# Chapter 4
# Workflow Modularization

## 4.1 Introduction

As more scientific workflows are built, workflow modularization mechanisms become essential for workflow languages or systems. For example, a domain scientist in one research group wants to run a workflow constructed by another group and to obtain results useful for his own research, but without knowing how to construct a scientific workflow application. In another case, domain scientists want to invoke a workflow multiple times within their own workflows (e.g., via a `for` loop). However, who creates the invoked workflow is not important, the workflow may have been composed by the same scientist or some other scientists, and it may have been defined inside or outside the workflow being composed. In addition, manually keeping track of the relationships among different versions of a workflow is a challenging task. Sharing workflows via email or file copying is also error prone. Therefore, workflow modularizations are important and must be supported by modern scientific workflow systems.

Following the specification of AWDL presented in the previous chapter, this chapter presents the workflow modularization mechanism of AWDL. It includes the definition and invocation of sub-workflows, as well as workflow libraries. An algorithm for detecting infinite recursive sub-workflow invocations is also presented. With the workflow modularization mechanism, domain scientists are no longer required to build every workflow from scratch. The workflow development time is significantly reduced and the quality of workflows is also improved by reusing established and validated workflows.

## 4.2 Sub-workflows

The informal XML grammar of `subWorkflow` in AWDL is illustrated in Fig. 4.1. The `subWorkflow` is similar to a procedure in other high-level programming languages. It is used to modularize, encapsulate, and reuse a code region.

J. Qin and T. Fahringer, *Scientific Workflows*, DOI 10.1007/978-3-642-30715-7_4,

```
<subWorkflow name="name">
  <dataIns>
    <dataIn name="name" type="type" />*
  </dataIns>
  <subWorkflowBody>
    <activity .../>+
  </subWorkflowBody>
  <dataOuts>
    <dataOut name="name" type="type" source="source"/>*
  </dataOuts>
</subWorkflow>
```

**Fig. 4.1** `subWorkflow`

Sub-workflows are organized in AWDL workflows. In an AWDL workflow, the name of each `subWorkflow` is unique.

A sub-workflow is invoked in the enclosing workflow which specifies the data to be processed in the sub-workflow through the data flow specification of the workflow. Therefore, there is no `source` attribute for the input data ports of the `subWorkflow`. This is one difference between a sub-workflow and a workflow. Inside a sub-workflow, only data flow among the sub-workflow and its inner activities is allowed. Such a data flow mechanism enables safe invocations of sub-workflows anywhere in the enclosing workflow, as well as publishing sub-workflows in a workflow library (see Sect. 4.3) as stand-alone workflows for further reuse. This is another difference between a sub-workflow and a workflow: a sub-workflow in a workflow is not visible outside of that workflow. Therefore, it cannot be invoked from another workflow unless it is published in a workflow library. An example of `subWorkflow` is illustrated in Sect. 4.5.

## 4.3 Workflow Library

In order to facilitate workflow sharing and reuse, as well as workflow version tracking, we put forward the concept of *workflow library*. By analogy to *unix libraries*, a workflow library is a set of established workflows and the corresponding management tool. Users can publish workflows in a workflow library, or mark workflows in a workflow library as deprecated. The workflows in a workflow library can be shared among research groups and can be invoked in other workflows. The functionalities provided by a workflow library are as follows:

- To publish workflows in the workflow library.
- To associate workflows with metadata, especially, workflow versions can be generated automatically based on the latest version in the workflow library.
- To search workflows by their metadata.

- To show workflow graphs, including graphs of its sub-workflows.
- To run workflows with a single click.
- To mark a workflow as deprecated for discouraging their use because, for example, a better alternative exists.

We have developed a workflow library called ASKALON Workflow Hosting Environment (AWHE) which will be demonstrated in Sect. 5.5.

The use of workflow libraries enables the categorization of ASKALON users into three classes. *Workflow Executors* execute existing, validated workflows by means of workflow libraries. *Workflow Developers* compose workflows by dragging and dropping activities, sub-workflows, and workflows into new workflows. Workflow developers can publish their workflows with the default input data (via the `source` attributes or the `value` elements of `dataIn` ports) and the default locations to save output data (via the `saveto` attributes of `dataOut` ports, see Sect. 3.2.1). Workflow developers can also leave the information empty for (potential) users to execute the published workflows with their own input data and their own locations for output data. *Application Providers* deploy binary executables, Web services, etc. into distributed systems and associate them with activity types via an activity registry service such as GLARE.

Users can run workflows directly from a workflow library without modifications, or with specifications of their own input data and locations for their output data. However, for the former to happen, users must have the necessary permissions to read the input data specified via the `source` attributes as well as to write the output data to the locations specified via the `saveto` attributes. Note that changes of back-end resources, e.g., the location of binary executables, are transparent to users because AWDL workflows are abstract, i.e., no concrete information such as the location of binary executables is encoded in AWDL workflows.

## 4.4 Invocation of (Sub-)workflows

As we described in Sect. 3.2.1, each atomic activity in AWDL has a `type` attribute which refers to an activity type. Figure 4.2 illustrates such an atomic activity referring to Activity Type (AT) *actt://mathematics/Matrix-Multiplication*, i.e., *MatrixMultiplication* in the *mathematics* domain. The execution of this activity means the execution of the specific AD mapped from the AT. Since the AT is abstract, i.e., implementation independent, the atomic activity can be reused in other workflows. This is called AT invocation. The invocation of (sub-)workflows is done similarly in AWDL. If an atomic activity in a workflow has a type referring to a sub-workflow defined in the workflow or a workflow published in a workflow library, executing this activity invokes the corresponding sub-workflow or workflow. Figure 4.3 illustrates an example of sub-workflow invocation: the workflow *w* defines a sub-workflow *subW1* (Line 2) which is invoked at Line 5 by the activity *a*. Figure 4.4 illustrates a workflow invocation: the workflow is referred by its domain,

```
<activity name="a"
      type="actt://mathematics/MatrixMultiplication">
  <dataIns> ... </dataIns>
  <dataOuts> ... </dataOuts>
</activity>
```

**Fig. 4.2** Activity type invocation

```
<workflow name="w">
  <subWorkflow name="subW1" .../>
  <dataIns>  ... </dataIns>
  <workflowBody>
    <activity name="a" type="subw://subW1">
      <dataIns>  ... </dataIns>
      <dataOuts>  ... </dataOuts>
    </activity>
  </workflowBody>
  <dataOuts>  ... </dataOuts>
<workflow>
```

**Fig. 4.3** Sub-workflow invocation

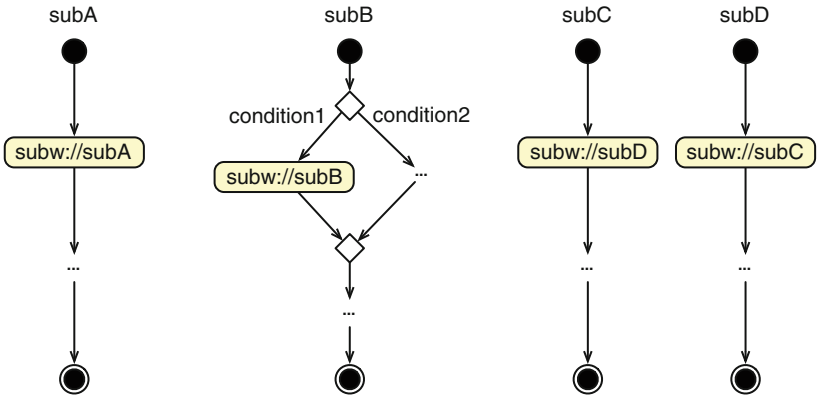
```
<activity name="a"
      type="wlib://domain/workflow/version">
  <dataIns> ... </dataIns>
  <dataOuts> ... </dataOuts>
</activity>
```

**Fig. 4.4** Workflow invocation

name, and version in a workflow library. In summary, the value of the `type` attribute of an atomic activity can be as follows:

- *actt://domain/activitytype*, referring to an activity type registered in an activity registry service.
- *subw://subworkflow*, referring to a sub-workflow that is specified as part of the workflow containing the atomic activity.
- *wlib://domain/workflow/version*, referring to a workflow published in a workflow library.

When an atomic activity refers to an AT, a sub-workflow, or a workflow, the types and the numbers of the input and output data ports are determined by the AT, the sub-workflow, or the workflow. As we will see in Sects. 5.4 and 5.5, the input and output data ports of the atomic activity are added automatically when the `type` attribute is determined.




(a) subA invoking subA (b) subB invoking subB (c) subC and subD invoking each other

**Fig. 4.5** Examples of incorrect recursive sub-workflow invocations

In order to support alternative execution, the value of the `type` attribute of an atomic activity can also refer to multiple ATs, multiple sub-workflows, multiple workflows, or any combination of them. Obviously, this requires that the referenced ATs, sub-workflows, and workflows must have the same input and output data structure.

### *4.4.1 Detection of Infinite Sub-workflow Invocation*

As mentioned, sub-workflows can be invoked when executing an atomic activity by referencing the sub-workflows in the `type` attribute of the atomic activity. This enables recursive sub-workflow invocations in the case where the atomic activity is defined in the sub-workflow. As in high-level programming languages (e.g., Java or C++), improperly designed recursive sub-workflow invocations can lead to infinite recursion. While some infinite sub-workflow invocations can be detected at composition time by analyzing the model of a workflow without actually running it, others, for example, depending on specific data being processed, can only be found at runtime. We deal with the former, i.e., the infinite sub-workflow invocations detectable at composition time, in this section. Figure 4.5 illustrates three examples of incorrect recursive sub-workflow invocations. In Fig. 4.5a, *subA* will cause infinite recursion because the first activity in *subA* invokes the sub-workflow itself. This is also the case with *subB* (Fig. 4.5b) if the branch at the left-hand side is executed at runtime. Finally, Fig. 4.5c shows a case of incorrect recursive sub-workflow invocations involving different sub-workflows invoking each other.

To address this problem, incorrect recursive sub-workflow invocations are detected by traversing workflow models while workflows are being composed and

**Algorithm 4.1:** Find incorrect recursive sub-workflow invocation

```
1  Method 1: detectIncorrectRecursiveInvocation()
   Input  : workflow w to detect incorrect recursive sub-workflow invocation
   Output : ordered list p, indicating sub-workflow invocation path, with the last element
            being the sub-workflow that is incorrectly recursively invoked, p = ∅ means
            no incorrect recursive sub-workflow invocation found.
2  Initialize the sub-workflow invocation path p = ∅
3  forall sub-workflow sub invoked in the workflow w do
4  |  detectRecursiveSubWorkflowInvocation(p, sub)
5  |  if p ≠ ∅ ∧ ¬ isCorrectRecursiveInvocation(p) then
6  |  |  return p
7  |  end
8  end
9  return ∅

10 Method 2: detectRecursiveSubWorkflowInvocation()
   Input  : sub-workflow invocation path p;   sub-workflow sub to be invoked next
   Output : true, if a recursive sub-workflow invocation is found; false, otherwise
11 if contains(p, sub) then
12 |  append(p, sub)
13 |  return true
14 end
15 append(p, sub)
16 forall activity a in the sub-workflow sub such that a has a type referring to a sub-workflow s
   do
17 |  isRecursion = detectRecursiveSubWorkflowInvocation(p, s)  // recursively call Method
   |  2
18 |  if isRecursion then
19 |  |  return true
20 |  end
21 end
22 removeLastElement(p)
23 return false

24 Method 3: isCorrectRecursiveInvocation()
   Input  : sub-workflow invocation path p with the last element being the sub-workflow
            being recursively invoked
   Output : true if correct, false otherwise
25 if there are activities between two invocations of the last sub-workflow then
26 |  return true
27 else
28 |  return false
29 end
```

before they are submitted for execution on distributed systems. Our approach for detecting incorrect sub-workflow invocations is described by Algorithm 4.1. *Method 1* is the entry point of the algorithm and for each sub-workflow invoked from the *main* workflow, it detects recursive sub-workflow invocations (*Method 2*) and, when found, it evaluates whether the recursive sub-workflow invocation is correct (*Method 3*). Specifically, *Method 2* detects recursive invocations by recursively

checking all possible sub-workflow invocation paths. The return *boolean* value of *Method 2* is used by itself to check whether it should continue expanding the sub-workflow invocation path $p$. The detected recursive sub-workflow invocation path $p$ is then evaluated in *Method 1* by calling *Method 3* ($p$ is shared by the three methods). *Method 3* evaluates the correctness of a recursive sub-workflow invocation path by checking whether there are data processing steps (i.e., activities) between two invocations of the same sub-workflow. If there are no activities in between, the sub-workflow invocation path is considered to be incorrect because it will cause infinite recursion. Otherwise, if there are some activities executed between two invocations of the same sub-workflow, it is considered to be a "correct" (from the workflow composition point of view) recursive sub-workflow invocation. Note that, in the latter case, we assume that in the second invocation of the sub-workflow, the input data of the sub-workflow comes from the output data of these activities and the output data of these activities are different from their input data. This is reasonable because it does not make sense for an activity to have output data identical to its input data.

### *4.4.2 Workflow Invocation*

As presented in Sect. 4.3, to facilitate workflow execution, *Workflow Developers* may publish their workflows into a workflow library with predetermined or commonly used input data (via the `source` attributes or the `value` elements of `dataIn` ports) and locations of output data (via the `saveto` attributes of `dataOut` ports). However, workflow invocation frequently requires that the invoked workflow process the data in the invoking workflow. To address this issue, the `source` attributes or the `value` elements of `dataIn` ports, and the `saveto` attributes of `dataOut` ports of the invoked workflow are simply ignored by the workflow engine. Instead, the data in the invoking workflow can be passed to the invoked workflow through the data flow between the two workflows.

For workflow invocation, we support two kinds of AWDL code generation mechanisms: (a) embedding the AWDL code of the invoked workflows as sub-workflows into the invoking workflow so that the workflow engine can execute the invoking workflow directly without retrieving the AWDL code of sub-workflows from workflow libraries. With this approach, we convert the workflow invocations into sub-workflow invocations; and (b) using the references of the invoked workflows in the invoking workflow without embedding the AWDL code of the invoked workflows. The workflow runtime system can obtain the code of the invoked workflows from workflow libraries when needed. This is the so-called late binding mechanism. This is also the distinction between sub-workflow invocation and workflow invocation as the AWDL code for sub-workflows is always embedded while the code for external workflows is not. Examples of the two code generation mechanisms are demonstrated in the next section.

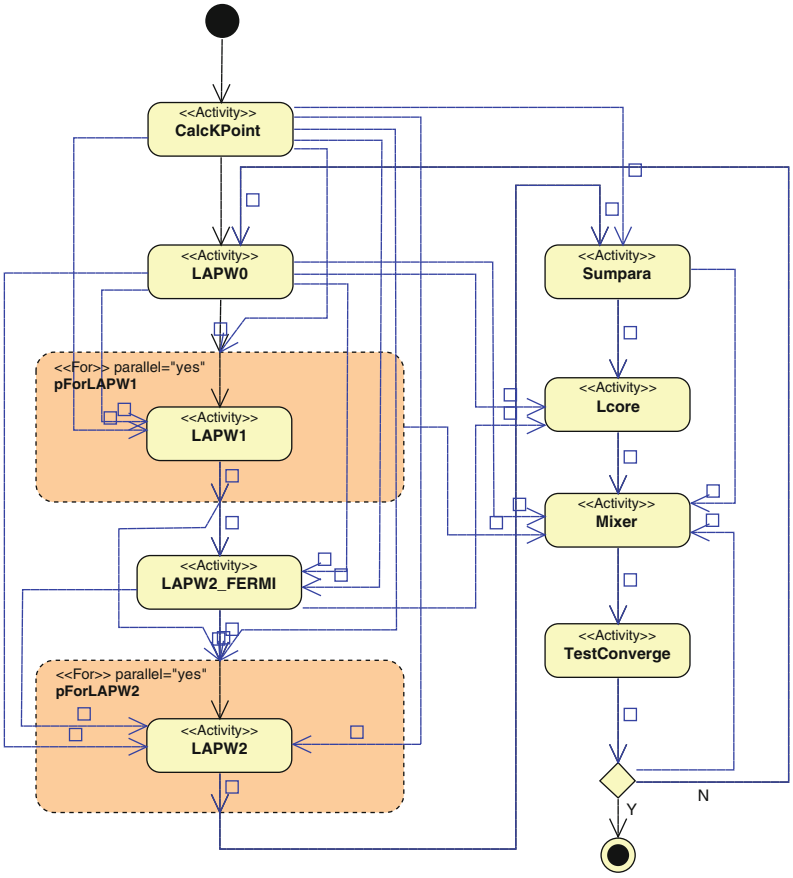
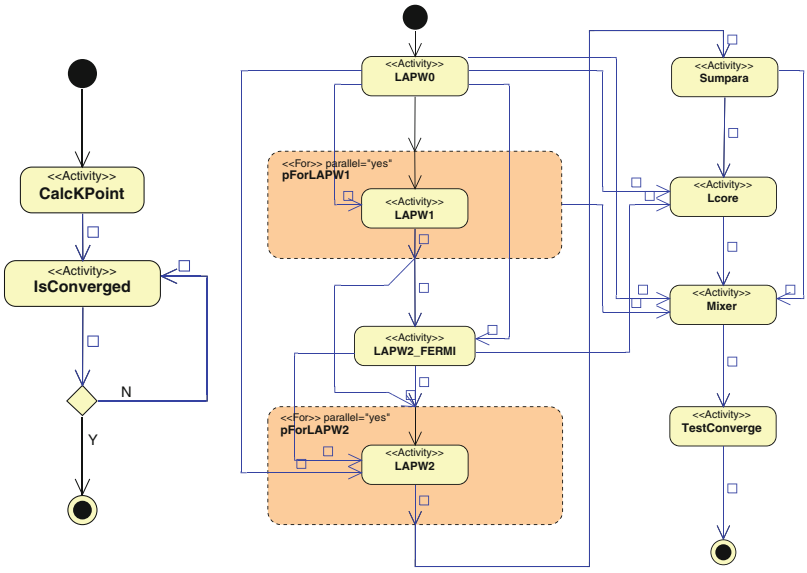



**Fig. 4.6** The WIEN2k workflow

## 4.5 Case Study: WIEN2k

The UML Activity Diagram representation of the workflow WIEN2k [25] is illustrated in Fig. 4.6, where arrows without small squares are control flow edges and arrows with small squares are data flow edges. Control flow and data flow edges may overlap. For the reason of simplicity, multiple data flow edges between any pair of activities are represented with a single data flow edge. The workflow consists of a `while` loop (represented by a DecisionNode and a control flow edge connecting backwards to a predecessor, i.e., the activity *LAPW0*. See Chap. 5 for the details on UML-based scientific workflow modeling). At the end of each loop iteration an output value is evaluated against a threshold to see whether the computation is converged. The workflow can be reorganized by extracting the body of the `while`




(a) the *main* workflow (b) the sub-workflow *IsConverged*

**Fig. 4.7** The WIEN2k workflow using sub-workflows

loop as a sub-workflow *IsConverged* (Fig. 4.7b) and then invoking the sub-workflow from the `while` loop in the *main* workflow (Fig. 4.7a).

The corresponding AWDL representation of the WIEN2k workflow using sub-workflows is illustrated in Fig. 4.8. The sub-workflow *IsConverged* is specified at Lines 2–21. The *main* workflow is specified at Lines 22–53 which invokes the sub-workflow at Lines 36–45. The data in the *main* workflow are passed to the sub-workflow through the `dataIn` ports (e.g., the `dataIn` port *in0File* at Line 38) of the activity *IsConverged*. The output data of the sub-workflow are passed back to the *main* workflow through the `dataOut` ports (e.g., the `dataOut` port *val* at Line 42) of the activity *IsConverged*.

Note that if the sub-workflow is published as a workflow *wlib://MaterialSicences /IsConverged/1.0* in a workflow library, the AWDL representation shown in Fig. 4.8 can be considered the case where the AWDL code of the invoked workflow *wlib://MaterialSicences/IsConverged/1.0* is embedded in the invoking workflow (i.e., the *main* workflow). If we remove Lines 2–21 and then change the `type` of the activity *IsConverged* to *wlib://MaterialSicences/IsConverged/1.0*, Fig. 4.8 would show the case of late binding code generation mechanism. In this case, the workflow runtime system can obtain the AWDL code of the workflow *wlib://MaterialSicences/IsConverged/1.0* from the workflow library when needed.

As a consequence of using sub-workflows, the workflow model is simplified in terms of the number of control and data flow connections. The modularization of this workflow is also improved: the extracted sub-workflow can be reused (i.e., invoked)

```
<workflow domain="MaterialScience" name="WIEN2k" ... />
  <subWorkflow name="IsConverged">
    <dataIns>
      <dataIn name="in0File" type="awdl:file" />
      <!-- other dataIns ports -->
    </dataIns>
    <subWorkflowBody>
      <activity name="LAPW0" ... />
      <for name="pForLAPW1" parallel="yes" ... />
      <activity name="LAPW2_FERMI" ... />
      <for name="pForLAPW2" parallel="yes" ... />
      <activity name="Sumpara" ... />
      <activity name="Lcore" ... />
      <activity name="Mixer" ... />
      <activity name="TestConverge" ... />
    </subWorkflowBody>
    <dataOuts>
      <dataOut name="val" type="xsd:boolean" source="TestConverge/outVal"/>
      <!-- other dataOut ports -->
    </dataOuts>
  </subWorkflow>
  <dataIns>
    <dataIn name="val"... />
    <dataIn name="in0File" ... />
    <!-- other dataIn ports -->
  </dataIns>
  <workflowBody>
    <activity name="CalcKPoint" ... />
    <doWhile name="doWhileConv">
      <dataIns> ... </dataIns>
      <dataLoops>
        <dataLoop name="val" type="xsd:boolean" initSource="WIEN2k/val"
             loopSource="IsConverged/val" />
        <!-- other dataLoop ports -->
      </dataLoops>
      <loopBody>
        <activity name="IsConverged" type="subw://IsConverged" >
          <dataIns>
            <dataIn name="in0File" type="awdl:file" source="Wien2K/in0File"/>
            <!-- other dataIns ports -->
          </dataIns>
          <dataOuts>
            <dataOut name="val" type="xsd:boolean"/>
            <!-- other dataOut ports -->
          </dataOuts>
        </activity>
      </loopBody>
      <condition combinedWith="and">
        <acondition data1="doWhileConv/val" data2="true" operator="=="/>
      </condition>
      <dataOuts ... />
    </doWhile>
  </workflowBody>
  <dataOuts ... />
</workflow>
```

**Fig. 4.8** The AWDL representation of the WIEN2k workflow using sub-workflows

anywhere in the *main* workflow. If the extracted sub-workflow has been published into a workflow library, it can be reused in other workflows as well. In this case, only the *main* workflow (Fig. 4.7a) needs to be composed, which significantly reduces the user effort needed for composing scientific workflows. Using sub-workflows also facilitates workflow maintenance and evolution as any change to the sub-workflow will be automatically reflected in any activity, sub-workflow, or workflow that has a reference to the sub-workflow.

## 4.6 Summary

As more and more scientific workflows are being designed and built, workflow sharing and reuse becomes important and necessary for enhancing scientists' productivity. In this chapter, we presented the AWDL modularization mechanism for workflow sharing and reuse in ASKALON. A sophisticated algorithm for the detection of incorrect sub-workflow invocation at composition time is also presented. The mechanism provides a simple and consistent way for reusing workflows, sub-workflows, and workflow activities. We demonstrated the effectiveness of our approach by applying it to a real-world scientific workflow application.

In the next chapter, we will present our approach for UML-based scientific workflow modeling which supports the visual composition of scientific workflows.

http://www.springer.com/978-3-642-30714-0

Scientific Workflows
Programming, Optimization, and Synthesis with ASKALON and AWDL
Qin, J.; Fahringer, Th.
2012, XXII, 222 p., Hardcover
ISBN: 978-3-642-30714-0