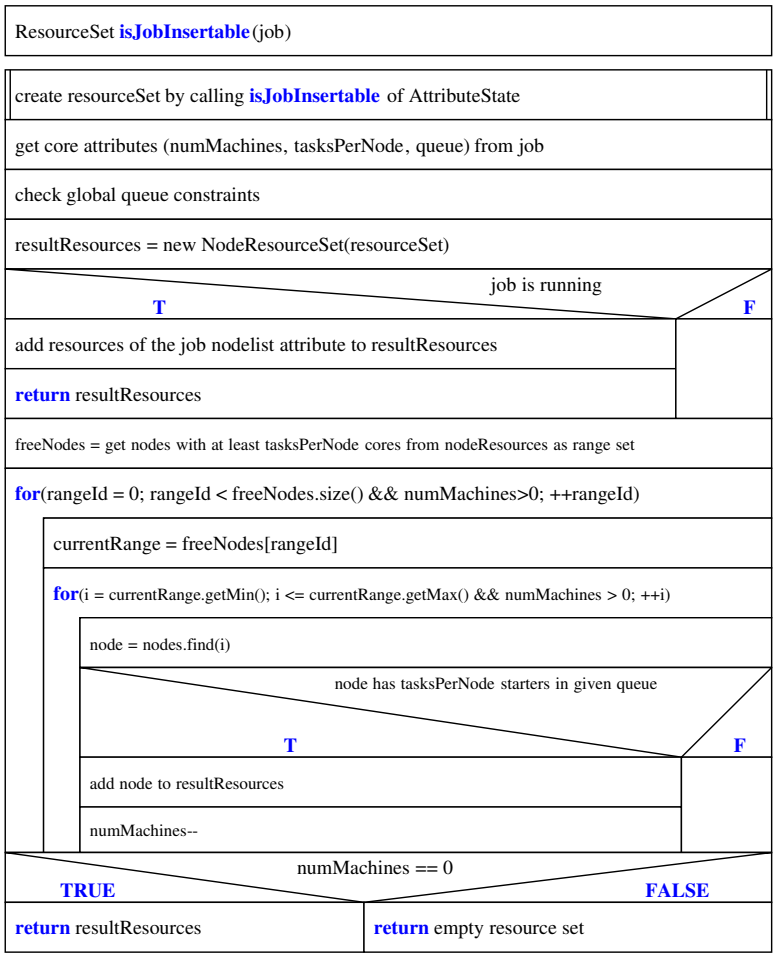

Figure 4.17: *isJobInsertable* implementation of class NodeState

the job attributes such as *numMachines*, *tasksPerNode* and the job queue are retrieved, which form the job request. The global queues stored in the NodeState add up all available processors on the compute nodes for each queue. This allows for a fast check, whether the job can be started. If there are not enough processors available in the global queue, the time consuming traversal of the entire nodes list can be skipped. For a running job, the allocated resources are defined by the input data. Thus, they do not have to be searched.

However, a waiting job requests a number of *numMachines* nodes each having *tasksPerNode* processors. These resources are extracted from the RangeSet stored in the *nodeResources* attribute of the NodeState instance. But, not every node with sufficient available processors is automatically suitable for every job request as nodes could prohibit the execution of a certain queue. Therefore, all nodes with sufficient processors have to be extracted from the range set in order to traverse the nodes list and check the queue constraints for each node individually. An extension of the RangeSet's *extractRangeSet* function, which is explained in the following section, is used to gather all suitable nodes. Then the *freeNodes* variable of type RangeSet contains all nodes, which provide at least *tasksPerNode* free processors. The ranges have to be traversed in the subsequent loop so that the individual queue checks can be executed. While the outer loop traverses all

ranges, the inner loop iterates over each ID within the current range. The node according to each ID is searched in the nodes list and assigned to the *node* variable. If the queue checks are successful for the particular node, it is added to the result resource set. Finally, the function returns this resource set, if sufficient nodes are found. Otherwise an empty set is handed back.

At first glance, this algorithm appears to be more time consuming than simply traversing the entire nodes list. However, as long as the ranges stored in *nodeResources* do not become too fragmented, it is more efficient to iterate over a few node ranges instead of hundreds of single nodes. Especially, when the system is completely filled with jobs, the benefits of using ranges are obvious: the nodeResources do not contain any suitable ranges so that the *extractRangeSet* function quickly returns an empty set. Without the ranges, the nodes list would have to be traversed entirely just to recognize that there is no node with enough processors.

The insertion of a job, for which a resource set is found, consists of the following steps:

1. adapt the global queue by reducing the available processors of the job queue
2. subtract the job resources given as range set from the *nodeResources*
3. traverse all nodes allocated to the job and adapt their queues

In order to delete a job, the inverse operations have to be applied. Since the job resource set is stored within each job, the *deleteJob* function only requires the job instance, which has to be deleted, as parameter.

4.3.4 Range set operations

The range sets are recursive integer intervals. They are used for efficiently storing available resources in a system state as well as resources allocated to a job. Since range sets are similar to mathematical sets, appropriate operations are defined on them. A *union* of two range sets corresponds with releasing a job resource set and adding it back to the available resources saved in the resource manager. The inverse operation is given by calculating the *difference* of two range sets, which represents the consumption of resources due to a job dispatch. In addition, a function is required for *extracting* a subset of resources according to a job request. E.g. a job could request five nodes each having ten processors. This function would search a matching subset or abort, if the requested resources cannot be found. The implementation of these operations is examined in this section.

Range set union framework

The union of two range sets is implemented by the *add* function of the *RangeSet* class. A rough overview of the algorithm's idea is outlined in 4.2.5. The described four steps are reflected by the implementation depicted in diagram 4.18.

Note, that the *ranges* variable contains the list of all ranges for this RangeSet instance, on which this function is called. At first, the highest level of *toAdd* is copied to *notIn-*

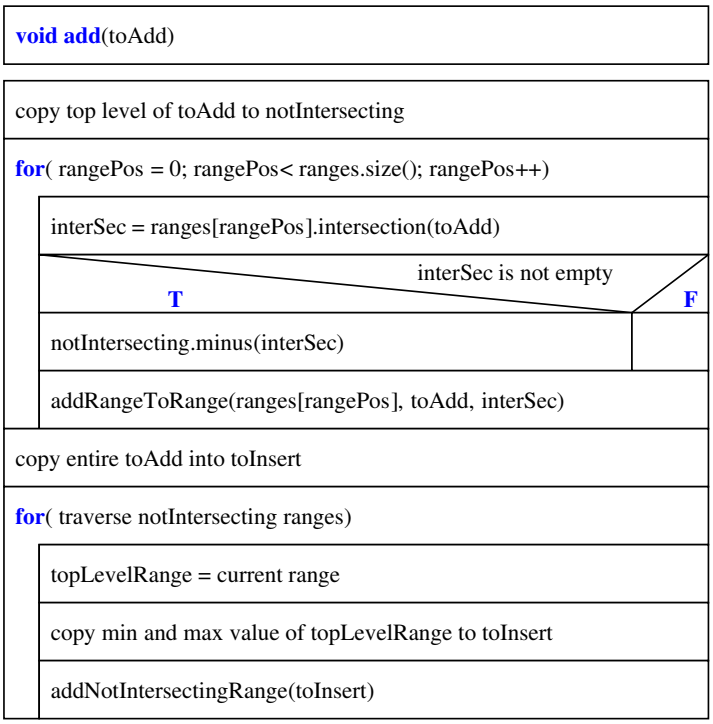

Figure 4.18: framework for the union operation on range sets

tersecting. The first loop traverses all ranges and calculates their intersections with the range *toAdd*. By calling the *minus* function, the intersections are removed from notIntersecting so that it only contains those intervals of *toAdd*, which are disjunct from all existing ranges. Thus, the range *toAdd* is split into intersecting and disjunct intervals. The function call *addRangeToRange* is responsible for calculating the union of the ranges recursively. If *toAdd* intersects with any part of a range in the *ranges* list, its children are added to the children of the range. In doing so, it is possible that the range has to be split, because toAdd might intersect only with parts of each range. After the recursive union the second loop simply inserts the disjunct parts of toAdd. The ranges within notIntersecting are traversed and each range is added to the *ranges* list by calling *addNotIntersectingRange*, which inserts the new range at its appropriate position. The range list is sorted in ascending order according to their *min* attributes.

Recursive range set union

For the intersecting parts the union has to be calculated recursively. E.g. a range set could be used for a two-level hierarchy of nodes and processors. Adding a node set to another requires to unite the processor sets of intersecting node ranges. The function *addRangeToRange* implements this recursive union. A pseudo-code diagram for this function is shown in figure 4.19.

This function only unites the two ranges *orig* and *toAdd* within the range of *orig*. Those range parts of toAdd, which do not overlap with *orig*, are disregarded by this function as they are already handled by the RangeSet's add function. The *addRangeToRange* function

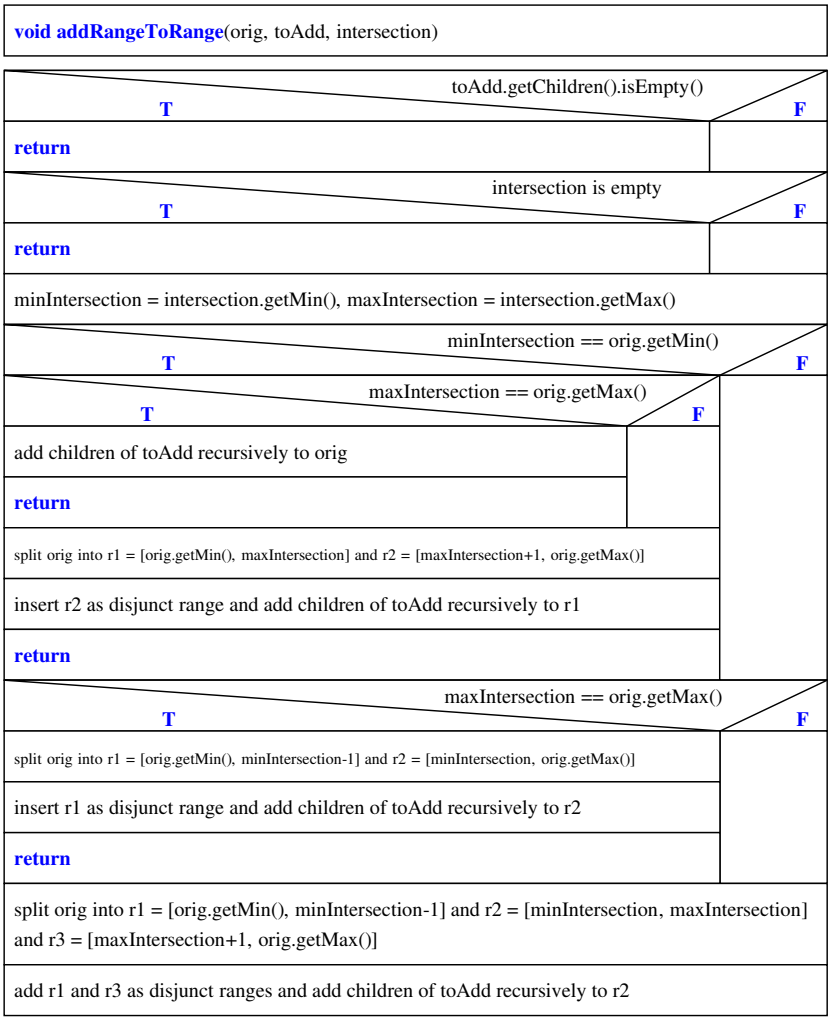

Figure 4.19: algorithm for recursive range set union

has to be called for each range in the RangeSet's list. Their intersection on the top level is passed as parameter. E.g. *orig* could be $[5,7]$, toAdd could be $[1,6]$ so that their intersection is $[5,6]$. Although orig and toAdd might have children, the intersection is calculated only on their top level. Similar to the RangeSet's add function, this algorithm also has to distinguish between intersecting and disjunct parts of the ranges. For the intersection, the add function has to be called recursively. But the disjunct parts are not changed by this operation. As a result, the intersection of these two ranges has to be calculated and the *orig* range needs to be split. E.g. the *orig* range could be $[5,10]$. Then the algorithm has to consider four possible cases:

Case	Example for toAdd	Disjunct parts	Intersection
left intersection	$[1,6]$	$[7,10]$	$[5,6]$
right intersection	$[8,20]$	$[5,7]$	$[8,10]$
subset range	$[8,9]$	$[5,7],[10,10]$	$[8,9]$
identical	$[2,20]$	$\emptyset$	$[5,10]$

Figure 4.20: possible intersections between two ranges

The interval of *toAdd* could overlap with a left sub range of *orig*. Moreover, it could overlap with a right sub range or it could be a real subset of *orig*. Finally, their intersection could cover the entire interval. These possibilities are handled one after another in the *addRangeToRange* function. For the first two cases the *orig* range is split into two ranges, while the *subset* case requires to split it into three parts. If toAdd covers the entire *orig* range, it is not split as there are no disjunct parts. While the children of the intersecting ranges are united recursively with the children of *toAdd*, the disjunct parts keep the children given by *orig*.

The functionality of this algorithm can be retraced with the help of the example union in figure 4.10. It is assumed that the range set with the nodes $[1,4]$ is the instance, on which the *add* function is executed, while the nodes $[3,5]$ are passed to this function as *toAdd* parameter. The first loop of the add function calls the *addRangeToRange* function for these range sets. Their intersection is calculated as $[3,4]$ so that the disjunct part is $[1,2]$, which must not be modified. As a result, *addRangeToRange* detects a *right intersection*, which causes the range $[1,4]$ to be split into $r_1 = [1,2]$ and $r_2 = [3,4]$. The children of the latter are united with the children of *toAdd*, while the children of r_1 stay unchanged. Afterwards, the second loop of the RangeSet's add function inserts the ranges of *toAdd*, which do not intersect with the existing range $[1,4]$. Thus, the range $[5,5]$ is added entirely with its child range $[11,15]$.

Range set difference operator

Since the difference operator is similar to the union in major parts, it is not documented as detailed as the union algorithm. The *difference* operator has to be calculated on a range set, because ranges might need to be split, which is not possible on a single range. Nevertheless, this operator is executed separately for each range in the *ranges* list. The bigger part of the algorithm deals with –similar to the union operator– detecting the type of intersection. Possible types are collected in table 4.20. Afterwards, the current range is split into disjunct and intersecting parts. The children of the intersections are reduced by the children of the subtracted range set. If all children are removed, the parent range instance is also removed from the range set.

This allows to fully understand the second example in figure 4.10. The intersection of the node ranges is $[3,4]$, which is a real subset of range $[1,5]$. Thus it is split into three ranges, whereat only the middle range $[3,4]$ has to execute the difference recursively. The processor set difference $[1,10] \setminus [1,4] = [5,10]$ is calculated. If the result set was empty, the entire middle range would be removed from the range set.

Extract range set

Range sets are used for managing the available resources of a simulated supercomputer. Jobs request parts of these resources. The standard use case of the range sets is a two-level hierarchy of nodes and processors. A job requests a number of nodes each having a number of cores. From a more abstract perspective, an arbitrary hierarchy of range sets is possible. A job can request a number n_1 of resources on the highest level, each of which possesses n_2 children on the second level, and so forth. These requests are handled by

the RangeSet's *extractRangeSet* function. A list of integer numbers containing $[n_1, \ldots, n_N]$, with N as the number of hierarchy levels, is passed to this function. It either returns an empty range, if the resources are not available, or a matching subset of the RangeSet's resources. This function is outlined in figure 4.21.

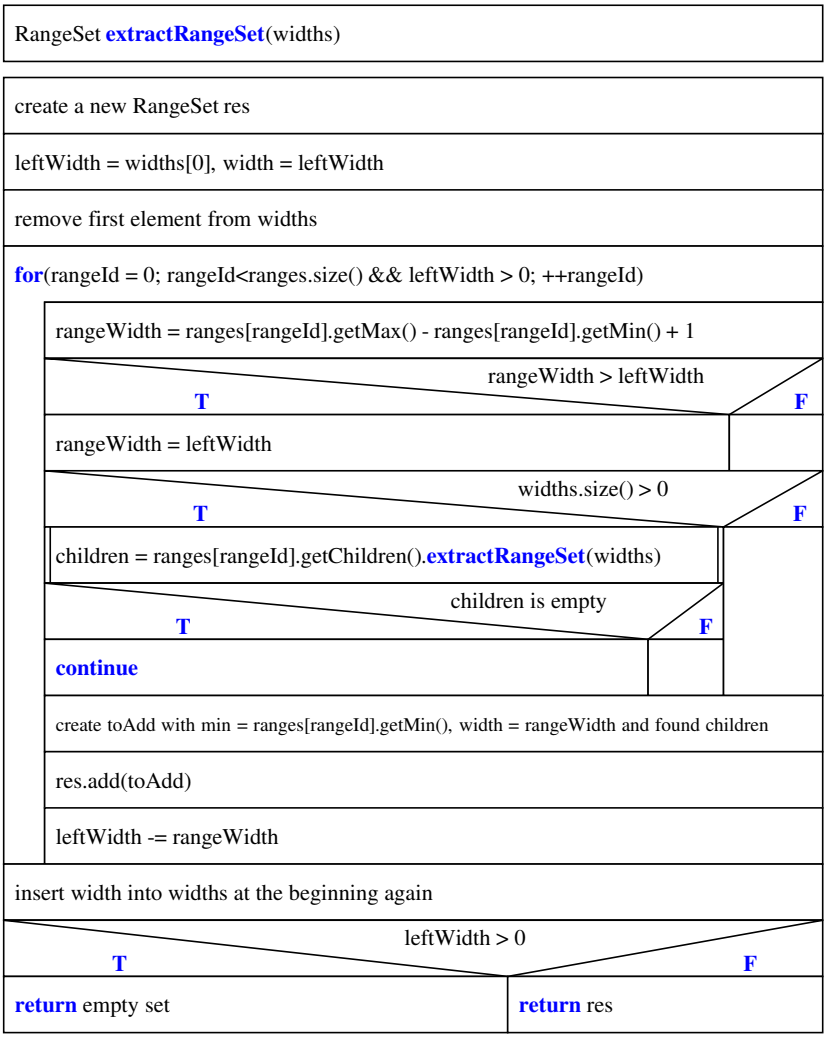

Figure 4.21: algorithm for the *extractRangeSet* function

The function saves the number of elements, which are requested on the current level, in the *leftWidth* variable. The loop traverses the *ranges* list and calculates the number of elements in each range, which is assigned to *rangeWidth*. If *widths* still contains requests for lower level elements, the corresponding child elements are extracted by the recursive call of *extractRangeSet*. If this is successful, the current range is added to the result set and the number of remaining elements, which have to be searched, is reduced by the range's width. It is possible, that not enough elements are found so that the function has to return an empty result set. This indicates, that the job request cannot be fulfilled by this range set.

4.4 Complexity

After simply describing, how the scheduling simulation works, another step in the analysis of JuFo is evaluating its complexity. Therefore, this section identifies the input determinants influencing the simulation run-time.

The simulation program is formed by three major components: sorting of jobs, scheduling algorithm and resource manager. Multiple implementations exist for each of these components, which can be combined arbitrarily. While FCFS, backfilling and List-Scheduling are possible scheduling algorithms, the simulation can choose between NodeState and AttributeState for the resource manager type. As a result, the simulation run-time strongly depends on the combination of these components so that no general formula or upper limit can be extracted. E.g. the number of nodes is crucial for a simulation using the NodeState resource manager, because the node list has to be traversed for each job insertion check. But in an AttributeState nodes are managed as a single scalar value so that increasing the number of nodes does not affect the simulation run-time.

Furthermore, not only the number of jobs and nodes determine the simulation run-time, but also their specific configurations. E.g. a job, which requests many compute resources and thus stays in the waiting job list for many iterations, is more expensive to insert than a job, which has little resource requests and which can be inserted at the beginning of the simulation. In the same manner one would expect the complexity of the simulation using a NodeState to increase with a growing number of nodes as more node instances have to be traversed to search for available resources. But if the number of nodes exceeds the accumulated requested resources of all jobs, all of them can be inserted at the first timeline iteration. As a result, the pure numbers of input scheduling objects is not sufficient for an estimation of the simulation run-time.

At least the number of simulated jobs is a factor, which influences any combination of the simulation components. The core complexity of the implemented scheduling algorithms is given by the following two nested loops: the outer traverses the timeline events and the inner iterates over waiting jobs in order to insert them at the current time position. This loop structure can be found in every scheduling algorithm, even if especially the backfilling algorithm nests further inner loops by searching reservations for top dogs or running forward simulations to check, whether a job can be inserted without future collisions. The number of iterations in the outer timeline loop as well as the number of iterations in the waiting jobs loop directly depend on the number of simulated jobs. The more jobs are simulated, the more events are placed into the timeline and the more jobs are available in the waiting jobs list, which has to be traversed in each timeline iteration. Although the number of waiting jobs might decrease in each timeline iteration the simulation run-time rises in worst case for each scheduling algorithm with the square of the number of simulated jobs.

The list of waiting jobs is sorted at most once in each timeline iteration. Since the order of the jobs is assumed to barely change, the waiting jobs list is nearly-sorted so that for example bubble sort allows to update the sorting with linear effort (see [20]). As the inner loop over waiting jobs, which tries to insert them at each time position, also depends at least linearly on the number of waiting jobs, the sorting of jobs does not affect the overall complexity of the simulation.

To conclude, the simulation complexity cannot be expressed by a simple formula based on the pure numbers of nodes and jobs. The combination of the different components and the wide range of configuration parameters complicate the estimation of the simulation run-time. However, a main structure of at least two nested loops, which depend on the number of jobs, can be found in each scheduling algorithm so that the number of simulated jobs strongly influences the simulation run-time.

4.5 Data format

JuFo uses LML as data format for input as well as output files. Although LML is actually designed to collect status information for supercomputers, which is easy to visualise, it also provides a core data model for identifying the objects described in each LML file. These objects are jobs, nodes, reservations, scheduler and system. LML allows for attaching information as list of key-value pairs to each object. As a result, all attributes of the objects are optional and at the same time any new attribute can be added in order to extend the information basis. However, the simulation program expects a set of core attributes to exist for each object type. E.g. a job should specify the number of requested processors, because otherwise the generation of a meaningful schedule is not possible. A shortened example input file is shown in listing 4.1.

Listing 4.1: LML input example

```
<lml:lgui>
 <objects>
  <object color="#f00" id="j1" name="myjob" type="job"/>
  <object id="node1" type="node"/>
  <object id="sys" type="system"/>
  <object id="sched" name="scheduler" type="scheduler"/>
 </objects>

 <information>
  <info oid="j1" type="short">
   <data key="owner" value="carsten"/>
   <data key="queuedate" value="05/31/12-08:27:00"/>
   <data key="nummachines" value="3"/>
   <data key="tasks pernode" value="8"/>
   <data key="totalcores" value="24"/>
   <data key="wall" value="3600"/>
   <data key="queue" value="medium"/>
  </info>
  <info oid="node1" type="short">
   <data key="ncores" value="8"/>
   <data key="avail_classes" value="(small,4)(medium,8)"/>
  </info>
  <info oid="sched" type="short">
   <data key="system_sysprio" value="-DATEqueuedate"/>
   <data key="system_state" value="NodeState"/>
   <data key="scheduling_algorithm" value="Backfilling"/>
  </info>
  <info oid="sys" type="short">
   <data key="system_time" value="05/31/12-08:30:00"/>
  </info>
 </information>
</lml:lgui>
```

This example specifies four objects and assigns an ID to each of them within the *objects* element. The *information* element attaches one *info* element for each object. Note, that multiple info elements are allowed for each object so that for example a short summary information is provided along with a more detailed version. Each *info* is mapped to the corresponding object via its *oid* attribute referencing the IDs defined in the *objects* element. An *info* element is formed by the mentioned list of key-value pairs. Both attributes *key* and *value* can contain arbitrary strings, which are not liable to any validity constraints. JuFo parses the object attributes and uses default values for possibly missing mandatory attributes. At least the attributes listed in this example should be provided to the simulation program. Otherwise, default values have to be used for this important input data, which might compromise the accuracy of the simulation. A job is required to define an owner, the queue date, the number of requested nodes and processors on each node as well as the total number of needed processors. Moreover, the estimation for the job duration in seconds should be passed with the *wall* attribute and its queue.

The node information collects all consumable resources for the particular node. This usually covers the number of processors and the configured queues. In the example above, the node has eight processors. Four of them can be used by jobs submitted to the queue *small*, while jobs of queue *medium* are allowed to consume all eight processors. These queue configurations are specified separately for each node, which allows for the definition of very flexible scheduling policies.

The scheduler object works as global configuration for the simulation behaviour. It is interpreted by the *Configuration* class, which chooses the actual implementations for the interfaces *JobSorter*, *SystemStateFactory* and the scheduling algorithm implemented by a subclass of *Simulation*. In this example the jobs are sorted by their queue dates. Usually, the *FormulaJobSorter* implementation directly converts the job attributes into numerical values and inserts them into the current priority formula. But the date values cannot be parsed directly into numerical values. They have to be interpreted according to their date format. Then they are converted into scalar values expressing a time difference, which is inserted into the formula. The string *DATE* before the actual attribute name in the priority formula indicates, that this variable is given in date format. The type of resource manager, which must be used by JuFo, is specified in the *system_state* attribute. Currently allowed values are *NodeState* and *AttributeState*. Finally, the scheduling algorithm is chosen by the corresponding attribute of the scheduler object. Valid algorithms are *FCFS*, *List* and *Backfilling*. If JuFo is extended by an additional implementation for sorting of jobs, resource managers or scheduling algorithms merely the *Configuration* class has to be adapted. It has to check for the new attribute values and pass the appropriate implementation to the *Simulation* class.

The *system* object contains general information about the simulated supercomputer. Here, it only defines the current system date in order to set the job queue and dispatch dates in relation to a fixed start date of the simulation.

This input file is processed by JuFo. It adds attributes to the job information, which contain the estimated dispatch time or reasons, why a job cannot be started with the given configuration. For jobs, which can be started, the allocated resource set is also written to the output file. This resource set depends on the resource manager type: for a *NodeState* the actual IDs of allocated nodes and processors are returned, while an *AttributeState* is only able to return the numbers of consumed resources. To conclude, the output LML file

is equal to the input file except that a well defined set of additional attributes extend the job information.

This core part of LML is very flexible and extensible. It only defines the allowed types of scheduling objects, but does not restrict the information attached to each object. Therefore, any information, which might influence the behaviour of the scheduler, can be passed to the simulation. New implementations of the simulation's extension points are simply integrated as new attribute values within the scheduler object, which does not require any modification of the data format. However, the same flexibility given by this data format is also required for JuFo. It cannot rely on the existence of mandatory attributes and has to check the validity of the input data before starting the simulation. Moreover, reasonable default values have to be defined in JuFo for possibly missing job or node attributes. On the one hand, this loose data format allows for flexible modification and extension of passed data. On the other hand, semantic validity checks are shifted from the LML Schema definition to the simulation program.

Chapter 5

Optimisation

The previous chapter documents in detail the structure of JuFo, outlines its components and their interactions and analyses the major algorithms, which form the basic implementation of the simulation framework. While this analysis covers the required design goals extensibility and abstraction, the third major target for this simulation program is efficiency. The first two goals are achieved by the generic and solid component architecture explained in the previous chapter. However, in order to enhance the efficiency of the simulation the algorithms have to be investigated in detail so that possible performance hot spots are acquired. This chapter extracts the crucial parts of the existing algorithms and provides approaches for decreasing the simulation's run-time by simultaneously keeping its functionality identical. At first, ideas for improving the serial algorithms are outlined, while the last section analyses how to parallelise the simulation.

5.1 Similar job requests

A simple idea for optimising the standard algorithm described in the previous chapter is given by the assumption that users often submit similar job requests. Jobs are considered as similar if they request similar resources. More precisely if the job insertion check fails for a job, it will also fail for all similar jobs. Instead of checking, if a job can be inserted, separately for each job, those jobs, which are similar to other currently blocked jobs, can skip this expensive test. For similar jobs their most important parameters such as the job owner, its queue, the number of requested processors and the wall clock limit are equal.

Moreover, users can submit consecutive job requests, which depend on each others completion. In doing so, a user could send a set of identical job requests enumerated by step numbers. Each job has to wait for all jobs with lower step numbers to be completed. As a result, all jobs depending on another job request, which cannot be started at the current position of the timeline, are also blocked from starting.

The generic scheduling algorithms of the previous chapter do not consider these assumptions and still check each job request separately. Instead of that, in each timeline iteration a list of all jobs, which cannot be started, has to be stored in order to compare subsequent jobs with this list for similarity. If another job is similar to one of the jobs in this

list, the expensive forward simulation can be skipped. Since checking for similarity is a fast operation, this optimisation reduces the number of these forward simulations with little additional effort. Its implementation can be inserted easily into existing scheduling algorithms. However, the definition of similarity could differ for each batch system and resource manager type. If jobs are misleadingly evaluated to be similar, it is possible that jobs are delayed without cause. As a result, it is recommended to rather use too many job parameters, which have to accord, than block misleadingly similar jobs from starting. But, the more parameters are included in the similarity check the less jobs will be evaluated to be similar, which decreases the positive effect of this optimisation. Thus, the best solution is to include as few parameters as possible, but sufficient to avoid incorrect similarity tests.

5.2 Handling simultaneous events

JuFo is event driven and manages all occurring events such as job starts or completions with a timeline, which sorts these events chronologically. The scheduling algorithms traverse these events one after another, execute their effects like consuming or releasing resources in the resource manager and add future events for inserted or reserved jobs. The presented algorithms execute each event separately and try to dispatch waiting jobs afterwards. For the backfilling algorithm a job insertion requires a forward simulation, which executes all future events within the current job duration in order to check for possible resource collisions. If many events occur at identical time, these forward simulations conduct the same events repeatedly. Assuming that all events at a specific time consume all available compute resources, these forward simulations are run needlessly as any job insertion at that time will fail. This problem is reconstructed with the help of the example timeline depicted in figure 5.1.

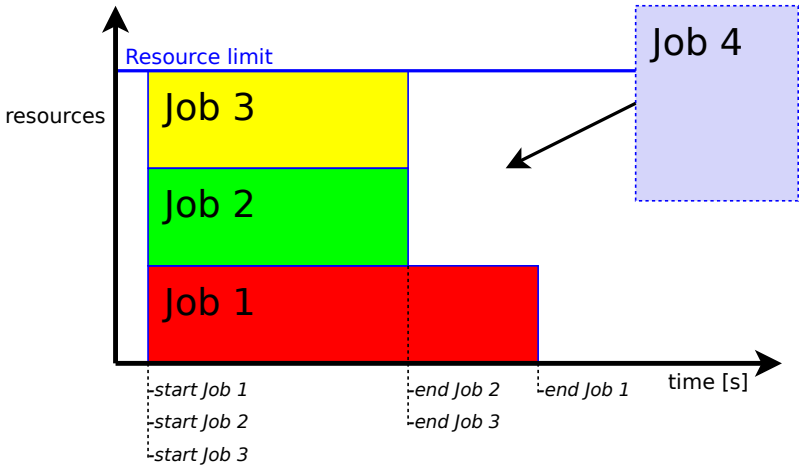

Figure 5.1: example timeline with simultaneous events

This timeline shows three running and one waiting job. All running jobs are started at the same time. The running jobs are assumed to consume all available compute resources so that *job 4* has to wait for the completion of *job 2* and *job 3*, which release the needed resources. By executing only one event in each timeline iteration, a forward simulation is run for job 4 for each job insertion event. I.e. after the insertion of *job 1* the algorithm

tries to add job 4 to the timeline, which fails since the forward simulation will detect a collision with job 2 and 3. The same happens after the actual insertion timeline point of job 2 and 3 in following timeline iterations. The obvious solution for this inefficiency is to execute all events with identical time values at once. As a result, all running jobs would be inserted within a single timeline iteration and the number of forward simulations for job 4 would decrease from three to one. In the same manner the simultaneous job completions for job 2 and 3 would be executed at once, which also reduces the number of insertion checks for job 4.

To conclude, JuFo can be optimised by executing all events with identical time values at once instead of iterating over every single event. This can reduce the number of forward simulations, because the total number of timeline iterations, in which the scheduling algorithms try to insert waiting jobs, is decreased. However, this optimisation might alter the resulting schedule. E.g. without this optimisation another waiting job 5 requesting the same number of processors as job 2 could delay the higher prioritised job 4. At first only job 2 would be removed from the resource manager so that there are not sufficient resources available for job 4. Assuming that job 4 is no top dog, job 5 can be started before the completion of job 3. When executing the job completions of job 2 and 3 at once, job 4 could be dispatched without being delayed by smaller jobs. As a result, this optimisation favours the dispatch of higher prioritised jobs with comparatively large resource requests. This is beneficial since the preference of large jobs often accords with the scheduling objectives for supercomputers. Large jobs have to be saved from starvation, which is achieved by avoiding the fragmentation of the system with small jobs. E.g. Moab addresses this policy with the opportunity to enable "Backfill chunking" [15, p.545], which allows to gather released resources for jobs larger than a configured size in order to reduce fragmentation. Note that the optimisation described in this section only influences the produced schedule, if events occur at identical times, which is probably a rare situation for real workload examples.

5.3 Backfill windows

The following optimisation can be applied merely to the backfilling algorithm. Since this algorithm represents the most complex and practically most relevant of the implemented scheduling algorithms, it is reasonable to especially focus on its optimisation. So far, backfilling is implemented by iterating over the timeline events and conducting a forward simulation for each waiting job. This causes that identical events are simulated multiple times without sharing the information generated by previous forward simulations. The introduction of backfill windows, which are also mentioned in the description of Moab's backfilling solution in section 3.2.2, allows for sharing the information gathered in a single forward simulation among the insertion checks for all currently waiting jobs. This approach strongly reduces the number of forward simulations without changing the resulting schedule.

A backfill window is defined by a set of compute resources together with a time span, in which these resources are available. E.g. ten processors available for one hour define a backfill window. They are used to model free resources before future reservations for top dogs. These resources can be backfilled by smaller jobs, which do not interfere with top

dog reservations. Figure 5.2 depicts backfill windows in an example schedule.

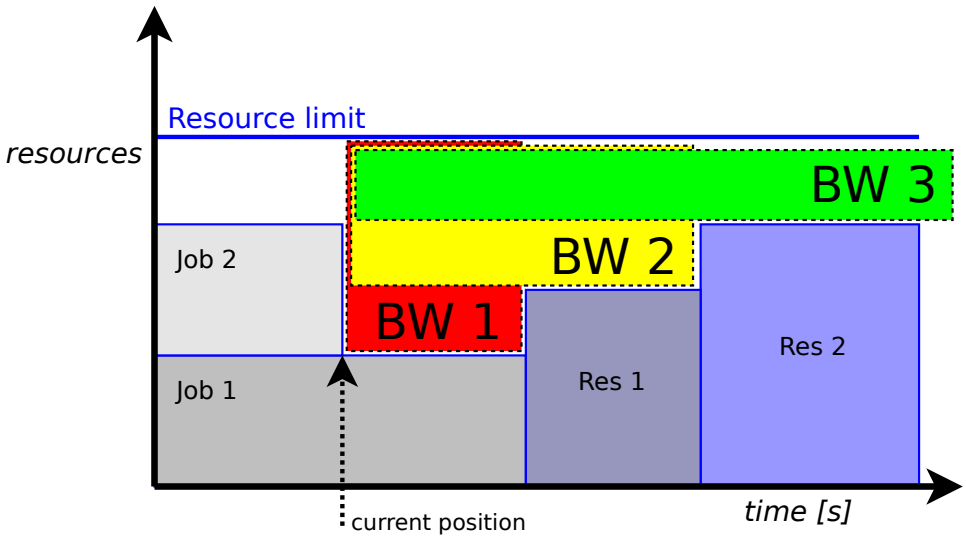

Figure 5.2: schedule with backfill windows

It shows two running jobs as well as two future reservations. The backfill windows are generated starting at the time position marked by the *current position* arrow. In general, the number of backfill windows exceeds the number of future reservations –relative to the current time position– by one. Since there are two reservations after the current time position in this example, three backfill windows are extracted. Extracting backfill windows is simple: a forward simulation is run starting with all currently available resources. As soon as an event such as a job dispatch or any future reservation consumes resources the available resources before this consumption form a backfill window together with the time span from the start of the forward simulation till the time of this event. The particular resources are consumed and the forward simulation is continued. As a result, each reservation causes the creation of a backfill window. Additionally, a backfill window is generated covering those resources, which are available throughout the entire time span of the forward simulation. In the example above this backfill window is named *BW 3*. Note that the backfill window resources can only be reduced throughout the forward simulation. They contain solely those resources, which are available for the entire time span.

Backfill windows have to be generated for each position in timeline. The results of previous timeline positions cannot be reused as soon as jobs are removed from the system. The additional resources released by the completed jobs cannot simply be added to the backfill windows, because this could overwrite the effects of future events on the same resources. E.g. in figure 5.2 the generation of backfill windows starting after the dispatch of job 2 creates effectively a single backfill window similar to *BW 3* except that it starts earlier. This window cannot be reused for the backfill windows generated at the position marked with *current position*. By releasing the resources of job 2 within the backfill window, one would get a backfill window with the resources, which *BW 1* provides. But this does not determine the time span of this altered backfill window. As a result, all future events would have to be repeated on this backfill window in order to retrieve the time span, in which its resources are available. However, this approach is identical to recreating all

backfill windows at each timeline position.

After generating the backfill windows, testing, whether a job can be inserted at the current time position, is a trivial task. For each job a backfill window is searched, which covers a time span wider than the job duration and which provides sufficient resources for the job request. The efficiency of using backfill windows in comparison to the previous backfill algorithm, which conducts forward simulations separately for each job, becomes apparent by analysing the main loop structure of both implementations shown in figure 5.3 and 5.4.

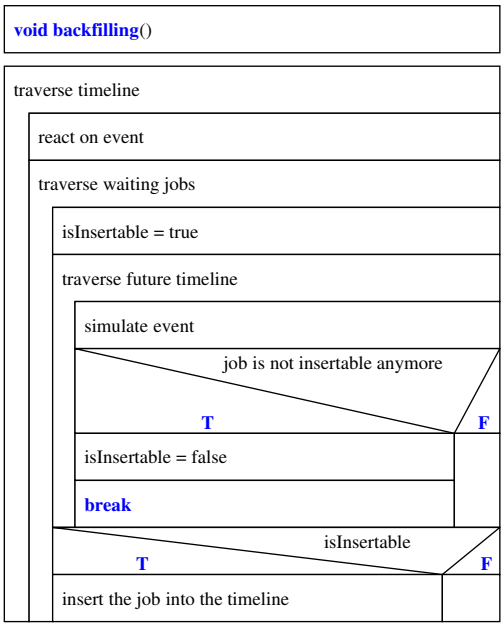

Figure 5.3: overview of the old backfilling algorithm described in section 3.1

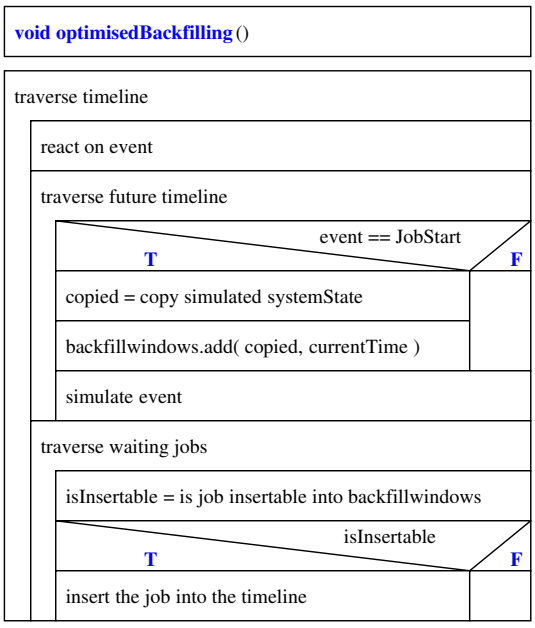

Figure 5.4: overview of optimised backfilling algorithm

The most important difference of both algorithms is that the old algorithm nests three loops, while the optimised version places two separated loops into the outer loop, which traverses the timeline. To sum up, the innermost loop responsible for running the forward simulation for each job is extracted and replaced by a loop directly nested into the timeline loop. However, this extracted loop becomes more complex. While in the old version the forward simulation is aborted as soon as the particular job collides with future reservations, the forward simulation for extracting backfill windows is forced to traverse the entire timeline in order to detect all future reservations. Moreover, the job insertion check of the optimised version actually has to execute an additional nested loop, which traverses the generated backfill windows for each job. Since the number of backfill windows is limited by the usually small number of reservations, the impact of this loop can be disregarded. Overall, backfill windows allow to implement the same behaviour with two instead of three nested loops. The produced schedule is identical for both implementations so that this optimisation does not affect the simulation's results. The optimised backfilling algorithm highly increases the efficiency of JuFo, which is reflected by the following complexity analysis. This analysis only focuses on how the simulation runtime is influenced by the number of jobs. Assuming that n is the number of jobs, F holds the complexity function for one insertion check of a job and m is the number of top dogs, the functions b_1 and b_2 model the complexity of the old backfilling and the optimised backfilling algorithm:

$$b_1(n) = n^3 * F \Rightarrow b_1 \in \Theta(F * n^3)$$
$$b_2(n) = n * (n + n * (m+1) * F) \Rightarrow b_2 \in \Theta(F * m * n^2)$$

The term $f(n) \in \Theta(g(n))$ expresses that $f(n)$ "is within a constant multiple of $g(n)$" [21]. I.e. f grows asymptotically as fast as g. For the functions b_1 and b_2 it is assumed, that the number of timeline iterations depends linearly on n. As a result, the old backfilling algorithm nests three loops, which depend on n, while the optimised algorithm only nests two loops. The factor F is identical for both complexity functions and is influenced by the chosen resource manager implementation. The number of inserted top dogs m is constant. Thus, its impact can be disregarded as long as it is significantly smaller than the actual number of jobs n. Since the number of jobs is –next to the number of simulated compute nodes– the most important determinant influencing the runtime of the simulation, these complexity functions reveal the gained efficiency of this optimisation. While the old backfilling algorithm depends cubically on n, the runtime of the optimised version only rises quadratically with n.

Finally, using backfill windows allows to simulate Moab's backfill algorithm more precisely. The presented optimised backfill algorithm at first iterates over waiting jobs and nests an inner loop, which traverses generated backfill windows. In contrast, Moab's algorithm swaps these loops: the outer loop traverses the backfill windows starting with the window, which possesses the highest amount of resources, while the inner loop searches for jobs matching the current window. Note, that backfill windows with wider time spans can only possess equal or less resources than those with small time spans, since the resources of a backfill window are only reduced in the future. By sorting the backfill windows in the mentioned way, at first those windows with smallest time spans are checked for fitting jobs. As a result, Moab's algorithm favors short jobs as they are likely to be inserted before higher prioritised jobs with longer wall clock limits. To conclude, the optimised algorithm can be adapted easily to Moab's behaviour. This is not possible based on the original backfill algorithm, because the required backfill windows are not extracted by this algorithm.

5.4 Parallelisation

An obvious approach for improving the simulation efficiency is parallelisation. So far merely a solid application architecture for a serial simulation program is developed along with ideas for optimising the implemented serial algorithms. Although the described optimisations strongly shortened the duration of the simulation, there appears to exist potential for parallelising the computationally intensive parts of these algorithms. However, JuFo is designed to be highly configurable and allows to combine the various implementations of the major simulation components such as the job sorter, the scheduling algorithm and the resource manager. Depending on the chosen component combination the focus of the simulation can vary. E.g. one simulation could use a complex and time consuming sorting algorithm in combination with a comparatively simple scheduling algorithm, while another could combine an expensive scheduling algorithm and might not sort the jobs at all. I.e. there is no generic approach for parallelising JuFo. Instead this section

extracts the most complex parts of the simulation and provides ideas for partitioning the sequential algorithms into separated tasks, which can be conducted simultaneously. While the optimisations of the previous sections are actually implemented, this section merely outlines ideas for parallelisation.

5.4.1 Profile analysis

Before collecting ideas for parallelisation the current status of JuFo has to be analysed for performance hot spots. This can be achieved by profiling the application with *gprof* [22]. It instruments the profiled program in order to count, how often a function is called, and monitors the execution time of each function dynamically (see [23]). This allows to extract, how much time is spent in each part of the simulation, by running a set of example simulations and evaluating them with gprof. With this information it is more reliable to predict the impact of parallelising a particular function. E.g. parallelising a function, which only consumes 5% of the total simulation time, will not cause significant speedup.

JuFo is tested and partly optimised for the supercomputer JUROPA. This system represents a challenge for the performance of the simulation program, because it provides more than 3000 compute nodes. In addition, on average more than 1000 jobs are queued by the scheduling system. Thus, it is reasonable to profile example simulations on JUROPA. Moreover, according to Fenlason et al. gprof is subject to "Statistical Inaccuracy" [22] so that its results are more reliable the longer the application's duration is. JUROPA currently provides the most time consuming input examples. As a result, these examples are suitable for profiling JuFo.

For this analysis ten example snapshots of JUROPA are chosen, which were collected from the 10th till 23rd of July 2012. Each LML file comprises 3290 compute nodes and between 900 and 1600 jobs. Between 12 and 26% of these jobs are running, while the remaining jobs have to be inserted within the simulation. These examples are simulated on a 64Bit OpenSuse 11.4 system with two cores at a speed of 2.53GHz. The simulation's duration varies depending on the executed example between 16 and 65 seconds. Detailed information about the example files can be found in table A.1 in appendix A.

The example files configure to sort the jobs by a priority term derived from the actual Moab configuration for JUROPA. They use backfilling with a maximum number of one reservation per queue. Note that JUROPA is split into two disjoint partitions. For each of them one top dog is allowed. The *NodeState* is chosen as resource manager, which is configured to forbid the sharing of compute nodes. This accords with the scheduling policy of JUROPA. The major part of the simulation's duration is spent in the *insertWaitingJobs* function of the scheduling algorithm. As a result, the profile analysis focuses on the composition of child functions invoked by *insertWaitingJobs*.

All results of the profiled examples are listed in table A.2 in appendix A. They are summarised and interpreted in the following. In average 89% of the total simulation time is spent in the *simulate* function of the backfilling algorithm. The remaining time is mostly spent for handling LML files with CodeSynthesis. The bigger part of the *simulate* function accounts for the insertion of queued jobs within the *insertWaitingJobs* function. On average 88.8% of the total simulation time is spent in this function, which is more than 99% of the time used by the calling *simulate* function. Thus, the composition of the compute

time for *insertWaitingJobs* is examined in detail as it represents the simulation's core.

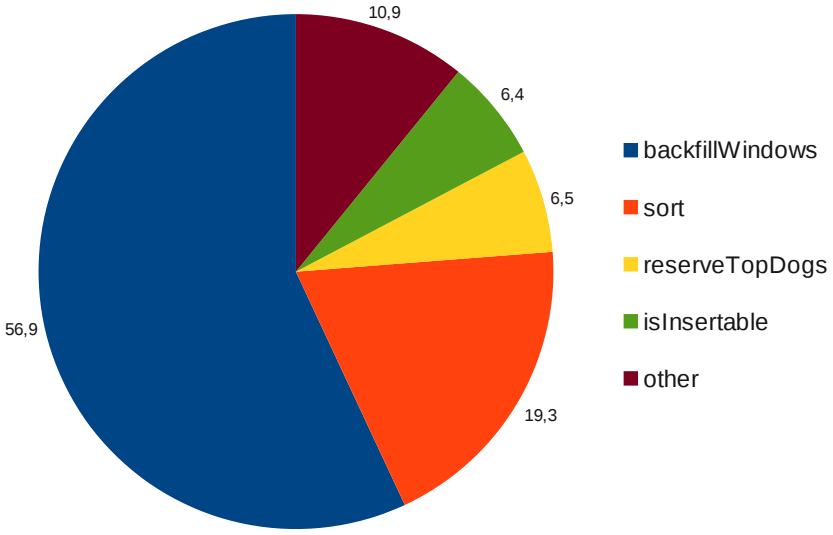

Figure 5.5: compute time composition of functions called within *insertWaitingJobs*

The pie chart 5.5 illustrates, which fraction of the compute time of *insertWaitingJobs* is spent on average in each called function. More than 56% of the time accounts for the generation of backfill windows. As described in section 5.3, backfill windows have to be extracted for each position of the timeline. Afterwards, the backfill windows are filled with suitable jobs. In the same manner, all waiting jobs need to be sorted by the given priority term at each timeline position. This includes the calculation of the current priority for each job as well as the actual sorting of jobs. This function consumes on average 19.3%. For the reservation of compute resources for top dogs forward simulations have to be executed, which takes 6.5% of the time spent in *insertWaitingJobs*. Since the usage of backfill windows simplifies the job insertion checks, they only consume 6.4%. The remaining 10.9% of compute time is among others spent for the execution of current events, logging the simulation status and comparing the current job with similar job requests.

5.4.2 Parallelisable algorithms

Due to the modular design of JuFo the entire simulation program has to be analysed for possibilities of parallelisation. The more parts of the simulation are run in parallel the higher is the chance of improving its efficiency. Moreover, considering Amdahl's law [24] the portion of a program, which cannot be parallelised, limits the possible speedup. Thus, it is beneficial to involve as many components of the simulation as possible while searching for parallelisable parts of the implemented algorithms. Thereby, the previous profile analysis helps to identify, which part of the simulation is promising for parallelisation. This allows to prioritise, which approach for parallelisation should be implemented first.

Parallel I/O

The first step of each simulation run is parsing the LML input data. The process of generating the object hierarchy from the LML file via CodeSynthesis cannot be parallelised, because it is a library call. However, this object hierarchy is converted into instances of the *SchedulingObjects* package. This conversion prepares the input objects for the simulation by parsing the most relevant attributes into actual attributes of a *Job* or *Node* instance. Thereby, these attributes can be accessed more efficiently and the refined objects are more comfortable to use within the other parts of the simulation. This conversion can be parallelised, since each job, node or reservation is processed independently. In the same manner, the inverse process of appending the simulation results to the object hierarchy of CodeSynthesis can be run in parallel. For each job its predicted dispatch and completion time as well as consumed resources have to be stored in the original CodeSynthesis data object. Since these input and output operations have to be run only once per simulation, the time currently spent for them is small. As a result, their parallelisation is not expected to significantly enhance the performance of the current simulation program.

Job sorting

Another simple approach for parallelisation is given for sorting jobs. In each presented scheduling algorithm all waiting jobs are sorted once for each event in the Timeline. E.g. for JUROPA that causes about 1000 jobs to be sorted at each event. For the implemented sorting strategy jobs are ordered in two steps: calculate the current priority for each job depending on the configured priority term and sort the jobs afterwards. Example profiles for JUROPA show that the priority calculation accounts for about 90% of the entire sorting algorithm. Thus, the priority calculation offers an effective way for parallelisation, since the job priorities at a given position in the timeline are calculated independently from each other. Note that there are job attributes such as the number of active jobs per user, which involve multiple jobs for the calculation of the attribute. However, the calculations can still be conducted independently, since these attributes are calculated globally prior to the actual sorting. Alternatively, JuFo could sort the jobs less frequently and use the same priorities for multiple timeline iterations. But this might change the simulation's results. Sorting of waiting jobs on average accounts for 17.1% of the total simulation time (see table A.2). Considering that the implementation of this parallelisation is comparatively simple, it represents a reasonable way for optimisation.

Scheduling algorithm

Most of the simulation time is spent for the insertion of waiting jobs. The presented scheduling algorithms have in common that they are nesting two major loops: the traversal of the timeline events and at each event the traversal of waiting jobs searching for a job, which can be inserted at the current position. The outer loop needs to be run sequentially as each iteration depends on previous iterations. However, the inner loop allows to be parallelised as long as no jobs are inserted into the resource manager. In this case JuFo only reads data, but does not modify the simulation's state. As a result, all jobs can be checked simultaneously. This allows to partition the inner loop among a number of parallel

threads or processes. But, if a job can be inserted, the state of the resource manager has to be changed by consuming the resources requested by the particular job. Therefore, the threads either have to check in a given interval, if another thread has found a suitable job, or insertion checks of jobs might need to be repeated for all jobs with a lower priority than the new inserted job. This parallelisation approach can be applied to all implemented scheduling algorithms regardless of the used resource manager or job sorting strategy. But, the effect on the performance of the simulation will vary depending on the parallelised scheduling algorithm and on the input data. E.g. the former backfilling algorithm (see figure 5.3) has a rather complex insertion check, because a forward simulation is executed for each job. In contrast, the insertion check of the optimised backfilling algorithm makes use of the previously generated backfill windows, which strongly reduces the complexity for each insertion check. While the insertion checks of the former backfilling algorithm account for a big part of the simulation time, the optimised version requires more time to generate backfill windows (56.9% of the compute time spent in *insertWaitingJobs*) than for the traversal of waiting jobs (6.4%). Thus, a comparatively small part of the simulation duration can be parallelised by this approach so that the speedup is expected to be low for the optimised backfilling algorithm.

Unfortunately, the generation of backfill windows cannot be parallelised efficiently so far. It requires to iterate sequentially over the timeline events starting at the current timeline position in order to simulate the future state of available resources. Moreover, this process has to be repeated for each timeline position as new jobs might have been inserted, which alters the backfill windows produced in the last iteration.

Resource manager

Another way to accelerate job insertion checks is to parallelise the *isJobInsertable* function of the resource manager. E.g. the *NodeState* retrieves eligible compute nodes for a job request with the help of range sets. But each node is able to configure individual queues, which are allowed to start jobs on the particular node. As a result, all eligible nodes have to be traversed in a loop, of which each iteration is independent from the other iterations, since each node's queues can be checked separately. Thus, this node traversal can simply be partitioned to parallel threads.

This function will require considerable more compute time once JuFo is extended for Loadleveler. As described in section 3.2.1 on Blue Gene systems Loadleveler has to allocate an exclusive torus or mesh network to each job. Therefore, a coherent unused sub block of the entire network has to be found, which matches with the shape requested by the particular job. A Blue Gene/P system provides a 3D torus network among all midplanes. In a simplified model they are arranged logically in a cuboid, where each midplane is connected with all adjacent midplanes. Additional cabling and allowing midplanes to forward network packages of foreign applications also connects midplanes, which are not directly adjacent (see [14]). These extensions are disregarded in the following explanations in order to keep the parallelisation approach simple. Each job requests a sub cuboid of a given shape. This leads to six nested loops: the first three loops iterate over the start edge of the sub cuboid, while the inner loops traverse the shape of the requested cuboid. If all nodes are available for one start edge, a possible position for this job is found. A similar algorithm can be applied for networks of higher torus dimension by nesting more loops.

The search for a sub cuboid within a 3D array of boolean values, which models the midplane network, can be directly parallelised. The outer loop of the algorithm can be split into equally sized parts. Each of these parts is then searched for a coherent sub cuboid. The idea of this parallelisation is depicted in figure 5.6.

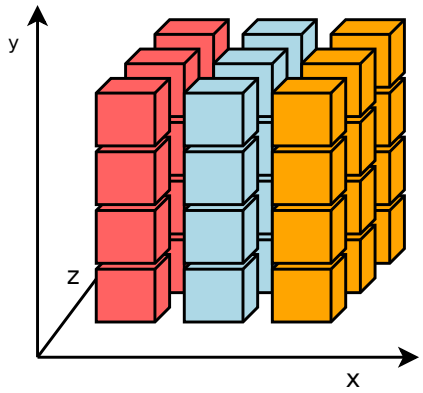

Figure 5.6: parallel search of sub cuboid within torus network

It shows how the search can be partitioned for three parallel threads. Each of the small cubes represents a midplane. Assuming that the outer loop iterates in x-direction, the search could be split into the parts identified by the three colours. Each thread can search independently for available sub cuboids starting at the midplanes assigned to the particular thread.

In the same manner other extensions of JuFo could become the most time consuming parts of the simulation. A profile analysis is helpful to identify the performance hot spots, which are worth for optimisation or even parallelisation. However, these optimisations often contradict with design goals like extensibility or simplicity. It is likely that a parallel algorithm is harder to understand than a serial algorithm just as the old backfilling algorithm is simpler than the optimised version using backfill windows. Thus, an optimisation or parallelisation should only be implemented, if the gained efficiency excuses the increased complexity. Moreover, JuFo will typically be executed on a single computer with multiple cores. As a result, its parallelisation does not require to be designed and optimised for high scalability, since the expected speedup is limited by the low number of parallel threads.

Chapter 6

Implementation aspects

While the previous chapters examined the design of JuFo independently from the actual programming environment and language, this chapter shortly discusses the implementation details. At first, the development environment is outlined by describing tools and libraries used for the simulation program. Afterwards, the designated extension points are summarized in order to simplify enhancing JuFo.

JuFo is written in C++. It is chosen for the simulation program due to its efficiency as a compiled language and the possibility for object oriented programming (OOP). Moreover, C++ is a widely spread programming language, which is important for a project designed for third party extensions like JuFo. It makes extensive use of the OOP features of C++ in order to encapsulate and separate the different components of this simulation program. Abstract classes are used as interfaces defining how the components are allowed to interact with each other. The actual implementations inherit from these interfaces and specify the component behaviour. By only using these abstract superclasses when interacting with another component, they are kept independent from their actual implementations. E.g. this allows for implementing new scheduling algorithms independently from the type of resource manager passed to the Simulation's constructor. If the scheduling algorithm was aware of the resource manager implementation, changing one of the components would probably cause the other to be adapted as well.

6.1 Development environment

The simulation program is developed on Linux and can be compiled with the *GNU project C and C++ compiler (GCC)*. Eclipse is used as development platform and especially the *C/C++ Development Tools (CDT)* plug-in supports programming by providing context sensitive code completion, on-line code analysis for errors, integrated building and running of programs and interactive debugging. JuFo is built with a customized makefile, which defines rules for compiling and linking the source code with various libraries used for the simulation framework.

6.1.1 Libraries

JuFo requires several external libraries, which are listed in the following along with a summary of their tasks.

CodeSynthesis XSD – C++ architecture for XML Data Binding, parsing of LML files (see [18])

Xerces – XML parser for C++, which is used by CodeSynthesis (see [25])

PCRE – handling of regular expressions, required for parts of LML parsing (see [26])

CppUnit – C++ port of JUnit for automated software testing, almost each class of JuFo is tested by a CppUnit testcase (see [27])

MathExpr – parsing of arbitrary mathematical formulas, needed for calculation of job priorities (see [28])

CodeSynthesis strongly simplifies the handling of XML files, which are validated by an XML Schema like LML. This library parses the LML files, checks for validity and converts the data into a C++ object hierarchy. These objects can be read and modified like standard objects. Then the modified objects are serialized back into valid LML automatically. The *PCRE* library is used for processing regular expressions within the simulation program. It is easy to install and simplifies several parts of interpreting parsed LML data. It is especially useful as SchedSim makes extensive use of regular expressions, which PCRE allows to insert without any modifications into JuFo. Small test classes are written with the help of *CppUnit*. For every new feature a test case is implemented, which checks the correctness of new functionality. E.g. test cases are written for parsing dates, for each scheduling algorithm or for checking, whether LML data is parsed as expected. This ensures the robustness of JuFo and allows for regression tests, which test, whether old functionality still works after implementing new features. Moreover, the test cases show how to use and combine the simulation classes, which simplifies understanding and extending the simulation program. *MathExpr* is an open source "mathematical expression parser in C++" (see [28]). It parses arbitrary formulas, allows to set the values for all included variables and finally evaluates the formulas. This library is applied to calculating the job priority in order to sort waiting jobs by the system priority defined by a mathematical formula.

JuFo intends to minimize the number of external libraries in order to keep the installation of pre requisites simple. CodeSynthesis and PCRE are required, because implementing their functionality from scratch would be very time-consuming. CppUnit is merely needed for development, while simply using JuFo does not depend on this testing library. MathExpr represents a single C++ class, which can be compiled without further requirements. Thus, including MathExpr into the simulation program does not increase the installation effort. To conclude, three libraries need to be installed on the target system in order to run JuFo.

6.2 Extension points

In general, all parts of the simulation basis provided by JuFo can be adapted or extended. However, several components are especially designed for adding new implementations and these are summarized within this section. Although all of these extension points are mentioned throughout section 4, concentrating them in the following paragraphs outlines how JuFo is meant to be extended. A job scheduler simulation in JuFo is a combination of three major class implementations: a *JobSorter*, a scheduling algorithm inheriting from *Simulation* and a resource manager implementing the *SystemState* interface. Each class can be implemented independently from the others as interactions to other components are limited to the abstract interfaces of each component. Due to clearly defined tasks of each component, the actual implementations can be exchanged arbitrarily. Extending the simulation program by an additional implementation comprises two steps: implement the abstract interface by a new subclass of the corresponding interface and add this new implementation to the *Configuration* class, which selects the actual implementation for each component by parsing the LML input file.

The JobSorter class implements the strategy for sorting lists of jobs by their priorities. It is realized by a single function, to which the unsorted list of jobs is passed. In order to add another scheduling algorithm a subclass of the framework given by the abstract *Simulation* class has to be created. Since this framework already inserts running jobs and provides functions for forwarding the simulation results to the object hierarchy of CodeSynthesis, the subclass only needs to implement the *insertWaitingJobs* function. It represents the core of each scheduling algorithm. This function decides, when the jobs need to be sorted, how many jobs are handled as top dogs and inserts as well as executes events in the timeline. Examples for this function are documented in section 3.1 with FCFS, backfilling and List-Scheduling. The third designated extension point is given by the *SystemState* interface, which simulates the behavior of the resource manager. To implement a new resource manager, functions for inserting and deleting jobs are required. In addition, a resource manager has to be able to copy its current state so that forward simulations can be run on a copied state without changing the actual system state within the simulation. Finally, a function has to be provided, which checks, whether a job can be inserted into currently available resources.

Since at least one reference implementation is given for each of these extension points, they can be used as guideline examples for new implementations. It is also possible to subclass existing implementations in order to add small extensions to the given functionality.

Chapter 7

Simulation tests

While prior chapters cover the design of JuFo and methods for improving the efficiency of this simulation program, this chapter discusses approaches for evaluating and testing the outcomes of the simulator. The first section explains how the different modules of the simulation are checked with the help of unit tests. This allows to control the functionality of each new implemented feature or component separately. However, these synthetic and restricted tests are insufficient for investigating the simulation's accuracy on real supercomputers. Although they can be used to test the correctness of the implemented algorithms, more complex tests are required in order to evaluate the predicted schedules for actual parallel systems. The idea for these tests is to gather the actual workload of the particular supercomputer and run the simulation starting in the past. Afterwards, the predicted schedule is compared with the actual schedule produced by the real scheduler. The concept of this test is presented in the second section. Based on this test an estimation for the accuracy of JuFo on the supercomputer JUROPA is acquired.

7.1 Module tests

Since JuFo is composed by a set of mostly independent modules, unit tests can be applied to control their functionality. Each unit test is used to check the validity of a so-called *test unit*, which is defined as a "set of one or more computer program modules together with associated control data" [29]. A test unit should be kept as small as possible in order to focus on special parts of the tested application. As mentioned in the previous chapter, the unit testing framework CppUnit represents the basis for unit tests of JuFo. For nearly all simulation classes corresponding tests are developed, which run common use-case scenarios and compare the expected results with the actually obtained results. E.g. the *Time* class stores relative time values and allows to calculate the difference of two Time instances. A corresponding CppUnit test function written in C++ is given by the following listing.

Listing 7.1: minus operator test for the *Time* class

```
1 void TimeTest::testOperatorMinus()
2 {
3   cout<<"testOperatorMinus"<<endl;
4   Time time1(100);
5   Time time2(200);
```

```
6   Time time3 = time2-time1;
7   CPPUNIT_ASSERT_EQUAL( 100.0, time3.getSeconds() );
8 }
```

It creates two Time instances and calculates their time distance by calling the difference operator. The call of *CPPUNIT_ASSERT_EQUAL* compares the calculated difference with the expected value *100.0*. If these values differ, an error message will be generated explaining which test case failed and list expected and actual values. A set of those test functions forms a test suite, which also provides functions executed prior and after the actual test cases in order to initialise and release test objects commonly used by all test cases. More than 20 test suites, which define over 100 test cases, build the module tests of JuFo. The complexity of the test cases depends on the tested module or feature. While testing the difference operator of the Time class is a trivial task, the test development for checking the variety of scheduling algorithms arbitrarily combined with different job sorting strategies and resource managers is more complex.

In order to test the implemented scheduling algorithms, synthetic LML files are used as input for running an entire simulation. Afterwards, the produced schedule is compared with the expected schedule based on the configuration parameters and chosen component implementations for the scheduling algorithm, the job sorting strategy and resource manager type. The expected schedule has to be acquired manually so that this approach is only feasible for small input files. These synthetic simulation runs are helpful for testing any adaption of the scheduling algorithms. E.g. JUROPA does not allow nodes to be shared by multiple users. Thus, the *NodeState* resource manager needs to be extended by a configuration parameter, which forbids node sharing. On the one hand, this feature could be tested by creating a NodeState instance with node sharing disabled. The insertion of jobs should then consume entire compute nodes. On the other hand, a corresponding test file can be created in order to run a simulation. Afterwards, the produced schedule is checked for validity. While the first approach exclusively tests the NodeState, the latter test checks the NodeState's implementation embedded into the simulation framework. If possible both test ideas are applied to each major feature of the simulation program. In order to illustrate this test approach the following section documents a synthetic test run and thereby explains how to use and interpret the results of JuFo.

7.1.1 Simulation example

The entire configuration and input data for JuFo is provided by a single LML file. This file is parsed in order to read the simulation's configuration and to run the simulation on given jobs, reservations and nodes. Afterwards, the output LML file is generated by extending the input data by attributes holding the predicted dispatch and completion time as well as used compute resources.

Listing 7.2 shows an example input file, which can be processed with JuFo.

Listing 7.2: input LML file

```
1 <lml:lgui>
2 <objects>
3 <object id="j000001" name="job1" type="job"/>
4 <object id="j000002" name="job2" type="job"/>
```

```
<object id="j000003" name="job3" type="job"/>
<object id="node1" name="theNode" type="node"/>
<object id="sys" type="system"/>
<object id="sched" name="scheduler" type="scheduler"/>
</objects>
<information>
<info oid="j000001" type="short">
<data key="queuedate" value="07/12/12-14:00:00"/>
<data key="dispatchdate" value="07/12/12-14:59:00"/>
<data key="state" value="Running"/>
<data key="nummachines" value="1"/>
<data key="taskspernode" value="2"/>
<data key="totalcores" value="2"/>
<data key="wall" value="65"/>
<data key="nodelist" value="(1-1(4-5))"/>
</info>
<info oid="j000002" type="short">
<data key="queuedate" value="07/12/12-14:00:00"/>
<data key="state" value="Idle"/>
<data key="nummachines" value="1"/>
<data key="taskspernode" value="4"/>
<data key="totalcores" value="4"/>
<data key="wall" value="20"/>
</info>
<info oid="j000003" type="short">
<data key="queuedate" value="07/12/12-14:30:00"/>
<data key="state" value="Idle"/>
<data key="nummachines" value="1"/>
<data key="taskspernode" value="3"/>
<data key="totalcores" value="3"/>
<data key="wall" value="4"/>
</info>
<info oid="node1" type="short">
  <data key="ncores" value="5"/>
</info>
<info oid="sched" type="short">
  <data key="system_sysprio" value="-DATEqueuedate"/>
  <data key="system_state" value="NodeState"/>
  <data key="scheduling_algorithm" value="Backfilling"/>
</info>
<info oid="sys" type="short">
  <data key="system_time" value="07/12/12-15:00:00"/>
</info>
</information>
</lml:lgui>
```

This LML file specifies three jobs scheduled on one compute node, which has five processors. The current system time of the simulated parallel computer, which also defines the start date of the simulation, is given by the *system_time* attribute of the *sys* object. This object can be used to provide any information about the simulated system architecture. So far merely the *system_time* attribute is interpreted. The first job is dispatched before the current system time and is already running on processors four and five, which is declared in the *nodelist* attribute. The other jobs are waiting. While *job2* requires four processors for 20 seconds, *job3* requests three processors for only four seconds. The *nummachines* attribute defines the number of requested compute nodes. Since there is only one node in this example, this attribute is set to *1* for all jobs. Finally, the *sched* object comprises overall parameters affecting the simulation algorithm. Here it defines to prioritise jobs by

their queue dates and it chooses to use the resource manager *NodeState* and *Backfilling* as scheduling algorithm.

This input data is sufficient for running JuFo. Considering the given configuration *job1* is assumed to finish after five seconds. Since *job2* gains a higher priority than *job3*, the simulation tries to insert *job2* at first. As only three of four required processors are available at the beginning, a reservation is searched. The requested compute resources become available as soon as *job1* is completed. Thus, *job2* is dispatched after five seconds. This leads to a backfill window of three processors available from the start till second five, which allows for backfilling *job3*. The produced schedule is depicted in figure 7.1.

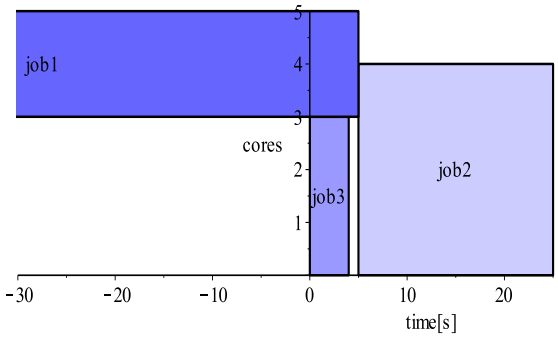

Figure 7.1: predicted schedule for LML input 7.2

A test case can be derived easily from this example scenario. JuFo processes this input file and generates a predicted schedule. If this schedule differs from the expected schedule, a corresponding assertion will detect this error and report that the algorithm produces invalid results. These test examples can be conducted as regression tests. After each adaption of JuFo all test suites can be executed. This accelerates error detection and enhances the robustness of the simulation program. Moreover, each test case functions as example code demonstrating the usage of the tested module. This simplifies understanding the code basis and provides entry points for further developments.

7.2 Tests on real systems

The described synthetic tests are helpful for controlling implemented algorithms in a number of well defined and rather theoretical scenarios. However, the question arises how the predicted schedules for real systems can be evaluated regarding their accuracy.

7.2.1 Concept

A first intuitive approach is running JuFo at a certain time and comparing the actual dispatch times of all involved jobs with their predicted times. For this comparison the actual dispatch times can be retrieved by waiting for all jobs to be started. The accuracy estimation obtained by this test accords with the accuracy, which the user can expect from JuFo when using it as prediction. But this test strongly depends on the accuracy of the wall clock limits, which users specify as upper limit for a job duration. E.g. assuming the users

double all wall clock limits, but the actual run-time of each job stays the same, the test will downgrade the accuracy estimation of the prediction, although the implementation is not changed at all. Moreover, future events such as additional job submissions or premature cancellation of waiting jobs influence the estimated accuracy, although these determinants are unknown to the simulation program at the time of its execution. As a result, this test reflects the ordinary accuracy of JuFo, but it is improper for evaluating how precisely the actual scheduling algorithm is simulated.

A second idea is comparing the predicted schedule of JuFo with existing job scheduler simulators especially implemented for a target system. E.g. the scheduling system Moab provides the *showstart* command for estimating job dispatch times (see [15, p.306]). A simple test is to run JuFo and the target system specific prediction simultaneously. Afterwards, their estimated dispatch times are compared. However, not every batch system provides such simulation programs. Moreover, *showstart* has a number of disadvantages when using it for this test:

- it has to be run separately for each job
- it takes several seconds for estimating the dispatch time of a single job
- the documentation of showstart is sparse

Thus, using the results of *showstart* as target value for JuFo is questionable. Iterating over all queued jobs of a large supercomputer with *showstart* cannot be achieved in a reasonable time span. That is why reliable dispatch values can be retrieved only for a small number of jobs. But due to the lack of detailed information about its configuration and algorithms it is difficult to interpret its results and relate them to the schedule predicted by JuFo.

A more methodical test concept is given by the following algorithm:

1. gather all events relevant for scheduling such as job submissions, starts, cancellations and completions
2. generate an LML test file containing all gathered events with adjusted wall clock limits
3. run JuFo as if it was started at the beginning of step 1, i.e. let the simulation predict, what already happened
4. compare the predicted schedule with real results gathered in step 1

This idea can be summarised as predicting scheduling events of the past and comparing the results with actually happened events. The first step means to take snapshots of the system status by calling LML_da in a given interval. In the second step all snapshots are merged into a single LML file, which has the same structure like a normal input file. The scheduling events are listed as if captured at the system time of the first snapshot. I.e. jobs are marked as running, if they were running at that time, and future jobs, which are started ahead of that time, are marked as submitted jobs. Furthermore, all wall clock limits, which are originally specified by the users, are adapted to the actual run-time of

each job. This is only possible for jobs, which are completed throughout step 1. Adapting the wall clock limits eliminates the former shortcoming of the first test approach of this section. The other shortcoming of missing future events is addressed by adding jobs, which are queued after the beginning of the first LML snapshot, with a corresponding future queue date. JuFo supports these jobs by excluding them from the scheduling simulation until the currently simulated time exceeds their queue date.

The third step of this test algorithm runs JuFo to produce a schedule. Since the actual dispatch times of all jobs, which are started during step 1, are known, the final step can compare the predicted with the real schedule. This test focuses on the accuracy of the simulated scheduling algorithm. It eliminates most of the unpredictable determinants such as imprecise wall clock limits and future events like job cancellations or new job submissions.

7.3 Test results for JUROPA

Since JuFo is especially tested for JUROPA throughout the profile analysis in section 5.4.1, its configuration is already adapted closely to the actual scheduling system. This allows to run the test concept explained in the previous section on JUROPA. The test evaluates the accuracy of the used configuration for simulating the actual scheduling algorithm used by JUROPA.

For eight different days between the 23rd of July and the 7th of August test runs are documented in the following. The more tests are executed the more reliable is the analysis of their results. For each test status snapshots in a given time span are merged into a single test file. The covered time spans vary between about two and more than five hours. In general, a longer time span causes the simulation to be less precise, because jobs, which are scheduled inaccurately, also change the scheduling of future jobs. As a result, the described test concept is more challenging for JuFo than a normal prediction based on a single status snapshot. For the latter use case at least the position of the running jobs is correct. For each job dispatched within the test duration its actual dispatch date is compared with the predicted dispatch date. A prediction error is calculated as the difference of both dispatch dates in seconds. Thus, a negative error indicates that the simulator schedules the corresponding job later than it is actually dispatched on JUROPA, while a positive error is returned for a job scheduled too early. Table 7.1 lists detailed results of the tests.

Test date	**Duration**	**Jobs**	**Mean**[s]	**Median**[s]	**Min.**[s]	**Max.**[s]	σ[s]
07/23/2012	01:43:27	107	33.6	34	-3020	4871	1154.3
07/24/2012	03:50:10	226	-38.3	31	-8175	7340	1709.6
07/25/2012	02:12:01	186	-773.6	-116	-7467	3336	1851.0
07/27/2012	04:22:20	191	34.2	42	-11038	7515	1866.3
08/01/2012	02:35:14	122	642.6	76	-4637	6815	1886.8
08/02/2012	04:25:37	280	468.6	64	-10462	14620	3360.9
08/06/2012	03:19:02	133	-326.1	54	-7783	7516	2473.2
08/07/2012	05:39:31	249	289.3	49	-15378	19796	4104.9

Table 7.1: prediction error distributions for example tests on JUROPA

The *Duration* column of this table holds the time span covered by each test. The third column lists the number of jobs, for which the prediction error is calculated. Only jobs are considered, which are started within the test duration as for these jobs the actual dispatch date is known. Moreover, jobs are excluded from the error calculations, if they are predicted to start after the end of the test duration. This ensures that JuFo only simulates within the actual test duration. The last columns hold the mean, median, minimum and maximum value as well as the standard deviation σ for the prediction errors. These values are listed in seconds.

The tests predict between 100 and 280 jobs. The mean values range from about -13 to 11 minutes. Along with the median values between -2 and 1.27 minutes the results show, that on average JuFo provides a useful simulation for JUROPA's job scheduler. However, the minimum, maximum and standard deviation columns indicate the existence of jobs, which are not scheduled accurately by the simulation. For each test there are jobs, for which the prediction error is almost as large as the actual test duration. These outliers also explain the large standard deviations. More detailed information about the error samples can be found in appendix B.

A possible reason for inaccurately scheduled jobs is that their system priorities are calculated wrong. Moreover, reservations are not collected by LML_da so far, which might influence the predicted schedule. Another reason for completely wrong scheduled jobs could be the adaption of scheduling policies by system administrators, which cannot be detected automatically by LML_da. E.g. the priority of a certain job could be increased for test purposes manually throughout the test duration. These system specific administration and scheduling details can only be handled by extending both JuFo and LML_da with the help of the particular system administrators in order to adapt the simulation closer to the actual scheduling system.

The test results demonstrate that JuFo can be applied for the prediction of JUROPA's scheduling system. Although the average error is rather low, the variance reveals that there is potential for improving the simulation's accuracy. Note that the shown accuracy is only achieved, because for these tests the wall clock limits are replaced by the actual job durations. The accuracy of JuFo for on-line predictions still depends on the accuracy of the wall clock limits provided by the users of the supercomputer. Based on these test files each adaption of the implemented scheduling algorithm can be tested. If the adaption decreases the average error as well as its standard deviation, it is likely that it improves the accuracy of JuFo. Similar tests can be conducted for other supercomputers, which are predicted by the simulation program.

Chapter 8

Conclusion and outlook

This thesis documents the major steps conducted in order to design and implement JuFo, a simulator for job schedulers of supercomputers.

8.1 Conclusion

At first the task is narrowed down to the simulation of the global scheduling layer responsible for partitioning available compute resources among eligible jobs. This layer is of high interest for administrators as well as users, who would like to know the future dispatch times of queued jobs. The corresponding scheduling problem is defined mathematically and related to existing complex scheduling problems. This formalisation is often referenced in the documentation of the design for JuFo in order to avoid circumscriptions and to pinpoint the scope of each documented component. Next to the problem definition the technical basis for the implementation of JuFo is examined. While LML_da gathers status information of the supercomputer and passes it to the simulation program as raw LML data, the visualisation clients included in LLview and PTP will illustrate the predicted schedules. This ensures that JuFo focuses on the scheduler prediction, since data gathering as well as visualising the simulation's results are implemented by the monitoring environment, into which JuFo is embedded. However, this concept hands the control about which information is gathered from the parallel computer over to LML_da. Thus, the simulator relies on this information basis and cannot request possibly missing data. The analysis of SchedSim, which is part of LLview, forms the basic concept of the developed simulator, but also obtains disadvantages such as limited efficiency and extensibility. These are eliminated in the evolved design of JuFo. After investigating the existing prediction implementation and monitoring environment, common scheduling algorithms targeted on the global scheduling layer are introduced. These algorithms are used by the scheduling systems Moab and Loadleveler, which are analysed from a global perspective in order to gain a common functionality basis for supported target systems. Both systems apply modifications of the backfilling algorithm. They separate the actual scheduling from the resource management. Furthermore, they make use of consumable resources, which are provided by compute nodes and requested by submitted jobs. Although a major design goal is to keep the simulator as generic as possible, it still requires the simulated system to provide

compute nodes and jobs running on them. Based on these considerations the design of JuFo is developed. It can be roughly divided into eight packages, which interact mostly via abstract interfaces. This allows for arbitrarily combining various implementations of the core components for sorting of jobs, resource managers and scheduling algorithms. The challenging task for JuFo is balancing abstraction and practicability as well as the wide range of configuration parameters and efficiency. While abstraction is achieved by strictly separating the major packages from each other, the reference implementations function as guidelines for the practical usage of the given interfaces. The simulator is configured with the help of the flexible raw LML format, which simplifies extending the existing implementation. However, this flexibility increases the effort for correctly parsing the input data. Approaches for enhancing the simulation's efficiency such as the generation of backfill windows are examined, which offers entry points for further improvements. The detailed documentation of all data structures and the core algorithms allows for extending JuFo or adapting it to additional scheduling systems. The simulation is tested with a large number of module tests each focusing on a specific aspect. Moreover, a test framework for adapting the simulation's scheduling algorithm to a real system's algorithm is developed. In spite of the demand for JuFo to simulate the particular scheduling system as precisely as possible, it can only function as model and cannot cover all possible configuration parameters supported by the actual scheduling system. Therefore, the simulator has to take into account the most significant determinants influencing the schedule. As these determinants differ for each supercomputer, the simulator's configuration needs to be adjusted to the particular target system by the administrators in order to achieve reasonable predictions.

8.2 Future work

This project especially designed for extensibility allows for a lot of further developments. A first step is adapting the simulator closer to the supercomputers maintained at JSC in order to optimise the prediction's correctness. Moreover, new target systems like JUQUEEN – the Blue Gene/Q successor of JUGENE also scheduled by Loadleveler (see [30])– have to be simulated with JuFo. The developed test framework, which eliminates inaccuracies of the input data such as wall clock limits provided by users, is a helpful tool for evaluating the accuracy of the simulated scheduling algorithm. In doing so, LML_da also needs to be extended. Further status information has to be gathered such as reservation data, a list of nodes assigned to each queue and user priorities based on site specific accounting. This will not only improve the simulator's efficiency, but will also help to understand and investigate the actual scheduling system by comparing the expected schedule with the real schedule.

As indicated by the documented approaches for optimising the simulator, there are still ideas for further enhancing the efficiency. Many components of the simulator might need to be reviewed repetitively for possible performance hot spots as depending on the particular configuration different parts of the simulation can consume the majority of its duration. Especially, parallelisation offers a number of opportunities to further minimise the simulation's run time. This is crucial for the practicability of JuFo, because the size of parallel systems and the number of simulated jobs and compute resources is steadily increasing.

The online visualisation of the predicted schedule is still work in progress. Both the

LLview and PTP clients have to implement parsing the simulator's results and displaying them in a diagram similar to the prediction window of LLview. Moreover, alternative visualisations are required, since the current diagram can become crowded for a large number of displayed jobs. E.g. the user has to be able to zoom into the diagram or filter displayed jobs. The schedule could also be visualised as an animated nodedisplay showing the predicted status of the supercomputer in a given interval.

In addition, the input data for JuFo can be enriched by speculative user data obtained from historical workload information. Instead of using the wall clock limit provided by the users as assumed job duration the average duration observed for past jobs of this user could function as predicted job duration. User profiles could be created storing the frequency of job submissions and repeating submission pattern. This would allow to include statistically predicted future jobs into the input data. The combination of methodical simulation of the scheduling algorithms along with statistical prediction for future submission and job duration could lead to significantly increased accuracy of the predicted schedule.

Bibliography

[1] Jochen Krallmann, Uwe Schwiegelshohn, and Ramin Yahyapour. On the Design and Evaluation of Job Scheduling Algorithms. `http://www.gwdg.de/fileadmin/inhaltsbilder/Pdf/Yayhapour/cei_ipps99.pdf`, 1999.

[2] Jülich Research Centre. JUROPA / HPC-FF System Configuration. `http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUROPA/Configuration/Configuration_node.html`, March 2012.

[3] Jülich Research Centre. Batch Job Processing on JUGENE. `http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUGENE/UserInfo/LoadLeveler.html`, March 2012.

[4] Jülich Research Centre. LLVIEW: graphical monitoring of LoadLeveler controlled cluster. `http://www.fz-juelich.de/jsc/llview/`, March 2005.

[5] Gregory R. Watson, Wolfgang Frings, Claudia Knobloch, Carsten Karbach, and Albert L. Rossi. Scalable Control and Monitoring of Supercomputer Applications using an Integrated Tool Framework, September 2011.

[6] Alfred Arnold. *Untersuchung von Scheduling-Algorithmen für massiv-parallele Systeme mit virtuell gemeinsamem Speicher.* PhD thesis, Jülich Research Centre, ZAM, 1997.

[7] Dror G. Feitelson and Larry Rudolph. Parallel job scheduling: Issues and approaches. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 1–8. Springer-Verlag, 1995.

[8] Peter Brucker and Sigrid Knust. *Complex Scheduling.* Springer Verlag, 2012.

[9] The Eclipse Foundation. Eclipse PTP. `http://www.eclipse.org/ptp/`, April 2012.

[10] Carsten Karbach. Konzeption und Umsetzung einer Beschreibungssprache für Statusinformationen von Parallelrechnern als Basis einer Webschnittstelle fur LLview, August 2010.

[11] David A. Lifka. The ANL IBM SP Scheduling System. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 295–303. Springer-Verlag, 1995.

[12] IBM. *Tivoli Workload Scheduler LoadLeveler, Using and Administering.* IBM, `http://www2.fz-juelich.de/jsc/jugene/documentation/`, 3.5 edition, January 2009.

[13] Carlos Sosa and Brant Knudson. IBM System Blue Gene Solution: Blue Gene/P Application Development. http://www.redbooks.ibm.com/redbooks/SG247287/wwhelp/wwhimpl/js/html/wwhelp.htm, May 2012.

[14] Gary Lakner. IBM System Blue Gene Solution: Blue Gene/P System Administration. http://www.redbooks.ibm.com/redbooks/SG247417/wwhelp/wwhimpl/js/html/wwhelp.htm, May 2012.

[15] Adaptive Computing. *Moab Workload Manager, Administrator Guide*. http://www.adaptivecomputing.com/resources/docs/, 7.0 edition, 2012.

[16] Adaptive Computing. *Torque, Administrator Guide*. http://www.adaptivecomputing.com/resources/docs/, 4.0 edition, 2012.

[17] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[18] Code Synthesis Tools CC. XSD: XML Data Binding for C++. http://www.codesynthesis.com/products/xsd/, May 2012.

[19] Ed Ort and Bhakti Mehta. Java Architecture for XML Binding (JAXB). http://java.sun.com/developer/technicalArticles/WebServices/jaxb/, March 2003.

[20] Owen Astrachan. Bubble Sort: An Archaeological Algorithmic Analysis. http://www.cs.duke.edu/~ola/papers/bubble.pdf, 2003.

[21] Paul E. Black. θ. In *Dictionary of Algorithms and Data Structures*. U.S. National Institute of Standards and Technology, August 2008. http://www.nist.gov/dads/HTML/theta.html.

[22] Jay Fenlason and Richard Stallman. GNU gprof. http://www.cs.utah.edu/dept/old/texinfo/as/gprof_toc.html, July 2012.

[23] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a Call Graph Execution Profiler. http://docs.freebsd.org/44doc/psd/18.gprof/paper.pdf, 2012.

[24] Gene Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, pages 483–485, 1967.

[25] The Apache Software Foundation. Xerces-C++ XML Parser. http://xerces.apache.org/xerces-c/, June 2012.

[26] Philip Hazel. PCRE - Perl Compatible Regular Expressions. http://www.pcre.org/, March 2012.

[27] Eric Sommerlade, Michael Feathers, Jerome Lacoste, J.E. Hoffmann, Baptiste Lepilleur, Bastiaan Bakker, and Steve Robbins. CppUnit Documentation. http://cppunit.sourceforge.net/doc/1.8.0/, June 2012.

[28] Yann Ollivier. Mathematical expression parser in C++. http://www.yann-ollivier.org/mathlib/mathexpr, September 2008.

[29] IEEE Standards Board. IEEE Standard for Software Unit Testing: An American National Standard, ANSI/IEEE Std 1008-1987. Technical report, The Institute of Electrical and Electronics Engineers, 1986.

[30] Jülich Research Centre. JUQUEEN - Jülich Blue Gene/Q. `http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUQUEEN/JUQUEEN_node.html`, July 2012.

Appendix A

Detailed profiling results

Nr.	System time	Duration [s]	Jobs	waiting jobs	running jobs
1	07/10/12-10:11:57	30	1204	965	239
2	07/12/12-13:37:47	16	941	696	245
3	07/13/12-13:10:51	56	1548	1234	314
4	07/16/12-12:24:14	39	1398	1189	209
5	07/16/12-12:47:33	44	1473	1223	250
6	07/16/12-14:44:19	56	1598	1403	195
7	07/17/12-16:32:50	65	1672	1354	318
8	07/18/12-08:42:13	40	1227	968	259
9	07/18/12-09:41:22	49	1385	1118	267
10	07/23/12-16:28:05	33	1047	785	262

Table A.1: overview of the example input files for the profile analysis in section 5.4.1

This table summarises global information of the example files used for profiling JuFo. The files are sorted by their system dates. The *Duration* column holds the total time in seconds required to simulate each input file. The *Jobs* column lists the total number of jobs submitted to JUROPA at the particular time, while the last two columns show the fraction of queued and already dispatched jobs.

In order to profile JuFo ten LML example files gathered from JUROPA are analysed. The table A.2 lists the core functions of the simulation and the percentage of the total simulation time spent in each of these functions and the child functions called by them. E.g. the second column contains the percentage of the total time spent in each of the listed functions for the first example file. The last column of the lower table comprises the average of each row's values. The *simulate* function is the template method of the *Simulation* class and executes the core simulation, which excludes parsing and marshaling the LML files, for example. All examples are configured to run backfilling as scheduling algorithm, so that the functions *insertWaitingJobs*, *updateBackfillWindows* and *findFirstFittingTimeSlot* are members of *BackfillingSimulation*. The *insertWaitingJobs* function simulates the insertion of queued jobs, which is the most time consuming function called by the *simulate* function. The process of generating backfill windows is conducted in *updateBackfillWindows*, while *findFirstFittingTimeSlot* searches for reservations of top dogs. The *sort* function prioritises currently waiting jobs and sorts them by the configured priority term. Finally,

Function name	**1**	**2**	**3**	**4**	**5**	**6**	**7**
simulate()	85.8	89.0	87.6	87.4	87.1	89.7	91.2
insertWaitingJobs()	85.6	88.6	87.4	87.3	87.0	89.5	91.0
updateBackfillWindows()	46.7	57.6	48.5	46.9	46.5	45.7	49.1
sort()	14.1	8.9	18.7	21.7	21.4	22.5	17.0
findFirstFittingTimeSlot()	3.0	9.8	5.6	3.7	4.1	3.2	7.9
isInsertable()	12.1	2.7	5.0	4.3	4.5	6.2	6.9

Function name	**8**	**9**	**10**	**Ø**
simulate()	89.6	90.0	93.0	89.0
insertWaitingJobs()	89.2	89.7	92.7	88.8
updateBackfillWindows()	53.5	51.4	59.8	50.6
sort()	18.2	18.1	10.7	17.1
findFirstFittingTimeSlot()	3.0	7.4	9.8	5.8
isInsertable()	6.0	4.8	4.6	5.7

Table A.2: profiling results listing the percentage of the total simulation time spent in the most time consuming functions of JuFo

isInsertable is a member of the *BackfillWindow* class. By calling the corresponding function of the resource manager implementation it checks, whether a job can be inserted into a generated backfill window. Note that *simulate* calls *insertWaitingJobs*, which again calls all lower functions. I.e. the time values for the *simulate* function include the time spent in *insertWaitingJobs*.

Appendix B

Detailed test results for JUROPA

In order to provide a more detailed view of the error distributions gained from the test examples of JUROPA in section 7.3, boxplots of the errors are displayed in figure B.1.

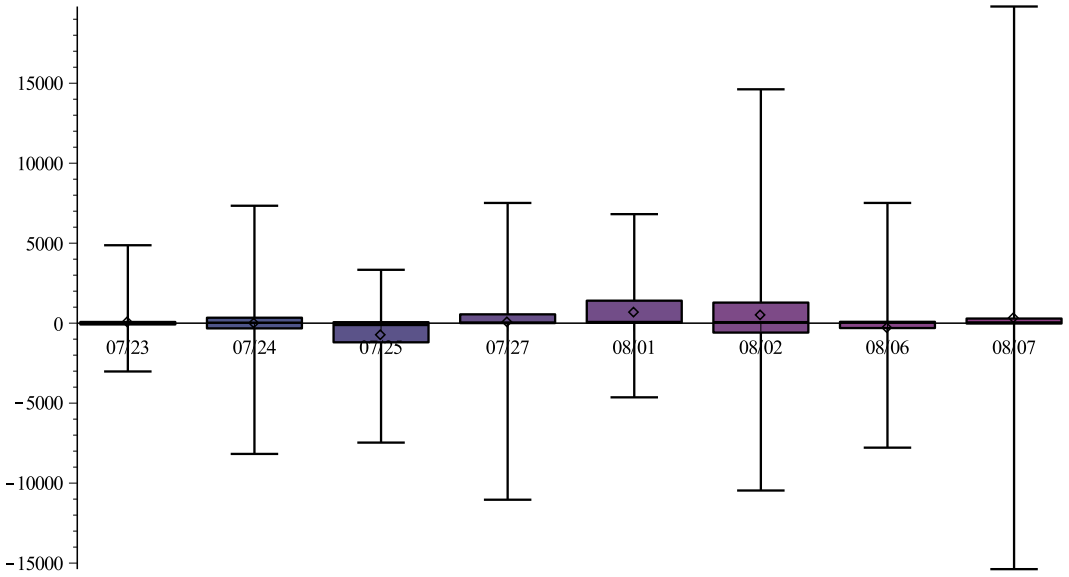

Figure B.1: boxplots for error distributions of example test files in section 7.3

To each boxplot the date of the corresponding test run is attached. The coloured boxes range from the first to the third quartile. Thus, each box encircles the range of the middle 50% of all errors for the particular test run. The whiskers indicate the maximum and minimum error values. Since the boxes are concentrated around the zero error value, the major part of the errors is small. However, the large range of the errors increases the variance. To conclude, the boxplots demonstrate that most of the jobs are scheduled nearly correct, while the existing outliers of inaccurately scheduled jobs cause the large variances of the error samples.

Jül-4354
September 2012
ISSN 0944-2952