3 Workflow Concept of WS-PGRADE/gUSE

Ákos Balaskó

Abstract. This chapter introduces the data-driven workflow concept supported by the WS-PGRADE/gUSE system. Workflow management systems were investigated by Workflow Management Coalition, among others in aspects of implemented data flow structures, and several workflow patterns are identified as commonly used and meaningful workflow structures. The workflow concept of gUSE is shown by introducing the supported data patterns and illustrating their creation in the system. Moreover, the possibilities of utilizing parallelization techniques are described, and then the different views of a workflow design and management are described covering the whole lifecycle of a workflow development. Finally, more complex composition of patterns and their creation techniques are shown.

3.1 Introduction

Scientific workflow management systems are widely known and highly accepted tools to connect stand-alone scientific applications and/or services together to access, process, filter, and visualize scientific data in an automated way. In most cases the composition of these steps is not a straight-forward process, moreover usually there is huge amount of data to be processed and stored remotely. In addition, the application can take a long time to deal with the data, which requires parallel processing to achieve results in a reasonable time. Another case is when the input data drives the interpretation of the workflow, and different applications should be executed depending on the input content. Such concepts indicate that formal investigations are needed to avoid adhoc – hence generally not optimal – solutions.

In technical terms a workflow or, as it also is known, a workflow composition means set of applications or entities connected to each other in order to process a complex algorithm in cooperation. A workflow engine or enactor, is capable of interpreting the workflow, identifying its nodes, and making decisions about which nodes can be executed according to the data dependency and environmental circumstances. Workflow management systems (WfMS) consist of such interpreters for workflow enactment and other additional tools that support the execution of the workflow (e.g., handling data transfer transparently).

Research on workflow management belongs to the area of service compositions [Dustdar/2005]. This area can be divided into two widely known fields concerning the enactment type: service orchestration and service choreography [Peltz/2003]. Service orchestration defines the workflow enactment and thus decides which jobs

are executed according to the workflow structure. It may be done in an adaptive way, namely taking the current state of the computational resources into consideration. Workflow management is definitely based on this concept. In contrast, Service choreography uses the idea of distributed enactment, where the enactment decisions are made by the workflow nodes; hence bottleneck issues caused by the single point of the enactor are resolved. To conclude, via WfMS tools scientists are able to design, manage, and reuse their own experiments executed locally on their own machine or by utilizing remote computational and/or storage facilities.

Widely known WfMS are Taverna [Hull/2006], Kepler [Altintas/2004] [Ludascher/2006], Triana [Taylor_Triana/2007], Pegasus [Deelman/2005], ASKALON [Fahringer/2007], and Galaxy[Goecks/2010] [Blankenberg/2010] [Taylor_Triana/2007]. Moreover, scientists can share workflows as good examples with their colleagues thanks to online workflow repositories such as MyExperiment [Goble/2007] or SHIWA Workflow Repository [Korhov/2011]. By understanding the importance of scientific workflows in scientific research, WS-PGRADE/gUSE was designed as a workflow-oriented science gateway framework where the most essential component of the system is the workflow engine. All the other components extend the workflow feature and are responsible for making its usage easier and more convenient.

3.2 Syntax of gUSE Workflows

Besides the term workflow, the other important term is the node (or job) that represents one particular stand-alone entity of computation (executable, web-service invocation, etc.). A workflow composition consists of connected nodes.

Ports represent data in many-to-one association with a node (namely many ports can be added to a node, but every port must only be added to one node). Types of ports can be set “in” or “out” denoting that the port represents a required input data, or an expected output data. Ports can be connected to each other defining the dataflow; thus, only ports of different types can be connected. Hence, this separation of ports defines an implicit semantics for the enactment of connected nodes.

gUSE applies an XML-based language for defining workflows and their graphical representation, including the structural configuration as well as execution information. Thus, the XML description of a gUSE workflow consists of three parts defined as tags: “graph”, “real” and “instances”. The “graph” tag defines the workflow as a set of nodes with associated ports including x and y coordinates for the canvas of the Graph Editor. The “real” tag contains all configuration information for the jobs and for the ports as well, extended by a history record that keeps track of changes in arguments. The “instances” tag stores reference records and brief status information about the executed workflow instances. At design time, the Graph Editor (introduced in Chap. 2) and WS-PGRADE UI ease the creation and the editing of workflow descriptions for the user.

3.3 Workflow Patterns

The Workflow Patterns Initiative (workflowpatterns.com) keeps track of scientific papers citing workflow pattern-related work (until 2009 but there are many more) showing that workflow patterns became a widely accepted way for designing and for re-factoring workflow applications. Taverna is used for investigating solutions for parallelism and pipeline processing [P. Missier/2010]; others such as Yet Another Workflow Language (YAWL) [Van Der Aalst/2005] or [McPhillips/2009] are specified directly using workflow patterns. Patterns play roles in [Yu/2005] to set up taxonomy for workflow structures. Since gUSE uses a data-driven workflow language, we focus on the identified dataflow patterns only.

3.3.1 Dataflow Patterns

Various pattern classes are identified in [Russel/2007] for data-driven workflow management systems such as *visibility of data, internal* or *external data interaction* or *data-based routing*. Several patterns are introduced clearly and are detailed to show the different cases within a class. For instance, the data visibility class contains patterns describing different scopes for accessing the data from the tightest task level until the widest environment level. The data interaction class incorporates the possible ways to communicate between tasks, focusing on their additional properties, e.g., communication with a task that represents multiple instances, or a subworkflow decomposition. In the followings we introduce those patterns that are supported by WS-PGRADE/gUSE system. Such patterns that are supported explicitly are shown in Fig. 3.1.

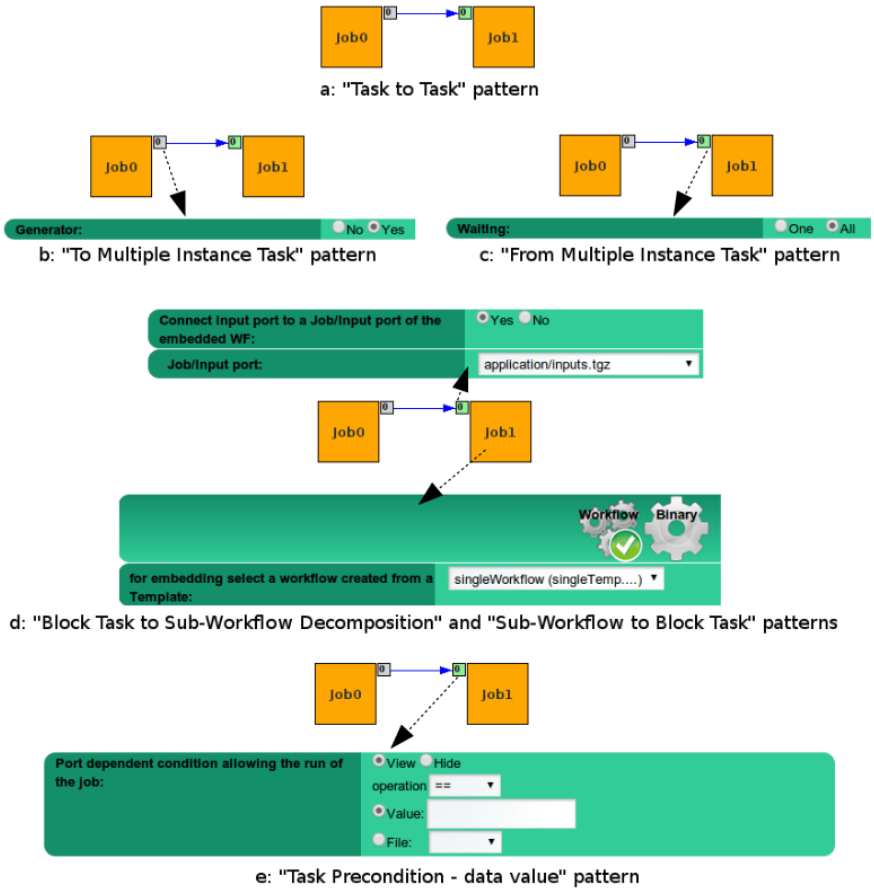

Fig. 3.1 Dataflow patterns supported by gUSE

In the class of data visibility, the task data pattern is supported by gUSE language only. Task data interprets the strictest possibility; it allows accessing data only for the job similarly to the definition of private variables in the object-oriented programming languages.

The internal data interactions pattern class investigates data transfer and their orchestration among the tasks. The simplest pattern supported by all of the data-driven workflow management systems is called task to task pattern (Fig. 3.1) specifying that data can be passed between the tasks. According to how the control and the data channels are used, three cases are defined: using data stored accessibly globally from/to where the jobs can read/write; use integrated channels for control and data transfer ; or using distinct channels for sending control and data information.

Integrated channels mean that the all the data generated by a job and the controlling actions travel together to the next job. A disadvantage of this solution appears in the case when part of the data set is not needed by the next job, but is required by one of the subsequent jobs. In this case, the data come through the intermediate job in vain. By contrast, transferring in distinct channels resolves this problem by using unique channels among source and sink jobs. This latter pattern is supported by gUSE.

The multiple instance task (TMIT) pattern and from multiple instance task (FMIT) pattern (Fig. 3.1b and 3.1c, respectively) specify coherent interpretation methods; therefore they are usually supported in pairs. Both patterns are defined among two connected tasks. While TMIT covers the situation of defining the data transfer if the subsequent job is going to be executed in multiple instances in parallel, FMIT focuses on the case when multiple jobs precede the single job. TMIT has three subpoints depending on the data partitioning and their access: (1) shared data accessible by references, (2) instance-specific data accessible by value or (3) instance-specific data accessible by reference.

A gUSE dataflow requires and generates data as files, which leads to the conclusion "TMIT with instance-specific data accessible by value pattern" is supported by gUSE via the concept of generator port types described in detail in Sect. 3.6.1. Nevertheless, in specific cases, when remote data storage systems are used, the other patterns are supported as well, meaning that access to remote data for manipulation means downloading a local copy of it. Therefore, the data manipulation does not take effect straight away on the shared data item, postponing, but not resolving consistency issues. The FMIT pattern is implemented using the concept of collector ports in gUSE.

Similarly to the previous patterns, the next two patterns are symmetric and are mostly implemented in pairs. In general, they are based on a modified job definition that allows the nodes to represent workflows as well. In this point of view workflows can be used as subworkflows triggered by a job submission that covers the sub-workflow in the outer workflow's point of view. Nevertheless, by definition subworkflows are the same as normal workflows. These patterns are allow to specify data transfer between the representing node and the subworkflow, and vice versa.

Block task to subworkflow decomposition (BTSWD) specifies transferring data into a subworkflow, while subworkflow decomposition to block task (SWDBT) specifies the opposite direction. Both are supported by the concept of templates in gUSE (Fig. 3.1d), introduced and detailed in Sect. 3.6.5.

The class of data transfer patterns contains patterns that focus on the different types of data transfer among the nodes. The pattern named data transformation – input/output describes the possibility to transform the incoming data before processed by the application, or to transform the data generated after the execution of the application. gUSE supports these patterns implicitly. Instead of simply executing the applications, a wrapper script is executed to set up the right environment. It copies the input files, manages the execution of the required application, and then

handles the generated outputs according to the type of output channel. Thus, it sends generated files back or stores them remotely and uses references considering the number of data sets in the case of generator port. The data transfer by reference – unlocked pattern is supported if the output files are stored remotely. In this case, just a reference is retrieved back to the portal. Since it does not deal with synchronization, the consistency of the remote data cannot be guaranteed; as a result, the latter modifications overwrite the former ones.

Last but not least, the class of data-based routing collects several cases when the existence or the value of the data affects the workflow's further interpretation such as task pre- and post-condition considering the existence and value of the data, or data-based routing. The task precondition – data value pattern (Fig. 3.1) describes the case when the job can be run, or can be blocked depending on the value of the incoming data. gUSE precisely covers this situation by introducing the concept of the "port-dependent condition" detailed in Sect. 3.6.4.

3.4 Levels of Parallelism

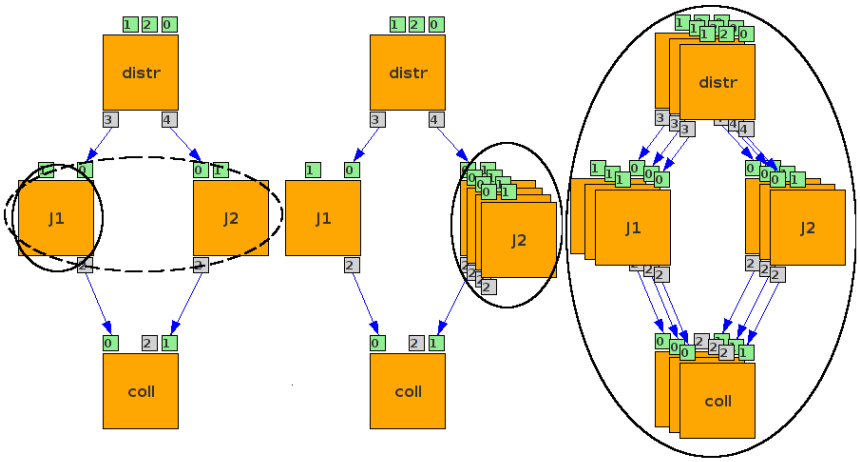

Fig. 3.2 Levels of parallelism

Depending on the execution of the jobs and workflow settings, four levels of parallelism can be identified in a WS-PGRADE/gUSE workflow. The lowest level, or node-level parallelism denoted as "J1" circled in Fig. 3.2, is where the application itself is prepared to utilize the benefits of multicore processors or cluster systems. In multicore environments these applications are usually designed as multithread applications like GPU programs. In the case of cluster systems, the applications use specific programming libraries that implement MPI specifications (such as OpenMPI). Besides this option, gUSE supports parallel execution of dif-

ferent jobs placed at different parallel branches of the workflow graph as the most intuitive and simple concurrent execution. It is denoted by J1 and J2 circled together in Fig. 3.2 and is called branch-level parallelism. A third level of parallelism covers the situation when one algorithm should be executed on a large parameter field, generally called parameter study or parameter sweep (PS) execution. This scenario is illustrated in the middle part of Fig. 3.2 and called PS parallelism, and the node that can expose such a feature is called PS node. Various opportunities support this level of parallel execution in gUSE, such as allowing generator ports and defining various parameter field generation methods. These possibilities are discussed in Sect. 3.6.1. Since any node can be an embedded workflow in gUSE, the PS node can also be an embedded workflow. We reach the highest level of parallelism (shown in the right hand-side of Fig. 3.2) where the execution of the same workflow is done in parallel. In fact, such a parallel execution of workflows can also be initiated by the user submitting the same workflow with different configurations (in gUSE terminology, in different instances).

3.5 Workflow Views

According to the design phases, workflows have different views separated by their focus. During design time the users are allowed to modify the structure, to add parameters to the jobs or to set the execution resources. Besides, execution management requires different views to be able to check states of the certain jobs, to get their outputs or standard output/error messages and based on them, make decisions to cancel or continue the execution. These views are introduced in this subsection.

3.5.1 Design Time – Abstract Workflow View

As a first step of designing a workflow in gUSE, the users define its abstract graph, using a web-start client-based application called Graph Editor. Fig. 3.3 illustrates a graph created in Graph Editor. As discussed briefly in Chap. 2, it offers a clear graphical interface for defining the nodes representing the component applications within the workflow, and in addition, to associate port entities for each job, name them, add a short optional description about them, and select their main type (input or output). These ports can be connected to each other reflecting that the data generated by the source node and associated to the given port must be transferred to the sink node. Evidently, an output port can be connected to several input ports, but an input port must be in connection at most to one output port. If an input port is not connected to any output port, it represents an input file that should be provided by the run-time environment where the workflow is executed. For example, it could be file stored on an FTP server.

However, this obvious semantics of graph creation would allow the creation of circles in the structure. As this class of structures is not supported by gUSE, loops are recognized and blocked by the Editor. Therefore, only directed acyclic graphs

(DAG) can be defined in gUSE. Finally, the graph can be saved on the portal server. The editor is an intuitive graphical tool to define the abstract part (the graph tag) of the workflow description.

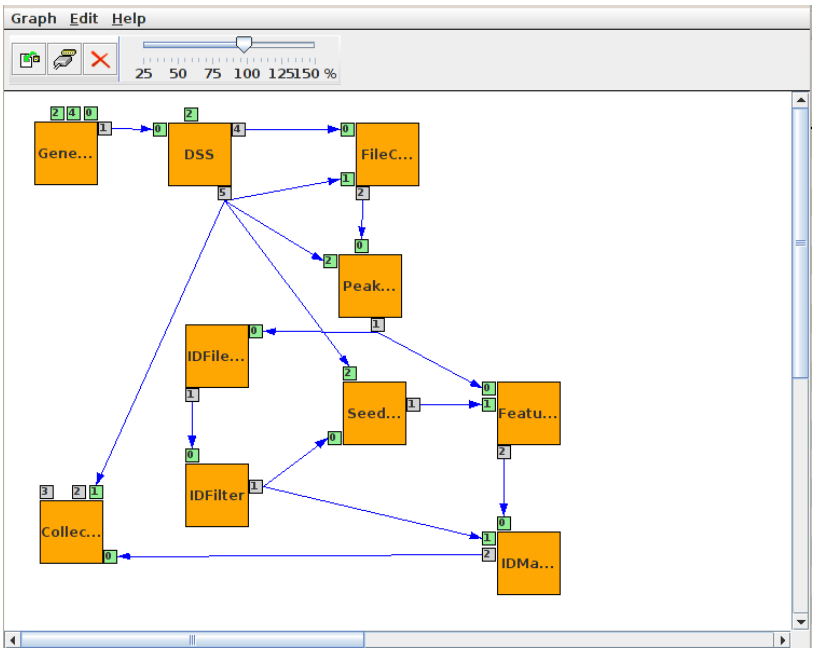

Fig. 3.3 A graph in Graph Editor

3.5.2 Design Time – Concrete Workflow View

Once the graph view of a workflow is created, the next step is to configure it with all those properties that define the circumstances of the concrete execution. Therefore, the result of the configuration phase is a concrete workflow. In the configuration phase the user should define the semantics of each workflow node, the files to be consumed and produced, as well as the infrastructure where the given node should be executed.

Figure 3.4 shows the possible options for a node configuration. As it can be seen, in the root the main decision to make is the type of node that can be set to represent a job (application binary), a service (service), or a workflow. Considering the learning curve of a job configuration, the simplest way is to invoke a service. gUSE language offers several ways to declare a service invocation: it can be set as a SOAP using AXIS implementation, a simple HTTP invocation, or using REST. Some options require specifying the method to be called (such as GET or PUT in case of REST). Users are allowed to send inputs encoded in the URL or as

files, of course, besides the returning stream are mapped to the first output port of the job constantly.

In most cases, users own their applications, and they would like to execute them on remote computational resource. gUSE, and its submitter component called DCI Bridge, support many different types of resources having different features, therefore – as seen in Fig. 3.4 – jobs must be configured according to the selected resource.

For instance, grid middleware types, such as gLite [Laure/2006], or different versions of Globus Toolkit [Foster/1997], require job descriptions written in JDL[JDL] format. Hence there is a possibility to add arguments fitting the JDL schema for those jobs, but these settings won't be enforced if the job's resource is configured, for example, for PBS[Henderson/1995].

Job configurations regarding the middleware types are not coherent; in some cases the resource requires us to specify the concrete computational resource where the job will be submitted (for instance, Globus Toolkit). Some others just need a broker server to be defined (e.g., gLite). On the other hand, the application itself influences the difficulty of the configuration, too. For example, if the algorithm is an MPI application, we need to define this property and then we must set the required number of processors for the execution as well.

In theory, the most complicated case is if the job refers to a workflow. To achieve this scenario is quite simple: only the required workflow's link should be selected for the job. We remark that only those workflows can be embedded that are inherited from a template. Then, as shown in Fig. 3.1d, the input and output ports of the embedded workflow must be connected with the container job's input and output ports, respectively, and to establish the information channel between the workflows.

The bottom of Fig. 3.4 shows possibilities of port configurations are with respect to their main "ordering" property. They can be set to be input or output ports. According to this setting, several properties can be defined related to the port, for example, the internal input/output file name (the file name required by the application, or needed to be generated by the application), the destination of the file in general (local or remote), and so on. To do all of these configuration steps, the WS-PGRADE user interface provides a handy configuration panel.

.

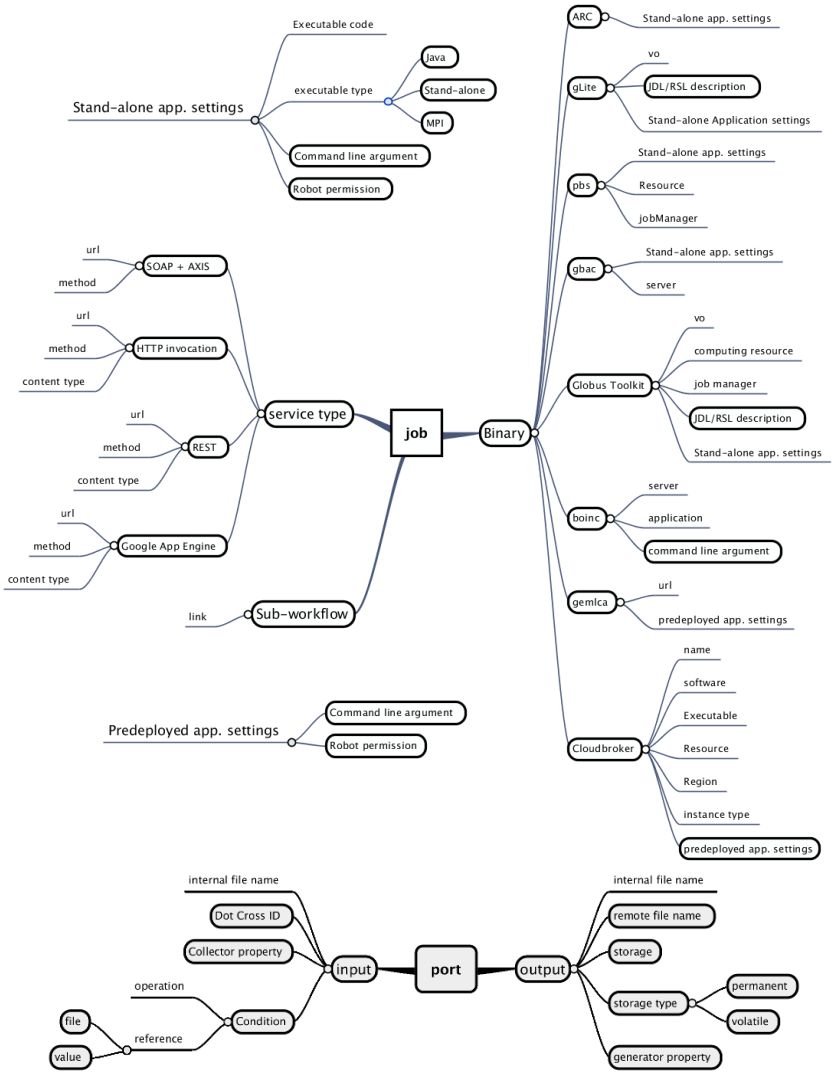

Fig. 3.4 Job and port configurations

3.5.3 Execution Time – Workflow Instance View

After designing a concrete workflow, it can be submitted for interpretation to the gUSE Workflow Interpreter (WFI). The same concrete workflow can be submitted several times in parallel, generating multiple instances of the configured workflow. Workflow instances reflect the enactment of the concrete workflow's configuration at runtime regarding the number of job instances generated dynami-

cally from PS nodes, and their execution states on remote resources mapped to status information, such as “submitted”, “running” or “finished”. Figure 3.5 illustrates the complete state diagram of the execution of a workflow instance, where the lines represent the actions performed (lines with circles denote automatic status changes), and rectangles represent the certain states.

By definition workflows stay in the “init” state. If the “delete” action is invoked, the workflow’s state is set to “deleted”, triggering its deletion automatically, and the process ends. Another option is when the “submit” action triggers, changing the state to “submitted”, then, supposing that DCI-Bridge has submitted the job correctly, the state will be changed to “running”. In the “running” state several automated status changes are possible, depending on whether the configuration is correct. Finally, the state can be changed to “finished” or “error”. The “Delete” action can be called in both states, resulting in “delete” state, while a “resume” action can be invoked in “error” state only. Resuming a workflow effects its submission as a new workflow instance again, excluding the jobs executed correctly earlier, so through a temporary “resuming” state the workflow will be in “running” state again. Nevertheless, users have the option to stop the execution any time when the workflow is in “running” state by clicking the “suspend” button. This action changes the state to “suspended”, in which the workflow can be deleted or resumed.

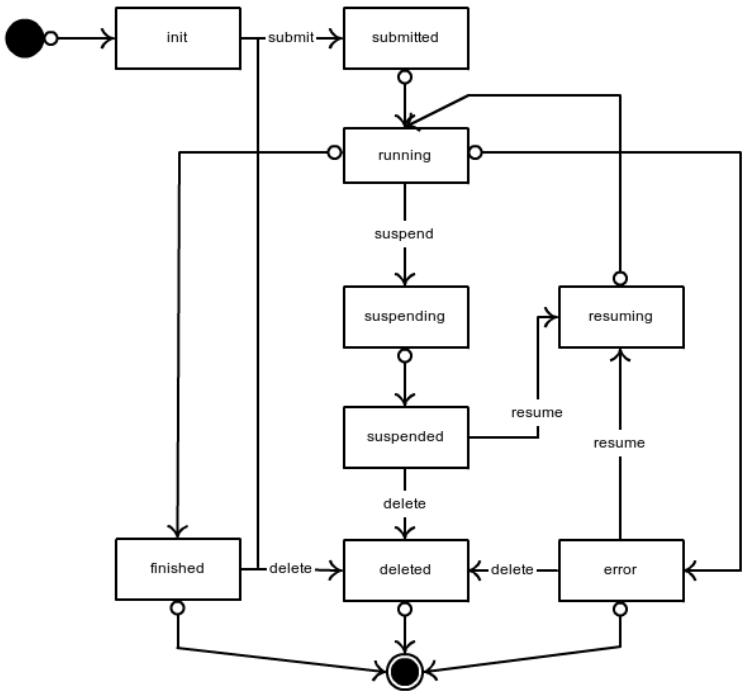

Fig. 3.4 State diagram of a workflow execution

As is illustrated in Fig. 3.5 the workflow enactment and all of the possible actions can be taken via the web-based user interface.

Additionally, users are allowed to suspend those jobs that are currently being interpreted; hence the workflow instance can be reconfigured, and then the instance can be resumed with its modified configuration. To let the users change the configuration considering the former jobs' output, all the files generated by job instances can be downloaded and standard output and standard error outputs can be visualized immediately after their execution.

3.6 Features of gUSE to Support Dataflow Patterns

Based on the primitive workflow patterns introduced in Sect. 3.1 we can define more complex workflow patterns in gUSE.

3.6.1 Generator Property

Considering a higher level of parallelism, ports can represent sets of files instead of single files. In the case of output ports this means that the job will generate multiple files with the given internal file name prefix extended by a unique id, and an index started by 0 as their postfix. For instance, if the internal file name of a generator port is set to "output.txt", then the interpreter will require the set of files generated as output.txt_0, output.txt_1, etc. (Fig. 3.1b). We call a node containing at least one generator output port a generator node.

3.6.2 Collector Property

Input ports can be set to collect all the items of the output set fitting the proper file name prefix typed, and start one job instance only (if sets of other ports do not interfere with it). This behaviour is called the collector property and it implements Dataflow pattern shown in Fig. 3.1c. We call a node containing at least one collector input port a collector node. Notice that a collector input port should always be connected to the output port of a PS node, and its meaning is to collect the N individual output files produced by the N instances of the PS node.

3.6.3 Generating Input Datasets

Since multiple ports can be associated to a job, and as each of the ports can represent a set of files, we therefore need to define relations among these sets. Then the interpreter can count the proper number of job instances to be executed on each item of the generated parameter field. An obvious strategy is to create the Cartesian product of the file sets resulting in ordered pairs of each file selected in different sets. In general, $X(P_1,\ldots,P_n) = \{(p_1,\ldots,p_n): p_i{:}P_i\}$.

However, creating Cartesian products, or in gUSE terminology, cross products, covers the whole parameter field. gUSE system is able to generate fields following a different strategy called dot products denoted by ".". It means the pairing of inputs according to the common index of enumerated members of constituent input datasets. If the size of one constituent dataset is less than the size of the largest set involved in the function, then the shorter file sets ordered by their index will be repeated cyclically. Cross product operates on file sets, and dot product operates on the produced paired or multiply paired sets of files. Therefore setting ports in cross product relation has greater precedence than setting them in dot product.

In technical terms in gUSE, a cross product relation of two ports is defined by setting their CrossDot PID to the same number. This method identifies those ports (or sets of ports) on which the dot product method should be applied, since they have different CrossDot PID set.

The following example illustrates a complex parameter field generation.

Assuming four ports, $P_1=\{a_1,a_2,a_3\}$; $P_2=\{b_1,b_2\}$;$P_3=\{c_1,c_2,c_3\}$; $P_4=\{d_1\}$, connected as follows: $(P_1xP_2).(P_3xP_4)$. This connection can be made in gUSE by setting CrossDot PID for P_1 and P_2 to 1, and for P_3 and P_4 to 2. Due to the greater precedence of the Cartesian product, those methods are applied, and P_1xP_2 produces the following pairs:

(a_1,b_1)	(a_2,b_1)	(a_3,b_1)
(a_1,b_2)	(a_2,b_2)	(a_3,b_2)

P_3xP_4 produces these: (c_1,d_1); (c_2,d_1); (c_3,d_1). Then the dot product is performed, resulting the final parameter fields as follows:

(a_1,b_1,c_1,d_1)	(a_2,b_1,c_2,d_1)	(a_3,b_1,c_3,d_1)
(a_1,b_2,c_1,d_1)	(a_2,b_2,c_2,d_1)	(a_3,b_2,c_2,d_1)

3.6.4 Port-Dependent Conditions

Last but not least, we must mention an interesting feature of input ports, called port-dependent conditions. Such conditions enable or block the submission of a job instance, depending on the value of the incoming data in comparison to a predefined value added as text or as a content of a file uploaded to the portal server. Moreover, the users can specify the relation that should be performed as comparison function; they can select "equal", "not equal", or "contains".

For example, let us assume a job having one input port only in connection with a generator port on which 3 outputs come containing 1, 2 and 3 respectively. Now if the port condition is set to compare the incoming data with value "1" and the relation is set to "equal", then the one job instance will be executed with the input "1". If the relation is set to "not equal", then two instances will be submitted with "2" and "3".

3.6.5 Creating Subworkflows

To serve the demand of reusability, gUSE supports the possibility of embedding a workflow as a subworkflow into another one. In order to embed workflows, the template concept of gUSE workflow language has to be applied. Templates have been introduced to support the reusability of defined and tested workflows. To be more specific, three goals are envisaged:

1. Simplified redefinition of a workflow;
2. Type checking of the "plug in" ability of an embedded workflow, where the embedding is made more secure by requiring that an embedded workflow must be inherited from a template;
3. Assistance in creating the simplified user interface for the common user automatically, on the basis of a template description.

A template is an extension of a workflow definition, in such a way that each configurable job or port-related atomic information item is extended at least - by an immutable Boolean value. The values of these Boolean metadata are mentioned as "Free" and "Close", respectively. "Close" means, that the related atomic configuration information item is immutable, i.e., in each workflow that references a given template, the "Close"-d atomic configuration information item is preserved, and it cannot be changed during the workflow (job) configuration process. "Free" means that the value of the related atomic configuration item is copied as default, but it can be changed by the user. Related to the "Free" state of the Boolean value, two other metadata can (and should) be defined: The first is a short string label identifying the given piece of information that can be changed, and secondly, the optional description may give a detailed description of the usage, either syntactically or semantically. Please note that the workflow configuration form used by a common user is generated upon these metadata.

Templates can be used for different goals in gUSE. Loops are technically equivalent with recursive invocation of functions. Hence – in this way – loops in workflow structures can be interpreted as recursive workflow invocations. Considering that a recursive invocation is mainly a particular point within a method on which the method calls itself again, in the case of workflow languages it means that the workflow itself is embedded as one of its jobs. In gUSE, workflows inherited from templates can be associated as jobs of a workflow, and if this workflow and the embedded one are the same, a recursive implementation is defined.

Templates technically define which arguments can be modified in the future and which cannot. This is a quite useful step if the users share their workflows and would like to guarantee the workflows' functioning. Users can create a template from the workflow by disabling the further modification of arguments and enabling just those, which can be configured by others.

Moreover, a possibility of generating a web-interface automatically hiding the complexity of the workflow and allowing inexperienced users to use it is based on templates in gUSE, too.

3.7 Pattern Compositions

Based on the previously introduced implementations of data patterns, in gUSE more complex control patterns can be created by composing and specializing them. Control patterns are identified by [Van Der Aalst/2005], and they investigate several commonly used open-source and commercial workflow management systems. In the following we introduce composition of control flow patterns and their implementation in gUSE using the features designed for data patterns.

3.7.1 Parameter Sweep Sequence

One of the simplest compositions of control patterns is the commonly used and widely known parameter sweep (PS) sequence. It is an ideal structure if large separable and therefore parallelizable parameter fields must be elaborated by the same application. It can be designed as a composition of "multiple instances with a priori run-time knowledge" pattern, which specifies the job generalization according to the number of data sets, and the "general synchronizing merge" control pattern that defines a structure for collecting the results of the parallel execution. Using the generator and collector port properties, we can define the frequently used PS workflow pattern consisting of a generator node, a PS node, and a collector node (Fig. 3.6). The role of the generator node is to split the large input data file received in input port 0 into a number *N* of small size data files and pass them to the PS node via its generator port (output port 0). The workflow interpreter will generate from the PS node as many jobs (PS node instances) as there are small data files (*N*) generated by the generator node. All these *N* PS node instances generate one output file. These *N* output files are gathered via the collector input port of the collector node and are processed by the collector node. Of course, this complex pattern can be further extended, enabling more generators, PS-nodes, and collectors in many different combinations.

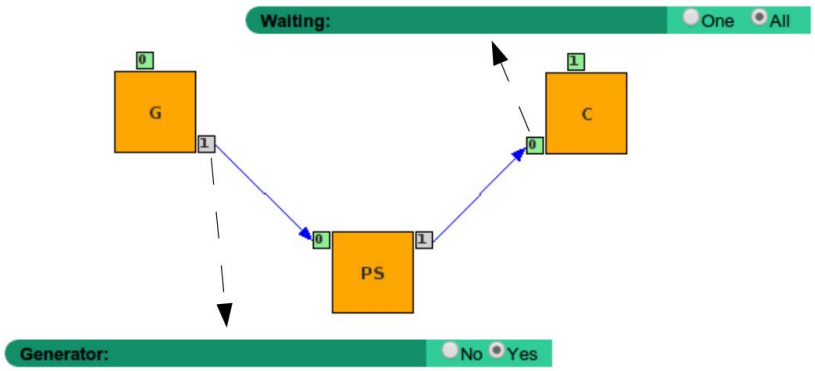

Fig. 3.5 PS workflow pattern consisting of a generator node, a PS node, and a collector node

It may be misleading, so we remark that, there is a specific case when multiple generator jobs connected together generating *N*M* instances (*N* come by the first and *M* by each parallel *N* job instance, respectively). This would imply that they can be collected correspondingly, namely for collecting all the items, we need two jobs, both with collector ports, but the truth is that the *N*M* data items handled in the same level will result in just one, but a greater set of data. Consequently, one job with a collector port gets all the data. If we wanted to handle the data split to their generating job accordingly, we should use the embedding feature of the system.

3.7.2 Enactment of Conditional Branches

Enactment of conditional branches is designed as a composition of deferred choice and simple merge control patterns, where deferred choice specifies two alternate branches representing “then” and “else” branches and a single control point before them. The control point chooses a branch to interpret, ignoring the other one. Subsequently, the simple merge pattern specifies the case of collecting alternate branches independently from what was interpreted or ignored before. Its implementation is shown in Fig. 3.6.

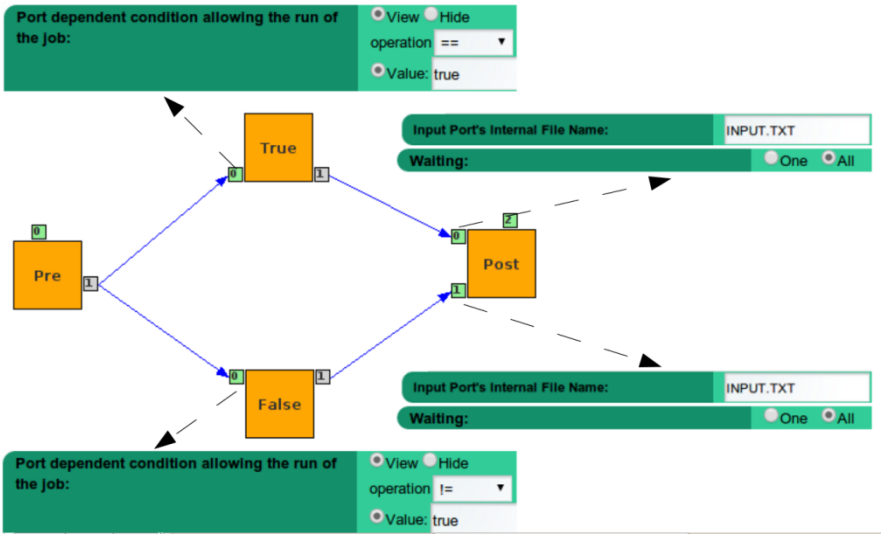

Fig. 2.6 Workflow with alternate branches

Although conditional job execution looks familiar, with the execution of complete branches depending on input data, there are some specific cases to consider at the point of merging these branches. First, for both of the alternate branches represented by jobs, "True" and "False" must have disjunctive data-dependent conditions (job "True" must equal value "true"; job "False" must not equal "true"). As was proven in [Balasko/2013b], this is equivalent to the deferred choice pattern. Then, to ensure the execution of job "Post" independently from the job executed before, which implements the simple merge pattern, both incoming ports of the job "Post" must be set to the same internal name (the alternate executions produce the same output file name). It must be set to have the collector property that allows job submission against a set of files, including the case of that set being empty. Otherwise, following the general semantics, job "Post" would not be submitted since it requires an input from both of the branches, including the disabled branch as well, which does not produce any inputs.

3.8 Sharing and Archiving Workflows

At the end of the lifecycle of a workflow, its developer has the privilege to share his/her workflow with colleagues who work on the same portal (so, within the community), or enabling its use by others for different communities (called cross-community sharing). Both ways are supported by WS-PGRADE/gUSE. In the prior case, the users export their workflow to the internal repository, which publishes it to all users after importing it in the same portal server. The latter case is based on the integration of a third-party solution for storing workflows that are accessible world-wide, called the SHIWA repository. It follows a more sophisti-

cated solution. During the export process the users can choose which arguments or input files are mandatory for the execution, and which can be changed. This information indicates a more clear view for an exported workflow than what an internal repository offers. It is discussed in detail in Chap. 9.

Besides sharing workflows, they can be also downloaded to be moved to other portals without sharing. During this process the workflow graph and the configuration are collected and stored as the XML introduced in Sect. 3.2. Together with the input files and binaries uploaded, it is compressed as a single zip file. The users are free to download the whole workflow with or without all the executed workflow instances.

3.9 Conclusions

In this chapter we introduced the features and the capabilities of the gUSE workflow language, in which the workflow applications can be defined. All the features and all the common workflow structures were described using Dataflow patterns. We separated the workflow design and management processes into three phases: the design-time step one, when the abstract workflow is created; the design-time step two, when the previously created abstract workflow is configured; and the run-time workflow management. Finally, we discussed the last step of the workflow's lifecycle, when the workflow developers can upload and publish their ready-to-use workflows in a workflow repository and the community can reuse these workflows.