Scientific Programming 15 (2007) 283–297
IOS Press

# Conditional workflow management: A survey and analysis

Emir M. Bahsi, Emrah Ceyhan and Tevfik Kosar
*Center for Computation and Technology (CCT) and Department of Computer Science, Louisiana State University, Baton Rouge, LA 70803, USA*
*E-mail: {embahsi, eceyhan, kosar}@cct.lsu.edu*

**Abstract.** Workflows form the essential part of the process execution both in a single machine and in distributed environments. Although providing conditional structures is not mandatory for a workflow management system, support for conditional workflows is very important in terms of error handling, flexibility and robustness. Several of the existing workflow management systems already support conditional structures via use of different constructs. In this paper, we study the most widely used workflow management systems and their support for conditional structures such as *if, switch, and while.* We compare implementation of common conditional structures using each of these workflow management systems via case studies, and discuss capabilities of each system.

Keywords: Workflows, conditional structures, conditional workflow management

## 1. Introduction

As the complexity of scientific applications increases, the need for using workflow management systems to automate end-to-end processing of applications increases as well. Static workflow management systems that do not provide any complex constructs such as conditions and loops become insufficient in addressing the needs of complex scientific applications. For instance, failure in one part of the workflow may cause the whole workflow to fail if there is no alternative branch. This problem could easily be elevated by the use of a condition and an alternative task to be executed in case of the failure of the original task. As a more specific example, if a file transfer task fails, the same file could be transferred using another protocol, or from another source depending on the type of the failure. Or, if no alternative way of transferring the same file is available, the file can be re-created using another task. This would require a conditional structure in the workflow. As another example, we may want to repeat a simulation (which consists of a sequence of tasks) until we get an error value between desired limits. This would require support for a condition as well as a loop in the workflow management system.

Several existing workflow management systems support conditional structures at some extent. Each of these systems implements these structures in different ways. While some systems introduce *if* and *switch* type of structures with which we are familiar from high-level programming languages, some of them introduce simple logical elements instead, which potentially could be used to implement higher level constructs. In the latter case, users must create their own conditional structures by connecting those logical elements with other elements.

In this paper, we study the most widely used workflow management systems and their support for conditional structures such as *if*, *switch*, and *while.* The systems we have studied include Apache Ant [4], ASKALON [23], DAGMan [18], GrADS [16], Gridbus [19], ICENI [5], Karajan [14], Kepler [15], Pegasus [9], Taverna [1,24], Triana [13], and UNICORE [8]. Among the systems that support conditional structures, we choose six of them, and perform detailed comparison of their support for these complex constructs. In our comparisons, we use implementation of simple con-

ISSN 1058-9244/07/$17.00 © 2007 – IOS Press and the authors. All rights reserved

Table 1
Conditional structures in grid workflow managers

| Name | IF | Switch | While |
|---|---|---|---|
| Apache Ant | Y | Y | N |
| ASKALON | Y | Y | Y |
| DAGMan | N | N | N |
| GrADS | N | N | N |
| Gridbus | N | N | N |
| ICENI | Y | * | Y |
| Karajan | Y | Y | Y |
| Kepler | Y | Y | N |
| Pegasus | N | N | N |
| Taverna | Y | N | N |
| Triana | Y | N | Y |
| UNICORE | Y | N | Y |

Y: Supports.
N: Does not Support.
*: Not much information found.

ditional structures which would be required to solve above mentioned problems. In the end, we discuss capabilities of each system.

The outline of this paper is as follows. In Section 2, we present our survey on existing workflow management systems and their support for conditional structures. In Section 3, we illustrate implementation of conditional structures in each system based on our case studies. We discuss some of the key observations we had during this study in Section 4, and we conclude the paper in Section 5.

## 2. Support for conditions in workflow management systems

Our study has shown that four of the twelve studied workflow management systems do not support conditional structures at all. These systems are: DAGMan, GrADS, GridBus, and Pegasus. Each of the remaining eight systems supports at least one type of conditional structure. Table 1 shows each of these systems and which conditional structures they support. Although some workflow management systems do not support conditional structures, other structures they provide can be used to imitate conditions at a certain extent. An example to this is, using the pre-script constructs in DAGMan to imitate *if* statements. Below, we present the workflow management systems which support conditional structures in more detail.

### *2.1. ASKALON*

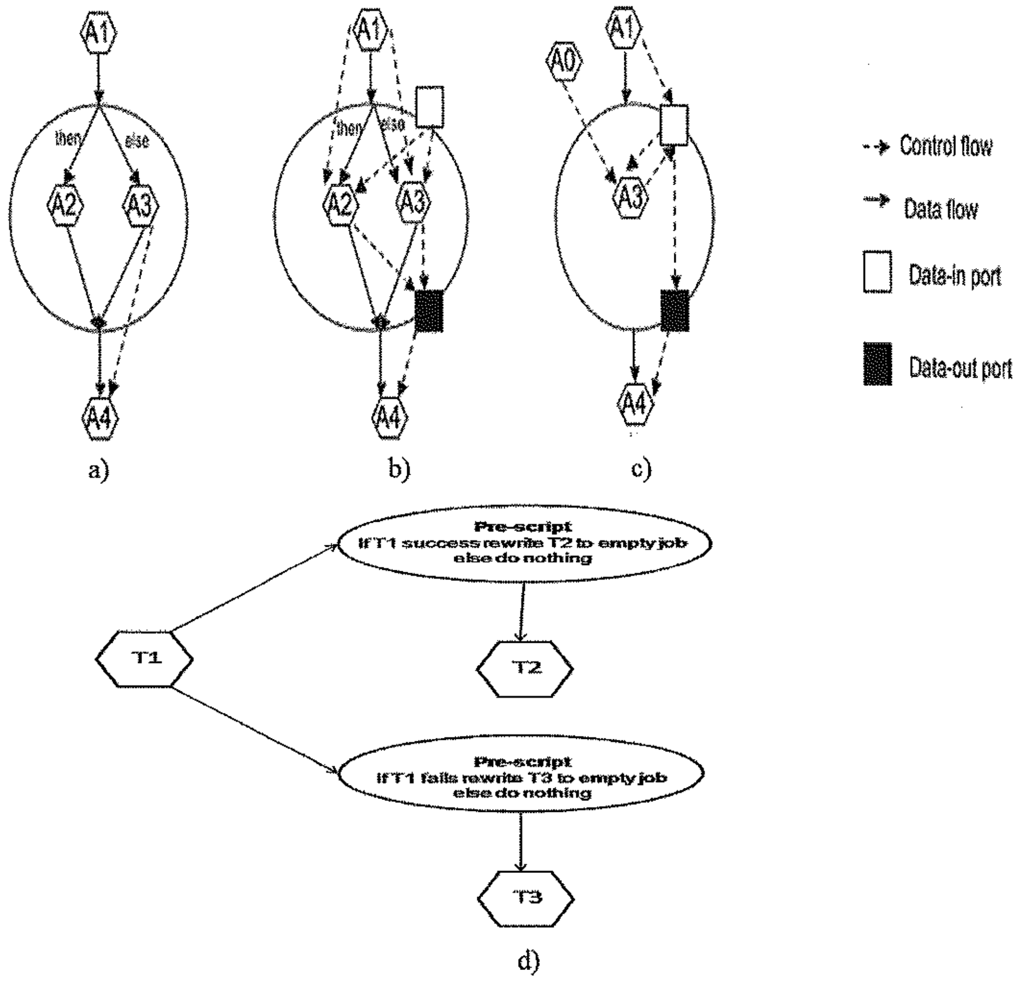
ASKALON is a grid application development and computing environment [23] based on AGWL [22], which aims to provide an invisible grid to the application developers. AGWL is an XML-based workflow language, which describes Grid workflow applications at a high level of abstraction. AGWL is located at the top of the ASKALON architecture, which is used for composition of grid workflows. AGWL supports both computational tasks and user defined tasks. These activities are connected by data and control flows.

AGWL supports two types of conditional activities: *if* and *switch* structures. Each activity has data-in and data-out ports. Figure 1a and 1b show two data flows of the *if* structure. The data flow is provided by connecting data-in and data-out ports to activities based on the control flow. However, control outcome of *if* or *switch* activity is not known at compile time. Therefore, which inner activity's data-out port should be connected to an activity outside of that conditional activity cannot be determined. As can be seen from Fig. 1b, this issue is solved by connecting all inner activities' data-out ports to the data-out port of the conditional activity and also connecting the data-out port of the conditional activity to the next activity that comes after the condition structure.

AGWL supports three loop activities: *while*, *for* and *forEach*. There is a conditional structure in the loop activity, which determines whether loop will continue or not. In addition to conditional structure, loop activity includes other structures. As in conditional structures, loop activities also include data-in and data-out ports. Figure 1c illustrates *while* loop in AGWL. The first activity of the loop is to retrieve data from the data-in port of the loop or from another activity outside the loop. The data-out port of the last activity of the loop is connected to the data-in port of the *while* structure. At the end of each loop iteration, the data in the data-in port is replaced with the data produced by the last activity of the loop. The data-in port of the loop is also connected to the data-out port of the loop structure. If loop terminates based on the result of the conditional activity, data packages of the data-in ports are mapped to the data-out ports of the loop. After the termination of the *while* loop, activities outside the loop can obtain the data from data-out port.

### *2.2. DAGMan*

DAGMan (Directed Acyclic Graph Manager) has been developed as part of the Condor project [18], and acts as the meta-scheduler for Condor. Given that there can be thousands of jobs that need to be executed in a certain order as part of an application; DAGMan han-



A1
then
else
A2
A3
A4
A0
Control flow
Data flow
Data-in port
Data-out port
a)
b)
c)
Pre-script
If T1 success rewrite T2 to empty job else do nothing
T1
T2
Pre-script
If T1 fails rewrite T3 to empty job else do nothing
T3
d)

Fig. 1. Conditional Structures in AGWL [22] – a) data flow in illegal form in *if* activity b) data flow in legal form in *if* activity c) *while* loop d) Imitating conditional dag in DAGMan [18].

dles dependencies between jobs and their execution orders.

DAGMan is a very simple workflow management system and does not provide any complex constructs such as conditions or loops. On the other hand, DAGMan provides other constructs such as pre- and post-scripts, which are lightweight processes run before or after a job's execution.

Although DAGMan does not support conditional structures, some users have discovered one usage of pre-scripts as imitating the functionality of conditional structures [18]. Based on the result of a previous DAG node, a pre-script can determine whether the node must be executed or not. An example to this is given in Fig. 1d. This usage in reality does not provide conditional execution of a task. It executes both tasks in either case, but simply overwrites the task which we do not want to execute with a no_op task which does not have any effect in the system.

## 2.3. *Triana*

Triana [13] is both a problem solving and a programming environment. Since it is written in Java, Triana can be installed and run almost on any system.

In Triana, composition of scientific applications can be done via its user portal. The graphical user interface of Triana is fairly convenient for users when creating workflows for their purposes. Users do not need to know anything about XML or Web Services Flow Language (WFSL) [10] to use Triana. They can compose applications by drag-and-drop mechanisms. Users can also drag programming units and tools from toolboxes, and drop them onto their workspaces. Units are connected drawing cables from one to another.

Triana supports two kinds of conditional processing: *if* and *loop* structures. Figure 2a and 2b show an example of how *if* and *loop* structures are implemented in Triana.

In the *if* structure shown in Fig. 2a, T1 sends a value to condition and then the value is compared with the value in condition. If the value in T1 is less than the value in condition, then T2 sends the data to T4, directing both control and data flow to T4, hence killing T3. On the other hand, if the value in T1 is greater than the value in condition, then T2 sends the data to T3, giving both control and data flow to T3, and kills T4.

In the *loop* structure shown in Fig. 2b, T1 sends a value to the condition. The value is first evaluated and if exit condition is met, then output of T1 is sent to T2 and T2 executes. Otherwise, output of T1 is sent to T3 to get a new value. After getting a new value, it is compared again with exit condition. This process continues until the value obtained from T3 is met to exit condition. One important point in this figure is that T1 sends a value only once. If the value in T1 does not match the value in condition, it keeps getting new values from T3 until the value from T3 matches the value in condition.

### 2.4. *Karajan*

Developed at the Argonne National Laboratory, Karajan is part of Java CoG Kit. It is derived from GridAnt [7] and provides additional capabilities such as scalability, workflow structure, and error handling [14]. It has two different syntaxes. One is CoG Kit K syntax with which we are familiar from other high-level languages and second one is CoG Kit XML which we explained in this paper.

Karajan supports *if* and *choice* constructs as conditional structures. Figure 2c illustrates how *if* structure is implemented in Karajan. *If* structure enables the execution of different tasks based on the results of some conditions. So a simple *if* construct can be formed by *if, condition, then, else,* and *elseif* elements.

*Choice* element of Karajan can also be considered as a conditional construct. It works similar to the *switch* statement which exists in many high-level languages. *Choice* element executes the inner elements sequentially. If an element fails to execute, then next element is executed and gains access to some variables about the previous task execution. These variables are: *element, error, stack trace,* the *exception* (if java exception is the reason of the failure). *Choice* element runs the inner elements until a successful completion is found. At that time, it does not execute next elements, but gives the control to the element that comes after *choice* element.

Karajan supports both *for* and *while* structures as loop constructs. *While* is executed until its condition element evaluates to false as shown in Fig. 2d [12]. *For* structure iterates for a range of values.

There are also some logic elements which help users to create conditions of both *if* and *while* structures. These elements are *and, or, not, equals, true,* and *false*. Conditions can be created by either choosing one of them or combining multiple elements.

### 2.5. *UNICORE*

UNICORE (Uniform Interface to Computing Resources) is a grid middleware, which has an open, service oriented architecture. Main goal of UNICORE is to provide seamless, secure, and intuitive access to distributed resources [8]. UNICORE provides to users a programming environment to design and execute their workflows. In UNICORE, the workflow is represented as a Directed Acyclic Graph (DAG).

UNICORE supports some advanced control structures, which are conditional execution (*if-then-else*), repeated execution (*do-n*), conditional repeated execution (*do-repeat*), and suspend (time conditional) action (*hold-job*). These control structures use three types of test conditions: *ReturnCode, FileTest* and *TimeTest.*

The *if-then-else* structure chooses one of two branches for execution at the run-time. It uses one of the three conditions mentioned above. If *ReturnCode* test is used, a dependency must exist between a task and the *if-then-else* construct. Client will check dependency and prevent user from submitting non-deterministic jobs.

The *DoRepeat* construct repeats a group of actions based on the result of a condition. It selects one of the three condition tests. This is same as *if-then-else* structure. If *ReturnCode* is selected, the result of an action in the body of *DoRepeat* becomes the *ReturnCode.*

The *HoldJob* construct waits for a specific amount of time before executing the task. It uses *TimeTest* condition.

The *DoN* structure is very similar to *for* structure in high-level languages. It has a counter named 'N', which determines the number of repetition. Users must set this counter before submitting the job. Therefore, this control structure differs slightly from the other con-

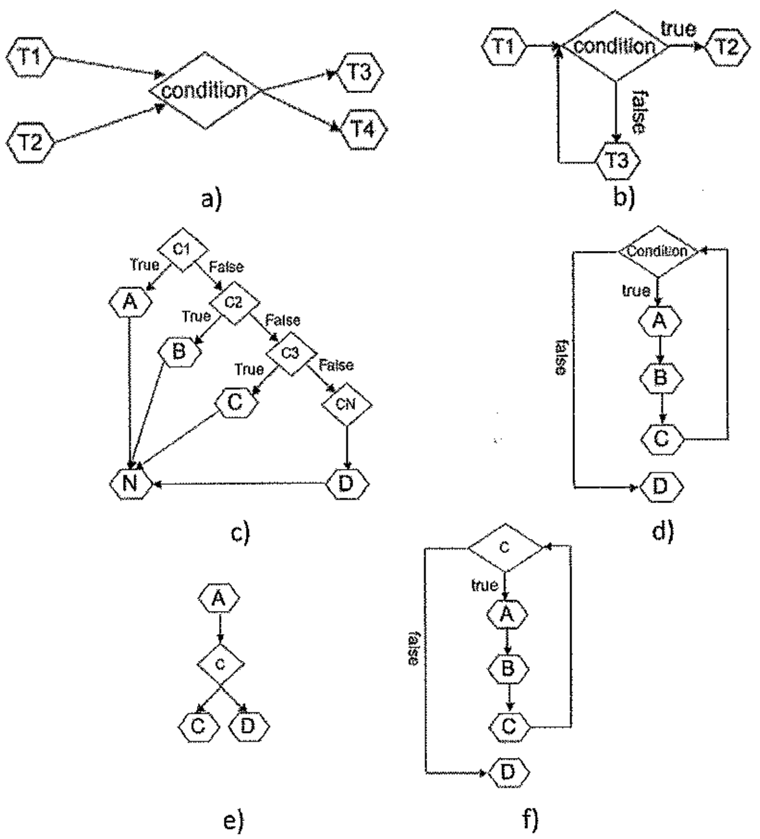


Fig. 2. Conditional structures in Triana, Karajan and UNICORE a) *if* structure in Triana b) *while* structure in Triana c) *if* structure in Karajan d) *while* structure in Karajan e) *if* structure in UNICORE f) *while* structure in UNICORE.

trol structures in that it does not use any condition because the conditions are determined at run-time.

The *ReturnCode* test uses result of a task as a condition for the next task. It compares the result of a task in three different ways to decide whether the next task will be executed or not. First method is to compare the result of the task with some specified value. If they are equal, next task can be executed. Second and third methods are successful execution and unsuccessful execution of the last task, respectively. *ReturnCode* test is very useful in terms of fault tolerance. When a task fails (which means becoming unsuccessful), neither an exception is thrown nor the whole workflow fails; instead, a new task is executed as an alternative.

By using *FileTest* condition, a task is executed according to a file's status. This status could be: file exists, file does not exist, file is readable, file is writable, and file is executable.

The *TimeTest* condition checks whether a specified time has passed or has reached to execute the task.

The implementation of *if* construct in UNICORE is shown in Fig. 2e. As can be seen from the figure, after the execution of condition, one of the tasks from C and D is selected for execution. Figure 2f illustrates the loop structure in UNICORE. It executes set of jobs based on correctness of the condition. Both *if* and *while* conditions could be one of the test conditions which are mentioned above.

## 2.6. *ICENI*

ICENI (Imperial College eScience Networked Infrastructure) provides and coordinates grid services for eScience applications. ICENI has a GUI construction tool which provides high-level abstraction for users to construct and define their own applications. This tool generates real execution plan in an XML format derived from YAWL (Yet Another Workflow Language) [11, 17,26]. ICENI workflow language supports conditions, loops and parallel execution.

ICENI has two compositions: spatial and temporal compositions. We are only interested in temporal composition since it represents the workflow of the application. Each component in the workflow is composed by collection of nodes. Nodes differ according to their behaviors. The types of nodes are: *activity, send, receive, start, stop, andSplit, andJoin, orSplit* and *orJoin* [5].

Conditions in ICENI are provided by *orSplit* and *orJoin*. *OrSplit* can be considered as a conditional structure and all nodes after that are choices for execution. Only one of the child nodes can be selected for execution. Similar to *andJoin*, *orJoin* is the point where all conditional branches converge. It requires only one of its parents to be completed to transfer control to next node. If one node after *orSplit* and before *orJoin* (meaning that a node is in one of the conditional branches) is connected to a node coming before *orSplit*, then a loop structure occurs.

## 2.7. Kepler

Kepler is a popular workflow manager which aims to produce an open-source scientific workflow system that allows scientists to design scientific workflows and execute them efficiently using emerging Grid-based approaches to distributed computation [15]. Kepler is derived from Ptolemy [6]. In Ptolemy, many actors have conditional behavior. For example, generic filters may use conditions to filter some tokens at the input ports to forward them to their output ports. However, we are not interested in these kinds of actors. Instead, we are focusing on workflow control actors (Fig. 3a and 3b).

The *Comparator* is a logic actor, which takes two inputs and compares them according to $<, <=, >, >=, ==$. Output is a *boolean* result.

The *Repeat* structure repeats input tokens on the output by specified number of times.

The *BooleanSwitch* actor has one data input and one control input. It has two output ports: *TrueOutput* and *FalseOutput*. Based on the value on the control input, data in the input port is forwarded to the output port. Since Kepler does not have an *if* construct, *BooleanSwitch* actor can be considered as the closest construct to it. Figure 3a shows an example of *BooleanSwitch* in Kepler.

The *Select* actor has a control input, a data input, and one output port. However, data input is divided into channels. *Select* construct forwards the value to the output port from the channel that is specified by the value in the control input.

The *Switch* actor has one data input port, one control input port and many output ports. Control input port selects one output port to forward the input data to that output port. Figure 3b illustrates an example of *Switch* structure in Kepler.

The *BooleanMultiplexor* does the reverse operation of BooleanSwitch. It has two data input ports, one control input and one output port. According to the value on the control input, one of the input ports is selected to forward data to the output port.

The *Equals* actor has one multi-input port. In other words, input port has many channels. *Equals* actor checks the values on the input channels and compares them. If all values are equal, then it produces a true output, otherwise a false output.

The *IsPresent* actor has one input and one output port. It produces true output if data is presented in the input port on each firing [20].

## 2.8. Taverna

Taverna [21] is the workflow manager of myGrid [1] project which is a collection of comprehensive loosely-coupled suite of middleware components for supporting silico experiments in biology .

The conditional structures in Taverna aim to achieve the same functionality as *if* and *switch* structures in high-level languages such as C, Java and PERL. When a branch is selected for execution based on a condition, both control and data links, which are coming to that branch, are satisfied. This branch becomes the 'true' branch of the condition. Since the other branch is not selected, it becomes the 'false' branch and is never executed.

There is a sample conditional structure in Fig. 3c. Input to the condition is a *boolean* value. This value is sent to both tasks: $C$ and $C'$. This figure is designed considering the *fail_if_false* and *fail_if_true* processors of Taverna. They are conditional processors and refer to $C$ and $C'$ tasks in our figure. When a *boolean* value is produced by T1, this value is sent to both conditional tasks. One of the conditional tasks fails and causes to all tasks coming after it to fail. The other conditional task completes successfully and gives the control to the next task.

Figure 3d shows the implementation of a *switch* structure in Taverna. As can be seen from the figure, there are many tasks connected to task $C$ which refers to *fail_if_false* processor of Taverna. Each branch has one conditional task which is very similar to the Fig. 3c. However the difference between two figures is the value coming from T1 which is sent to each $S$ task referring to java beanshell script. Each script evaluates a *boolean* value and sends it to the next conditional task.

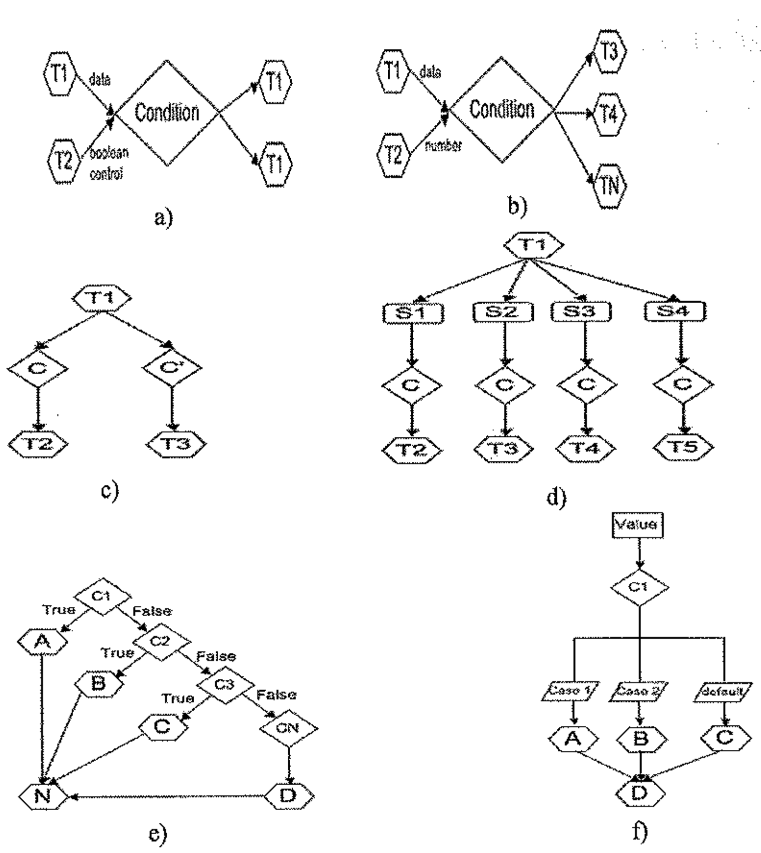


Fig. 3. Conditional structures in Kepler, Taverna, and Apache Ant a) *BooleanSwitch* structure in Kepler b) *switch* structure in Kepler c) *if* structure in Taverna d) *switch* structure in Taverna e) *if* structure in Apache Ant f) *switch* structure Apache Ant.

If the value coming to $C$ is true, all tasks in that branch will be executed. If the value is false, that branch will not be executed at all.

### 2.9. *Apache ant*

Apache Ant is a java-based tool for controlling build processes. Ant built files are written in XML and each one includes one project. Each project has at least one *target* and can have many *targets*. *Targets* are set of tasks and have five attributes: *name, depends, if, unless* and *description*. Each *target* must have a *name*. *Depends* attribute creates dependencies between *targets*. These dependencies determine the execution order of the *targets*. *Description* is an optional attribute that can be used to provide a one-line description.

In some cases, it is necessary to execute one *target* based on the value of a condition. If a *target* is needed to execute when a condition becomes true, *if* attribute is used, otherwise *unless* attribute is used. This is one way of using conditions while creating a workflow [4].

There is also a *condition* task in Ant. The *condition* task has *property* attribute, which is set when a condition becomes true. Ant has also some other conditional elements to be used in *condition* task. These elements can be used as nested elements in each *condition* task. *And, not, or, xor, available, equals, isset,* and *contains* are mostly used conditional elements in the *condition* task. These conditional elements provide users with the possibility of creating more useful conditions.

Besides these core tasks, some conditional and iterative tasks are developed by Ant-contrib [3] project. These new conditional and iteration elements would make it difficult for new users to understand and increase the maintenance cost if they were included in the core tasks group. However, the goal of Ant is to exclude complexity and functionality as much as possible [1].

Ant-contrib [2] is a project that contributes to Ant by developing conditional and iterative structures. These structures are:

*If* structure performs some tasks based on the value of a condition. Condition sets the value of the specified property to true if condition holds true. There are a number of conditional tasks that can be used in the *condition* task.

The task used in condition does not have an attribute. However, it may have up to three kinds of children elements: *elseif, then* and *else.* Figure 3e illustrates an example of how *if* structure is implemented in Apache Ant.

The *switch* structure is developed by Ant-contrib. As can be seen from Fig. 3f, it is very similar to *switch* structure in C and Java. Only difference is that it is written in XML format in Ant-contrib.

## 3. Case studies

In this section, we compare six of the studied workflow management systems in more detail using three case studies. The workflow management systems we compare in this section are: Kepler, Triana, Taverna, Apache Ant, Karajan, and UNICORE.

### *3.1. Case Study – I*

In this case study, we have two tasks in the workflow: Task A for staging some input data, and Task C for performing a computation using the data transferred in A. The goal of this case study is as follows: if task A fails for some reason, execute an alternate task (Task B) which will transfer the same file from a different source location and will feed this file into task C. Figure 4 shows the implementation of this workflow structure using six different workflow management systems mentioned above, for which we give the details next:

**Kepler.** Figure 4d shows the implementation of the described conditional structure in Kepler. In this particular example, we download a file from a URL by using *wget* command. For this purpose, *execute cmd remotely/locally* actor is selected. This actor executes a command in a specified system. It has two input ports. *Target* is the first input port that takes a *string constant*, specifying the target system where our command will be executed. *Local string* is used as input for this port since we want to execute our code in local machine. Second input is a *command* which takes a *string constant* that will be executed. This actor has four output ports. These are: *stdout, stderr, exitcode*, and *errors*. Since we need to know whether our command is successfully executed or not, we are only interested in *exitcode* port. *Exitcode* output is connected to *select* actor. *Select* actor chooses one of the commands based on the value coming from *exitcode* port. If *wget* command fails, alternative *wget* command will be forwarded to second *execute cmd remotely/locally* actor. Otherwise a meaningless *wget* command will be sent because first *execute cmd remotely/*locally actor must have already downloaded the file.

**Triana.** In Triana, we have written our own tool in Java for downloading a web page, which we call *my_stage_in*. It has one output port and produces '4' if web page is downloaded successfully and '1' if there is an error. Figure 4e shows how our implementation of *if* is done in Triana. As can be seen from the figure, two *my_stage_in* tools are used. These tools are connected by an *if* tool which is described in previous section. *if* tool's test value is set to '2'. When first *my_stage_in* tool fails, the control is sent to second *my_stage_in* tool since test value is greater than the failure code of *my_stage_in*. On the other hand, when first *my_stage_in* completes execution successfully, the control is sent to *merge* tool.

**Taverna.** In Taverna, when a processor fails, other processors, whose execution depends on the processor that is failed, can never execute. Therefore, we have performed small modifications in our experiment to implement a similar example. After modification, our goal in *if* case is to choose one alternative webpage based on a numeric return value of the previous task. For this purpose, we used two *Get_web_page_from_URL* processors to download web pages, one *Write_Text_File* tool to save the web page to the local system, and two conditional processors: *fail_if_false* and *fail_if_true*. In addition to these tools, we have implemented a Java beanshell in order to cast the numeric value to a *boolean* value. Figure 4f shows this *if* implementation in Taverna. As seen in the figure, the input is taken from the user and assumed to be the result of a processor. Beanshell script changes the numeric value to *boolean* and sends the output to both *fail_if_false* and *fail_if_true processors*. The failure of the condition is based on the *boolean* value and it will cause that complete branch to fail. The other branch corresponding to the successful condition will continue execution and the target web page will be downloaded as a result of the related *Get_web_page_from_URL* processor's execution.

**Apache Ant.** As mentioned in Section 2.9, *if* and *switch* structures are added to Apache Ant by Ant-Contrib project. However, we used *if* structure to implement both *if* and *switch* examples since the *switch* structure of Ant-Contrib task is not suitable for our scenario as *if* structure. *Switch* structure is appropriate for the situations where execution of a task depends on a

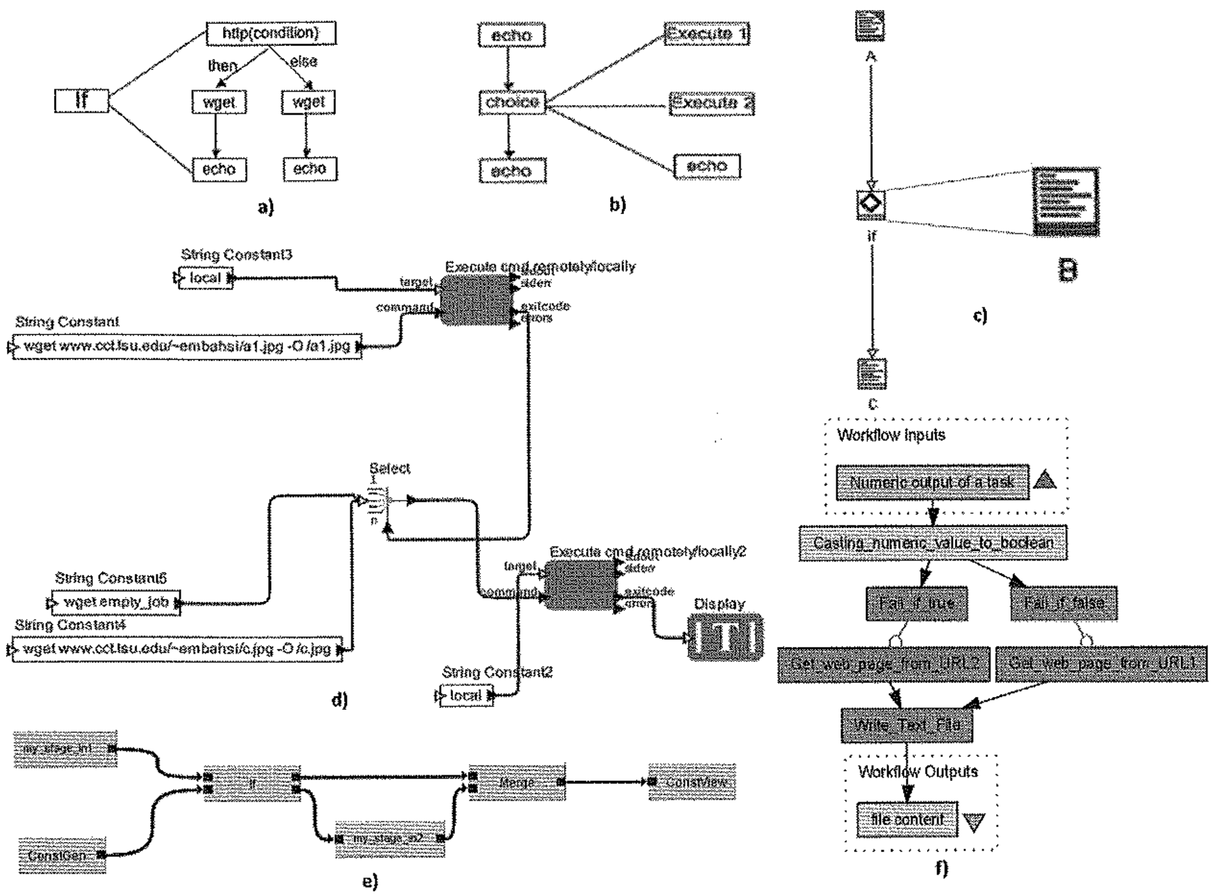


Fig. 4. Implementation of *if* Structures in: a) Apache Ant b) Karajan c) UNICORE d) Kepler e) Triana f) Taverna.

value. However, in our case study execution of a task depends on the failure of the previous task. Figure 4a shows the implementation of *if* example. As can be seen from the figure, *http* element is used as the condition to check whether the given URL exists or not. Based on this condition, one of the *then* and *else* elements is selected for execution. Both *then* and *else* element include a *get* element whose job is downloading a file from a URL.

**Karajan.** Figure 4b shows our *if* example in Karajan. As can be seen from the figure, *choice* element is used including two *execute* tasks and one *echo* task. *Execute* tasks are used for executing *wget* command by different URLs. If the first *execute* task fails, second task is executed. If both of the *execute* tasks fail, *echo* task creates an error message.

**UNICORE.** For all of implementation examples in UNICORE, we have used script tasks. In these tasks *wget* command is called. Figure 4c shows our *if* implementation in UNICORE. As can be seen, program executes task $A$, *if* (*IfThenElse*) task, and task $C$ sequentially. Task $A$ includes a *wget* command which tries the first URL. *If* task includes another script task called $B$. Task $B$ has also a *wget* command with an alternative URL. Based on test condition of *if*, *wget* command in task $B$ is executed if task $A$ fails. Task $C$ is a script that includes an *echo* command. It is executed after *if* task. Its execution does depend on neither success nor failure of task $A$.

### 3.2. Case Study – II

In this case study, we want to be able to choose among more than two alternative tasks available. In order to implement this, we need a structure more complex than a simple *if*, such as a *switch* structure.

**Kepler.** *Switch* implementation in Kepler (shown in Fig. 5d) is very similar to *if* implementation. In this case we have used three more *string constants* for the third *execute cmd remotely/locally*. However, execution of the third set of actors depends on results of the two command executions. First two *exitcode*

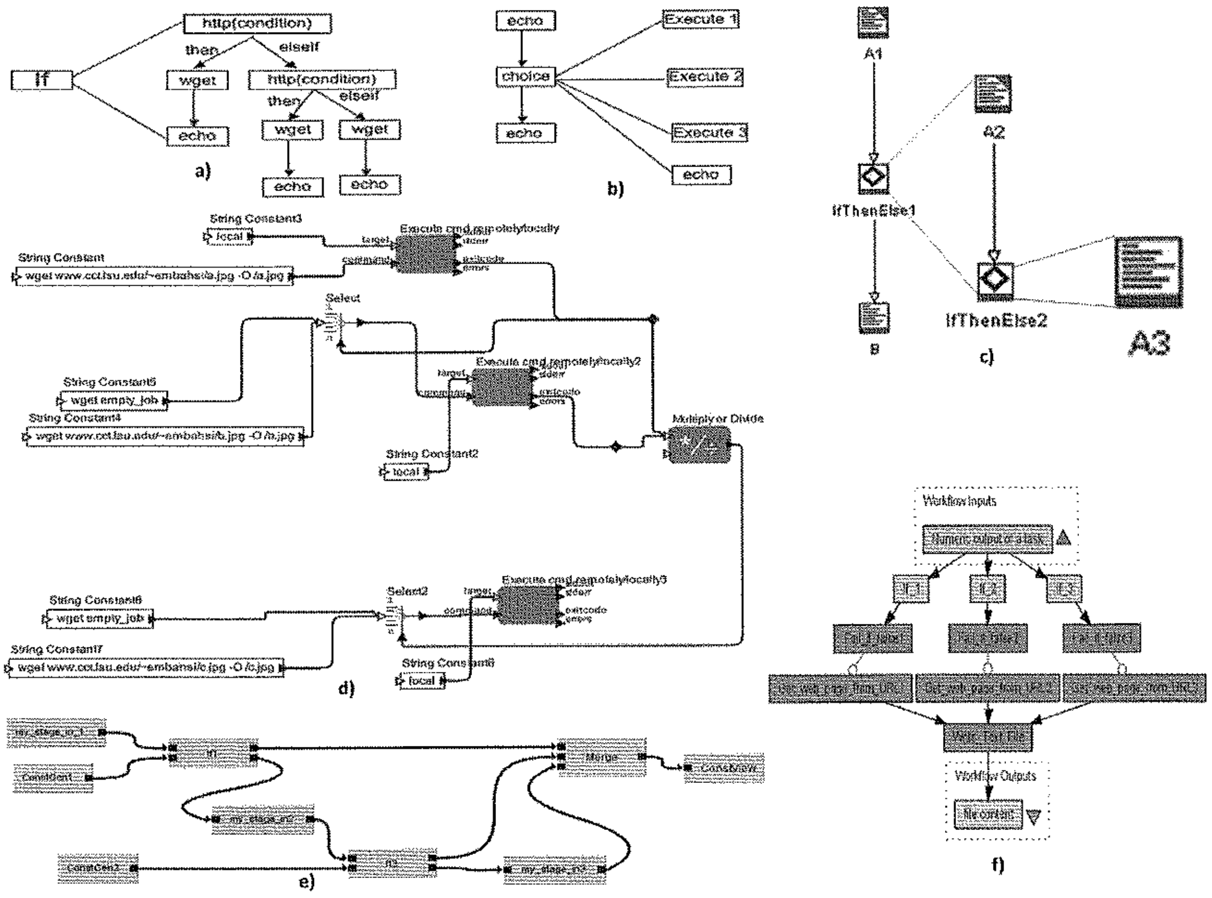


Fig. 5. Implementation of *switch* Structures in: a) Apache Ant b) Karajan c) UNICORE d) Kepler e) Triana f) Taverna.

outputs are sent to *Multiply or Divide* actor and the result is connected to second *select* actor. *Select* actor will choose the third meaningful *wget* command if none of the previous *wget* commands were successful. It will choose meaningless *wget* command if at least one of the previous *wget* commands was completed successfully.

*Execute cmd remotely/locally* is the most suitable actor for our purpose since it does not throw any exception when execution of the command fails. Instead, it sends an error message. However, many actors in Kepler throw exception when they fail. Therefore, the possibility of adding conditional behavior to Kepler depends on which tasks will be used for specific purposes.

**Tirana.** Figure 5e shows our implementation of *switch* in Triana. Similar to the implementation of the *if* example, we used *my_stage_in, if* and *merge* tools. In this case, one more *if* and *my_stage_in* tools are used since three different transfer tasks are involved. The second *my_stage_in* is connected to second *if* since the execution of the third *my_stage_in* tool should depend on the success of previous *my_stage_ins*. The model can be expanded by adding more *my_stage_in, if* tools and incrementing the number of input ports of *merge* tool.

**Taverna.** Figure 5f shows our implementation of *switch* by using three beanshell scripts, three *fail_if_false* and three *get_web_page_from_URL* processors. There are three branches and each beanshell script is used for each branch. The workflow runs very similar to *if* case. The major difference is in *switch* case where each beanshell script evaluates its own *boolean* value based on its implementation and sends to different branches. The *fail_if_false* processors, which retrieve true value from previous scripts let its branch execute. The branches can be incremented by adding sets of beanshell scripts, *fail_if_false*, and *Get_web_page_from_URL* processors.

**Apache Ant.** Figure 5a shows the implementation of *switch* structure using Apache Ant. In this example, there exists an additional *elseif* element. Inside this element, there is another *http* element for checking the existence of second web page. If condition is met

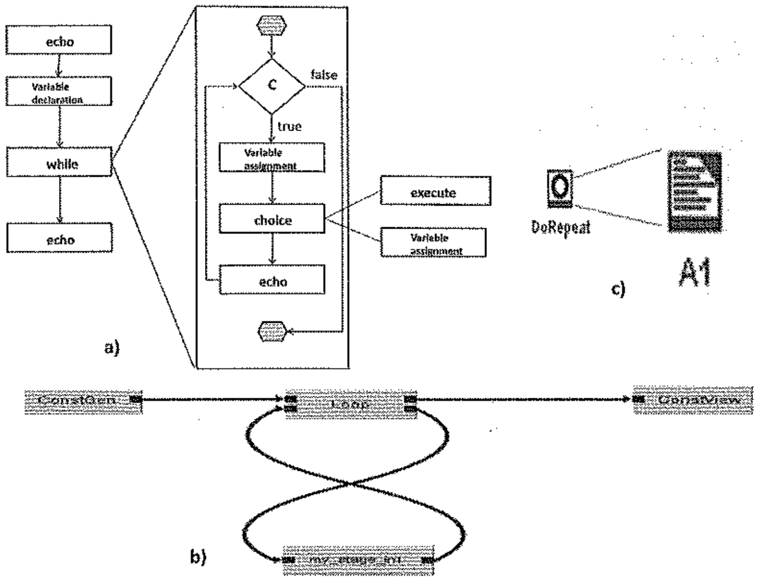


Fig. 6. Implementation of *while* Structures in: a) Karajan b) Triana c) UNICORE.

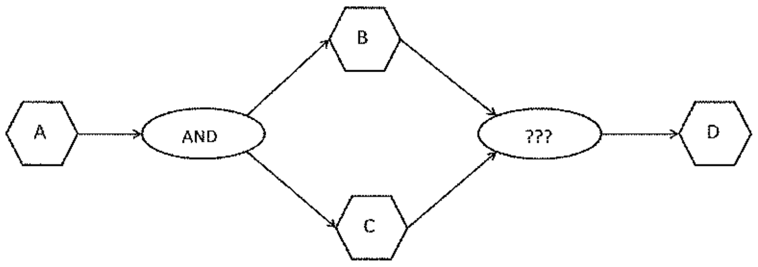


Fig. 7. How to Merge? [25].

the file is downloaded from the web page URL that is specified in *elseif*, otherwise web page URL in the *else* element is tried for downloading the file. To expand the branches of our *switch* example, more *elseif* elements can be added.

**Karajan.** In Figure 5b, *switch* example is implemented using *choice* element in Karajan. In this example there is one extra *execute* task. Therefore, one more *execute* task will be executed if the first two tasks fail. In order to expand the examples, the number of alternative *execute* tasks can be increased.

**UNICORE.** Although UNICORE does not have a special *switch* structure, we have used two *if* structures to imitate it. Figure 5c shows our implementation of *switch* in UNICORE. As can be seen, there are four script tasks called as A1, A2, A3 and B. A1, A2, and A3 tasks include *wget* commands with different URLs. If A1 fails, first *IfThenElse* task executes A2 which has an alternative URL for A1. *IfThenElse2* runs after completion of A2 and gives control to A3 if A2 fails to retrieve web page. A3 executes *wget* with its URL and delegates control to B. B task has an *echo* command and executes in any case.

### 3.3. Case Study – III

In this case study, we implement a loop structure which repeats a certain part of the workflow until it meets a certain condition.

**Triana.** Figure 6b illustrates our *while* implementation in Triana for downloading a webpage. As can be seen from the figure, *my_stage_in* tool's input and output ports are connected to output and input ports of *loop* tool, respectively. *Loop*'s exit condition is set based on the return value of *my_stage_in* tool, so that *my_stage_in* will try to download the webpage until it succeeds.

```
<project>
  <include file="cogkit.xml"/>
  <echo message="Program started"/>
  <choice>
    <task:execute executable="wget" arguments="http://www.cct.lsu.edu/~embahsi/zzz.jpg" provider="local" redirect="true"/>
    <task:execute executable="wget" arguments="http://www.cct.lsu.edu/~embahsi/das.jpg" provider="local" redirect="true"/>
    <echo message="{error}"/>
  </choice>
  <echo message="Program finished"/>
</project>
```

a)

```
<project name="Emir" default="init" basedir=".">
  <taskdef resource="net/sf/antcontrib/antcontrib.properties">
    <classpath>
      <pathelement location=".\ant-contrib\ant-contrib-1.0b3.jar"/>
    </classpath>
  </taskdef>
  <target name="init">
    <if>
      <http url="http://www.cct.lsu.edu/~embahsi/a.jpg" />
      <then>
        <get src="http://www.cct.lsu.edu/~embahsi/a.jpg" dest="./myfiles/a.jpg"/>
        <echo>File is being downloaded from first resource</echo>
      </then>
      <else>
        <get src="http://www.cct.lsu.edu/~embahsi/b.jpg" dest="./myfiles/b.jpg"/>
        <echo>First resource did not work so file is downloaded from alternative source</echo>
      </else>
    </if>
  </target>
</project>
```

b)

Fig. 8. Codes Generated in Implementation of Case Study–I a) using Karajan b) using Apache Ant.

**Karajan.** Figure 6a shows *while* example implemented in Karajan. In this figure, *while* element, which is already provided by Karajan libraries, is used. In order to control the execution of *while* loop, a nested *condition* element is used inside the *while* element. This condition checks the value of a variable for controlling execution of *while*. In addition, *while* element includes a *choice* element, which has an *execute* task to retrieve a web page. If it fails, next element in *choice*, which is a variable declaration, executes and sets the variable to '0' in order to specify that the *execute* element has failed. Otherwise variable declaration is skipped. This variable declaration will affect the condition's value in the next iteration, when the loop control comes back to condition again.

**UNICORE.** Figure 6c illustrates our *while* implementation by using *DoRepeat* task of UNICORE. As can be seen *DoRepeat* task contains task A1 which has *wget* command with a URL address. *DoRepeat* task's condition checks the success of A1 and repeats running it until A1 downloads the file from URL and finishes execution.

## 4. Discussion

Our study showed that the level of support for conditional structures in each of the workflow management systems is quite different. Systems like UNICORE and Karajan provide very rich and powerful conditional structures, which allow the user to build almost any conditional workflow possible. On the other hand, although the systems like Kepler and Apache Ant provide some level of conditional structure support, they are very limited in terms of functionality and do not allow to create some types of conditions (i.e. loops). Additionally, some conditional structures are developed to implement very specific use cases and do not allow control of flow for all types of tasks. For instance in the first two scenarios of Apache Ant we have had to use *http* core conditional task to check the existence of a URL. Although this worked in a "file downloading" example, it would not work for selecting between other types of tasks.

In some of the workflow management systems, failure of a task causes whole workflow to fail. This makes the implementation of a condition more difficult espe-

```
<tool>
 <toolname>emir_if</toolname>
 ...
 <tasks>
  <task>
   <toolname>ConstView</toolname>
   <package>Common.Const</package>
   ...
  </task>
  <task>
   <toolname>Merge</toolname>
   <package>Common.Stream</package>
   ...
  </task>
  <task>
   <toolname>my_stage_in2</toolname>
   <package>emir</package>
   <proxy type="Java">
    <param paramname="unitName">
     <value>emir.my_stage_in3</value>
    </param>
    <param paramname="unitPackage">
     <value>emir</value>
    </param>
   </proxy>
   ...
   <input>
    <type>java.lang.Object</type>
   </input>
   <output>
    <type>Const</type>
   </output>
   <parameters>
    <param name="file_url" type="userAccessible">
     <value>http://www.cct.lsu.edu/~embahsi/c.jpg</value>
    </param>
    <param name="outputType0" type="unknown">
     <value>triana.types.Const</value>
    </param>
    <param name="file_location" type="userAccessible">
     <value>C:\triana\myfile.jpg</value>
    </param>
  ...
   </parameters>
  </task>
  <task>
   <toolname>my_stage_in1</toolname>
   <package>emir</package>
   <proxy type="Java">
    <param paramname="unitName">
     <value>emir.my_stage_in3</value>
    </param>
    <param paramname="unitPackage">
     <value>emir</value>
    </param>
   </proxy>
   ...
   <input>
    <type>java.lang.Object</type>
   </input>
   <output>
    <type>Const</type>
   </output>
                                                          →*
```

```
*→
<parameters>
<param name="file_url" type="userAccessible">
     <value>http://www.cct.lsu.edu/~embahsi/a.jpg</value>
    </param>
    <param name="outputType0" type="unknown">
     <value>triana.types.Const</value>
    </param>
    <param name="file_location" type="userAccessible">
     <value>C:\triana\myfile.jpg</value>
    </param>
    ...
   </parameters>
  </task>
  <task>
   <toolname>ConstGen1</toolname>
   <package>Common.Const</package>
   ...
   <parameters>
    <param name="constant" type="userAccessible">
     <value>2.5</value>
    </param>
    ...
   </parameters>
  </task>
  <task>
   <toolname>If</toolname>
   <package>Common.Logic</package>
   ...
   <parameters>
   <param name="threshold" type="userAccessible">
     <value>2.000</value>
   </param>
   ...
   </parameters>
   ...
  </task>
  <connections>
   <connection type="NonRunnable">
    <source taskname="Merge" node="0" />
    <target taskname="ConstView" node="0" />
   </connection>
   <connection type="NonRunnable">
    <source taskname="If" node="0" />
    <target taskname="Merge" node="0" />
   </connection>
   <connection type="NonRunnable">
    <source taskname="my_stage_in2" node="0" />
    <target taskname="Merge" node="1" />
   </connection>
   <connection type="NonRunnable">
    <source taskname="If" node="1" />
    <target taskname="my_stage_in2" node="0" />
   </connection>
   <connection type="NonRunnable">
    <source taskname="my_stage_in1" node="0" />
    <target taskname="If" node="0" />
   </connection>
   <connection type="NonRunnable">
    <source taskname="ConstGen1" node="0" />
    <target taskname="If" node="1" />
   </connection>
  </connections>
 </tasks>
</tool>
```

Fig. 9. Code Generated in Implementation of Case Study–I using Triana.

cially in the scenarios where an alternative task execution is needed in the case of a task failure. In order to prevent the whole workflow to fail, tasks should pass an exit code to the workflow manager when they fail. For instance, Kepler has an actor called *execute cmd remotely/locally* that returns an exit code. The value of the exit code depends on the success of the execution. However not all of the actors in Kepler return exit code. Processes in Taverna cause the rest of the workflow to fail in the case of a failure. They do not return an exit code. On the other hand each process has retry, delay, and backoff behavior that increase the level of fault tolerance. In Triana the logical and control tasks provide control flow by passing data to the appropriate branch. In order to implement our case studies we have written our own task called *my_stage_in* in Java that produces an output based on the success of the task. *If* and *loop* tasks choose the appropriate branch for the rest of the execution of the workflow. As can be seen from Triana case studies, in some cases users may need to write their own tasks. However, most of the times only small modifications to the existing tasks can be enough. These modifications include adding a new output port and sending an output value. By this way users can add exit code to any task.

An important factor in choosing between different workflow management systems is user friendliness, or in other words the ease of implementation of the same structures using each system. In our case studies, we have spent least amount of time for implementations in UNICORE since it has a condition called *ReturnCode* test. By using *ReturnCode* test a new task can be executed as an alternative when the previous task fails. In addition, *ReturnCode* test can be used for comparing the return value of a task with a number. This property of *ReturnCode* can be beneficial for some situations where a task's execution depends on the previous task's return value.

Similarly, the amount of code that needs to be generated in implementing those structures can be important for some users. Our studies showed that least code generation is required by Karajan and Apache Ant, whereas most code generation is required by Triana and Taverna. Of course, most of these systems provide graphical user interfaces which takes the responsibility of code generation from the user. But still, this can be an important factor in choosing the most suitable workflow management system for some users. We have provided the codes generated by some of the workflow management tools as examples in Figs 8 and 9. Figure 8a shows the code generated for the implementation of case study – 1 (if construct) using Karajan. Figure 8b shows the same for Apache Ant, and Fig. 9 for Triana.

The conditional structures can be observed in two parts: **exclusive choice** and **simple merge**. Exclusive choice is the point in the workflow process where, based on a decision or workflow control data, one of several branches is chosen. Simple merge is the point in the workflow process where two or more alternative branches come together without synchronization [25].

**Multi-choice** is the point in the workflow process where number of branches is chosen based on a decision or workflow control data. Implementation of multi-choice is somehow easier than the *merge* construct. In *merge* construct, it is very difficult to determine where to synchronize and when to merge (Fig. 7).

Improper usage of *split* and *join* pairs may cause deadlock. When a process splits into multiple sub-processes through an *OR-Split*, and the sub-processes subsequently join via an *AND-Join* or synchronize using a *synchronizer*, then any activities and rules following the *join* or the *synchronizer* could never get activated – i.e., resulting in deadlock, because the activities following the *split* would not terminate together [25].

Some tasks may fail at some point in the execution of the workflow. This failure may cause the whole workflow to fail. In order to prevent this, workflow structure may execute alternative tasks when a task fails. Each alternative task must have a priority value and this priority value will show the execution order of the alternative tasks in case of a failure.

In conditional parts of the workflows, it is not possible to decide at compile time which branch is going to be executed next. This decision is made at the run-time. Therefore, workflow management systems should also provide a dynamic data flow like control flow.

## 5. Conclusion

In this paper, we studied the most widely used workflow management systems and their support for conditional structures such as *if, switch, and while*. We have observed that some of the most popular systems do not support conditional structures at all. We have also observed that, although some workflow management systems do not support conditional structures, users may find alternative ways to imitate the conditional structures using other structures provided by these workflow management systems.

We have compared implementations of common conditional structures using each of these workflow management systems via case studies, and discussed capabilities of each system. Our study shows that the same structure can be implemented in completely different ways by different workflow management systems. And, although some systems do not support all of the conditional structures we have studied, more basic structures such as *if* can be used to implement other complex conditional structures in many cases.

## Acknowledgements

This work is partially supported by NSF grant CNS-0619843 (PetaShare) and Louisiana BoR-RCS grant LEQSF (2006-09)-RD-A-06 (CoastDap). We also would like to thank Jeremy Cohen. Tom Oinn, Stian Soiland. Mattew Shields. Gaurang Metha, Kassian Plankensteiner, Ken Kennedy, Jia Yu, Mihael Hategan, Gregor von Laszewski for their feedback and contribution.

## References

[1] About myGrid accessed December 2006 [Online]. Available: http://www.mygrid.org.uk/?&MMN_position=1:1.

[2] Ant-contrib Tasks accessed December 2006 [Online]. Available: http://ant-contrib.sourceforge.net/.

[3] Ant-contrib Tasks accessed December 2006 [Online]. Available: http://ant-contrib.sourceforge.net/tasks/tasks/index.html.

[4] Apache Ant accessed December 2006 [Online]. Available: http://ant.apache.org/.

[5] A. Mayer, S. McGough, N. Furmento, W. Lee, S. Newhouse and J. Darlington, ICENI Dataflow and Workflow: Composition and Scheduling in Space and Time, In UK e-Science All Hands Meeting, Nottingham, UK, IOP Publishing Ltd, Bristol, UK, September 2003, 627–634.

[6] B. Ludascher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E.A. Lee, J. Tao and Y. Zhao, Scientific Workflow Management and KEPLER System, Concurrency and Computation: Practice & Experience, Special Issue on Scientific Workflows, to appear, 2005.

[7] Cog Kit GridAnt Project Page accessed December 2006 [Online]. Available: http://www.globus.org/cog/projects/gridant/.

[8] D. Erwin et al., UNICORE Plus Final Report – Uniform Interface to Computing Resources, The UNICORE Forum e.V., ISBN 3-00-011592-7, 2003. Online:http://www.unicore.org/documents/UNICOREPlus-Final-Report.pdf.

[9] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.H. Su, K. Vahi and M. Livny, Pegasus: Mapping Scientific Workflow onto the Grid, *Across Grids Conference 2004*, Nicosia, Cyprus, 2004.

[10] F. Leymann, Web Services Flow Language (WSFL 1.0), IBM, May 2001.

[11] Grid Workflow: ICENI accessed December 2006 [Online]. Available: http://www.gridworkflow.org/snips/gridworkflow/space/ICENI.

[12] G.v. Laszewski and M. Hategan, Java CoG Kit Karajan/GridAnt Workflow Guide, Technical Report, Argonne National Laboratory, Argonne, IL, USA, 2005.

[13] I. Taylor, S. Majithia, M. Shields and I. Wang, Triana WorkFlow Specification, GridLab Specification available at: www.gridlab.org/WorkPackages/wp-3/D3.3.pdf.

[14] J. Yu and R. Buyya, A Taxonomy of Workflow Management Systems for Grid Computing. Technical Report GRIDS-TR-2005-1, Grid Computing and Distributed Systems Laboratory, University of Melbourne, 2005. http://www.gridbus.org/reports/GridWorkflowTaxonomy.pdf.

[15] Kepler Project accessed December 2006 [Online]. Available: http://www.kepler-project.org/.

[16] K. Cooper, A. Dasgupata, K. Kennedy, C. Koelbel, A. Mandal, G. Marin, M. Mazina, J. Mellor-Crummey, F. Berman, H. Casanova, A. Chien, H. Dail, X. Liu, A. Olugbile, O. Sievert, H. Xia, L. Johnsson, B. Liu, M. Patel, D. Reed, W. Deng, C. Mendes, Z. Shi, A. YarKhan and J. Dongarra, New Grid Scheduling and Rescheduling Methods in the GrADS Project, *NSF Next Generation Software Workshop*, International Parallel and Distributed Processing Symposium, Santa Fe, IEEE CS Press, Los Alamitos, CA, USA, April 2004.

[17] LESC – London e-Science Centre ICENI accessed December 2006 [Online]. Available: http://www.lesc.imperial.ac.uk/iceni/.

[18] P. Couvares, T. Kosar, A. Roy, J. Weber and K. Wenger, Workflow Management in Condor, In Workflows for e-Science, Editors: I.Taylor, E.Deelman, D.Gannon, M.Shields, Springer Press, January 2007 (ISBN: 1-84628-519-4).

[19] R. Buyya and S. Venugopal. The Gridbus Toolkit for Service Oriented Grid and Utility Computing: An Overview and Status Report, in *1st IEEE International Workshop on Grid Economics and Business Models*, GECON 2004, Seoul, Korea, IEEE CS Press, Los Alamitos, CA, USA, April 23, 2004, 19–36.

[20] S.S. Bhattacharyya, C. Brooks, E. Cheong, J. Davis, II, M. Goel, B. Kienhuis, E.A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, B. Vogel, W. Williams, Y. Xiong, Y. Zhao and H. Zheng, Ptolemy II Heterogeneous Concurrent Modeling and Design In Java-Volume 1: Introduction to Ptolemy II. Memorandum UCB/ERL M05/21 EECS UC Berkeley, CA 94720, July 15, 2005.

[21] Taverna 1.5.2 Manual" accessed August 2007 [Online]. Available: http://www.mygrid.org.uk/usermanual1.5/index.html.

[22] T. Fahringer, J. Qin and S. Hainzer, Specification of Grid Workflow Applications with AGWL: An Abstract Grid Workflow Language. IEEE International Symposium on Cluster Computing and the Grid 2005 (CCGrid 2005), Cardiff , UK, May 9–12 2005. IEEE Computer Society Press.

[23] T. Fahringer, R. Prodan, R. Duan, F. Nerieri, Podlipnig, J. Qin, M. Siddiqui, H. Truong, A. Villazon and M. Wieczorek, ASKALON: A Grid Application Development and Computing Environment, 6th International Workshop on Grid Computing , Seattle, USA, IEEE Computer Society Press, November 2005.

[24] T. Oinn, M. Greenwood, M. Addis, M.N. Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin, P. Li, P. Lord, M.R. Pocock, M. Senger, R. Stevens, A. Wipat and C. Wroe, Taverna: Lessons in Creating a Workflow Environment for the Life Sciences, Concurrency and Computation: Practice & Experience, Volume 18, Issue 10 (August 2006) Workflow in Grid Systems Pages: 1067 – 1100, 2006, ISSN:1532-0626, John Wiley and Sons Ltd., Chichester, UK.

[25] W.M.P. v. d. Aalst, A.H.M. t. Hofstede, B. Kiepuszewski and A.P. Barros, Workflow Patterns, Technical Report FIT-TR-2002-02, Queensland University of Technology, Brisbane, Australia, 2002.

[26] YAWL: Yet Another Workflow Language accessed August 2007 [Online]. Available: http://www.yawl-system.com/.