COMAD User Manual

Version 2.0
April 19, 2010

COMAD User Manual 1

1. Introduction to COMAD 3
2. Installing and Running COMAD 5
3. Build Your First COMAD Workflow 7
   - *3.1. Step1. Open a blank workflow and choose the COMAD director* 8
   - *3.2. Step2. Add CollectionComposer actor* 9
   - *3.3. Step3. Add StatisticCalculator actor* 10
   - *3.4. Step4. Add CollectionDisplay actor* 11
   - *3.5. Step5. Run the workflow* 12
4. COMAD Data Stream Composition 12
5. COMAD Actor 16
   - *5.1. Basic COMAD Actor Composition* 16
   - *5.2. Input and Output Binding Configuration* 17
     - *5.2.1. Signature* 17
     - *5.2.2. Read scope* 19
     - *5.2.3. Input port data binding* 20
     - *5.2.4. Output port data binding* 24
   - 5.3. *COMAD* Actor Classification 27
     - *5.3.1. AtomicCoactor* 27
     - *5.3.2. CompositeCoactor* 30
     - *5.3.3. Extended AtomicCoactor* 33
   - *5.4. Provenance Recording* 34
6. COMAD Path Expression Syntax 37
   - *6.1. Path Element* 38
   - *6.2. Qualifier* 39
   - *6.3. Cardinality* 42
   - *6.4. Decoration* 43
     - *6.4.1. Life management* 43
       - 6.4.1.1. Creation 43
       - 6.4.1.2. Deletion 46
     - *6.4.2. Exact type match* 47
     - *6.4.3. Split of structural type* 48
   - *6.5. Port Reference* 49
7. Type System 51
   - *7.1. Type Category* 51
     - *7.1.1. System Built-in Type* 51
     - *7.1.2. User-Defined Type* 55
   - *7.2. Type Compatibility and Data Format conversion* 57
8. Demo 60
   - *8.1. Comet* 60
   - *8.2. PICalculation* 63

8.3. MobyService ... 63
9. Appendix: Actor Reference ... 64
9.1. CollectionComposer ... 65
9.2. CollectionReader ... 65
9.3. CreateRootCollection ... 65
9.4. CollectionDisplay ... 65
9.5. XmlDisplay ... 66
9.6. TraceWriter ... 66
9.7. StartLoop ... 67
9.8. EndLoop ... 67
9.9. Filter ... 67
9.10. CompositeCoactor ... 68

1. Introduction to COMAD

Usually the scientific data is in hierarchical structure with collection or nested collection of data. For example, the meteorological data collected from multiple stations is organized in multiple levels of collections based on the geographical locations, including station, country or state etc. Different kinds of analysis is made by summarizing or comparing different levels of the data to study the way in which multiple environmental factors, including climate variability, affect major ecosystems.

In the workflow of such scientific system, besides the necessary data analysis logic, a lot of shim units must be included to assemble/disassemble the data collections, convert the data into expected format for each analysis and keep the association between the input and generated data. Such none functional while necessary data processing not only makes it hard to model the workflow, but also make it hard to understand the workflow, especially to the scientists, since the main analysis pipeline is totally hidden inside the large amount of none functional processing units. Moreover, such workflow is also not easy to be maintained or evolve.

COMAD (Collection-Oriented Modeling and Design) is proposed to support design, modeling and execution of the scientific application with the collection-oriented data. In COMAD, such data is flattened from a tree-structure into a stream without any loss of information. The stream consists of multiple tokens. Besides the token representing the concrete data value, there're special delimiter tokens to denote the start and end of the collection. In this way, the original tree structure of the data is still kept. Each processing unit in the workflow is called actor. The data stream flows through the whole workflow from one actor to another. As the workers besides the assembly line, the actors in the COMAD workflow pick up the data they're interested from the stream, process it and put the output back to the stream. A COMAD path denoting expression is used to declare where to pick up and output the data in the stream.

COMAD brings the following benefits to the workflow modeling and execution.

Clear view to the original scientific data analysis process:
The actor in COMAD doesn't need to deal with the assembly/disassembly of the data collection anymore since all the collections are already disassembled when the data is flattened into the COMAD token stream. Once it's declared where to pick up and output the data in the stream, the system will automatically grab the data from the stream, feed the data into the actor after necessary type check and conversion, and finally write the data output by the actor into the target location of the data stream. Such easy-handling of the structured data and automatic data massaging behavior removes a bunch of shim units from the workflow and make the workflow very neat. Usually, the COMAD workflow is linear. It presents a very clear view to the original scientific data analysis process. It's easy to be modeled and understood, especially by the scientists.

Easy actor development:

The COMAD actor developer can focus on the data processing logic while leave all the other work, like input data grabbing, output data writing, buffering, validation and conversion etc., to the system. All the data processing logic is implemented in one method which will be invoked by the system with the prepared input data. The result of the data processing will be returned and then how to write it back to the stream will be handled by the system. Meanwhile, simplicity is also very good for the reliability of the actor development.

High reusability and adaptability of the actor:
COAMD actor is very easy to be reused to assemble different workflow. During the actor development phase, the signature is defined to declare what kind of type is expected for each input and output data port. And then during the workflow assembly phase, the concrete data for each port is specified through the data binding by using the COMAD path expression. Signature makes it easy to know how to use the actor correctly. And the data processing logic inside the actor only deal with the structure-independent data. By changing the data binding to adapt to the data stream with different structure, the actor is easy to be reused to assemble different workflow. The high reusability and adaptability of the COMAD actor meet the evolve requirement of the scientific workflow with exploiting feature. For scientific application, change is normal.

Improved performance due to the streaming mode:
In the COMAD workflow, data is re-structured into a stream flowing through each actor. Therefore the workflow performance is improved since multiple actors could work on different part of the data stream at the same time.

Great support for provenance:
To trace back how each data comes from, the provenance information, including each insertion and deletion operation, is recorded in the data stream. (In COMAD, once the data item is created, it's not allowed to be modified. So there's only deletion and insertion instead of modification operation.) By using the tool of "Provenance Browser", the provenance information could be easily browsed and queried.

Potential powerful workflow management ability:
By analyzing the input data structure and each actor's COMAD path expression about how to bind the input and output data with the data stream, it's possible to know how the data stream is changed by each actor, how the actor depends on each other, when the input data is ready and then the actor is fired. Such information is very valuable to the workflow system management. It could be used to test the correctness of the workflow configuration before the workflow execution. For example, it's easy to find out whether there's an actor never be fired due to the wrong data binding expression. It's also possible to change the configuration of the workflow during the workflow executing phase for some important management purpose, like fault tolerance or load balance etc.
COMAD is built on top of Kepler workflow system. Actually it's a special computation model of Kepler. For more details about Kepler, please refer to the https://kepler-project.org/.

2. Installing and Running COMAD

The COMAD system could be installed through the module manager of Kepler or from the svn repository. The module manager is mostly suggested since it provides a convenient way to install and run different suite of Kepler.

To install COMAD system from module manager, firstly you need to install and run the Kepler system and then do the following steps:

1. select the Tools menu of kepler and then select the module manager in the drop down menu to open the module manager as showed in Figure 1.

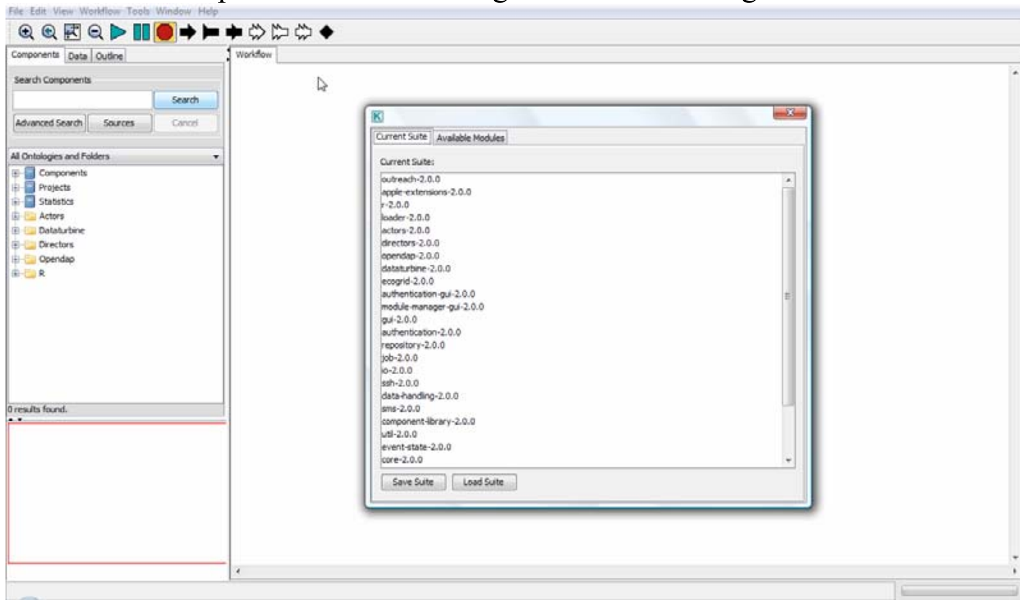

Figure 1. Opened Module Manager Window in Kepler

2. In the module manager, select the “Available Modules” tab.
3. Choose the comad-exp suite from the available suites, click the right arrow and then this suite appears in the “Selected Modules”.

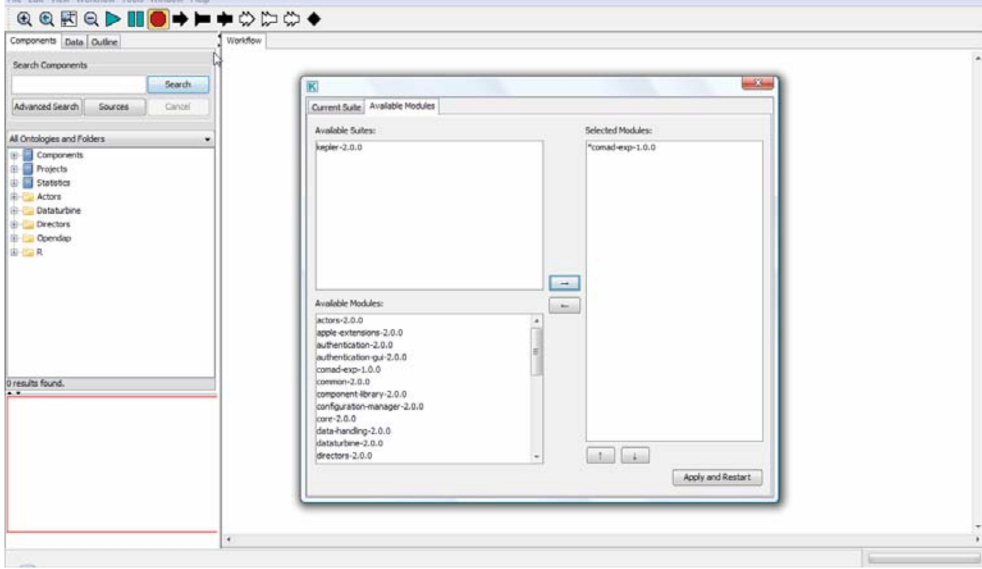

Figure2. Select comad-exp Suite for Install

4. click the “Apply and Restart” button in the lower right corner. The original opened kepler window will be closed and a new window will be opened automatically. The comad-exp is installed successfully now. If the kepler is installed at <*install_dir*>, then the <*comad-exp_install_dir*> is <*install_dir*>/comad-exp-1.0.0

To install the COMAD system from the svn repository, you need do the following steps:

1. build up required environment
   a. install JDK1.5 or higher version
   b. install Ant 1.7.1 or higher version
   c. install SVN client 1.5 or higher version
2. download the build system with the following commands:
   mkdir <*install_dir*>
   cd <*install_dir* >
   svn co https://code.kepler-project.org/code/kepler/trunk/modules/build-area
   cd build-area
3. retrieve COMAD module with the following commands
   ant change-to –Dsuite= comad-exp
4. Start COMAD module with the following commands. The <*comad-exp_install_dir*> is <*install_dir*>/comad-exp.
   cd <*comad-exp_install_dir*>
   ant run

Until now, the Kepler workflow system should be started and you should see its GUI like this:

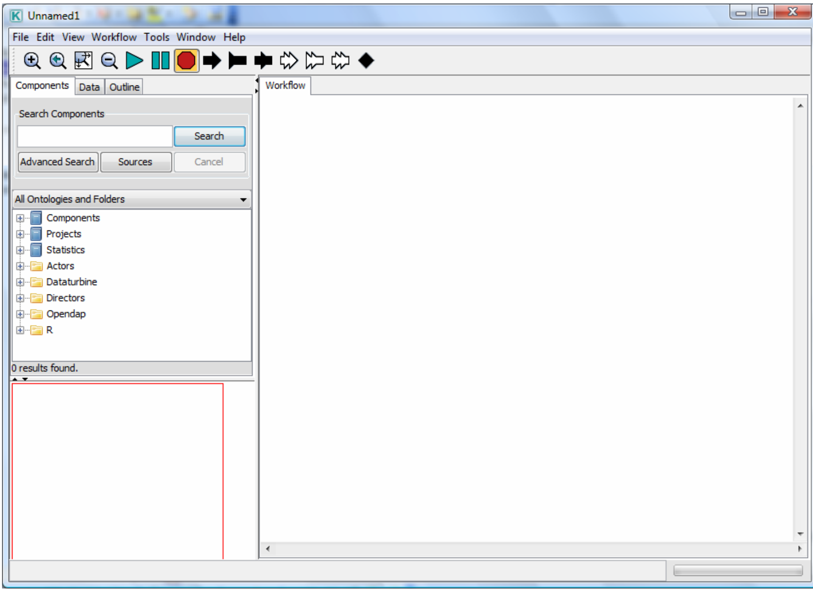

Figure3. Kepler Workflow System GUI

To run the COMAD workflow, like the workflow making statistics of geographical data, you need do the following steps:

1. Go to the “File” menu in toolbar of the GUI and select the “Open File” item.
2. In the popup file browser, go to the COMAD simple demo workflow directory which locates at <*comad-exp_install_dir*>/workflow/demo/Simple/, and select the Statistic.xml to open the Statistic workflow.

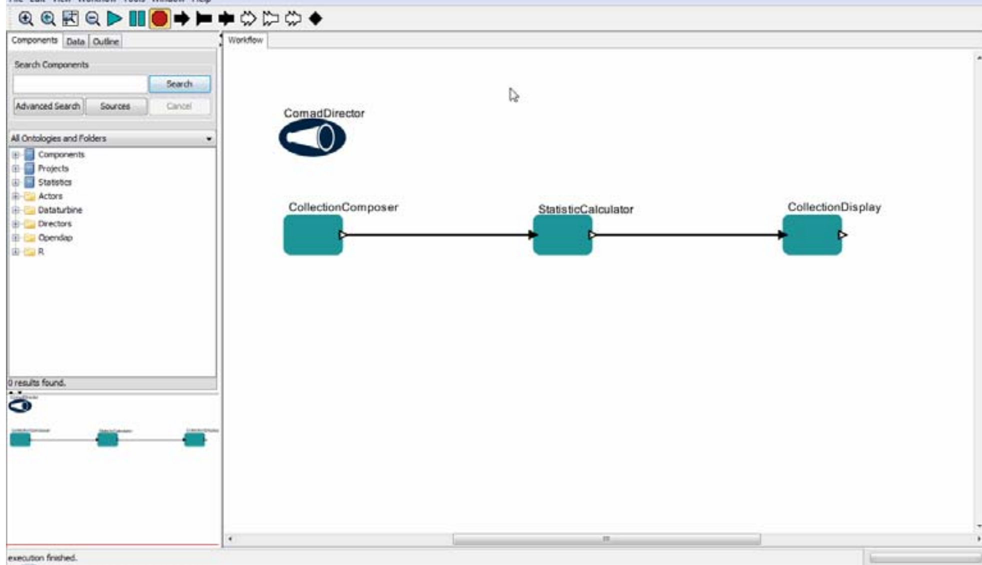

Figure4. The open COMAD Statistic workflow

3. Click “Run” button (the big green triangle) from the toolbar to run the Statistic workflow. Once the execution is successfully finished, a text display window will pop up to show the execution trace.

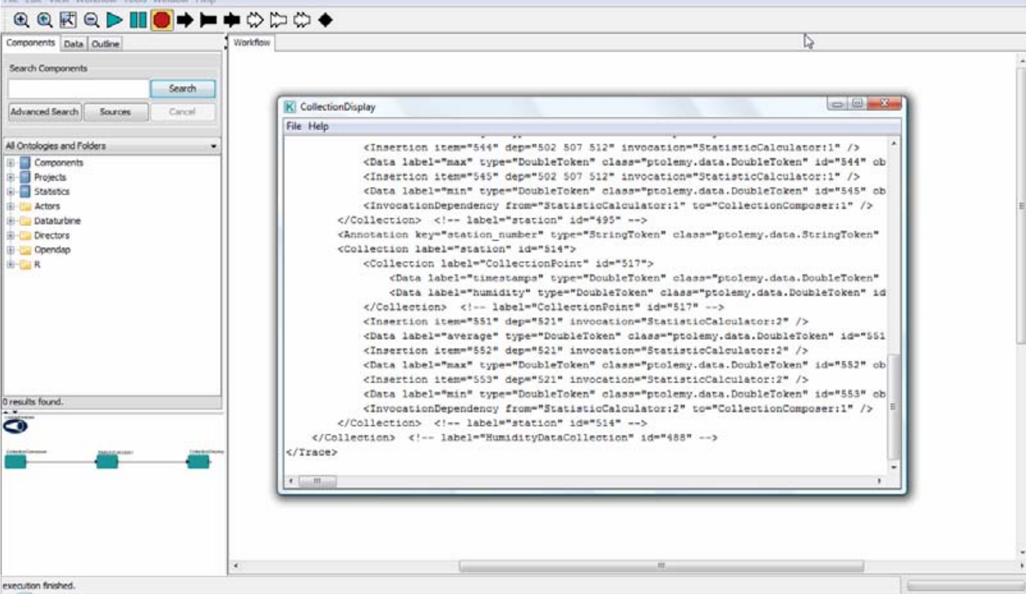

Figure5. The execution trace of the Statistic workflow

3. Build Your First COMAD Workflow

Before we go to any technical details of the COMAD workflow system, let’s build up your first simple COMAD workflow: Statistic workflow.

The Statistic workflow receives the geographical data collected from multiple stations and makes simple statistics analysis. It's composed by three actors:

- CollectionComposer: As the source of the workflow, this actor gets the external input data and converts it into the inner COMAD data stream.
- StatisticCalculator: This actor make static analysis based on the data grabbed from the data stream and puts the statistic result back into the data stream.
- CollectionDisplay: This actor shows how the final data stream looks like.

The complete Statistic workflow could be viewed and executed at "<*comad-exp_install_dir*>/workflow/demo/Simple/Statistic.xml". The source code for the involved actor could be found under the directory "<*comad-exp_install_dir*>/src/org/kepler/demo/simple".

3.1. Step1. Open a blank workflow and choose the COMAD director

Go to the File menu in the toolbar and navigate to the New Workflow -> blank menu, or simple type Ctrl+N to open a blank workflow.

Besides the actors implementing specific processing logic, each Kepler workflow must have a director to decide how the constituted actors communicate and cooperate during the workflow execution, just as the movie director. For the COMAD workflow, its director is ComadDirector. To add ComadDirector into the workflow:

1. Go to tool menu and click the "Instantiate Attribute" item to open the "Instantiate Attribute" window.
2. In the text field for the "class name", type "org.kepler.domains.ComadDirector" and click "ok".

The empty COMAD workflow looks like:

Figure 6. Empty COMAD workflow

3.2. Step2. Add CollectionComposer actor

1. Create CollectionComposer actor:
    1) Go to tool menu and click the “Instantiate Component” item to open the “Instantiate Component” window.
    2) In the text field for the “class name”, type “org.kepler.coactors.CollectionComposer” and click “ok”.
2. Configure the CollectionComposer with the input data by double clicking its icon. And click the “Commit” button when the configuration is finished. Two things need to be configured for the input:
    1) Input schema: choose “Native Comad” as the input format
    2) Input value: paste the following text as the input value. It’s the humidity data collected from stations “s2” and “s5”. For each data collection point, it contains the timestamp and the humidity data at that time.

```
<Annotation key="start_time">"01-01-2008"</Annotation>
<Annotation key="end_time">"01-01-2009"</Annotation>
<Collection label="HumidityDataCollection">
    <Annotation key="station_number">"s2"</Annotation>
    <Collection label="station">
            <Collection label="CollectionPoint">
                    <Data label="timestamps">1.196499599E9</Data>
                    <Data label="humidity">29.700001</Data>
            </Collection>
            <Collection label="CollectionPoint">
                    <Data label="timestamps">1.196503199E9</Data>
                    <Data label="humidity">28.799999</Data>
            </Collection>
            <Collection label="CollectionPoint">
                    <Data label="timestamps">1.196510399E9</Data>
                    <Data label="humidity">29.200001</Data>
            </Collection>
    </Collection>
    <Annotation key="station_number">"s5"</Annotation>
    <Collection label="station">
            <Collection label="CollectionPoint">
                    <Data label="timestamps">1.196499599E9</Data>
                    <Data label="humidity">36.799999</Data>
            </Collection>
    </Collection>
</Collection>
```

Figure7. Input data of the Statistic workflow

The configuration of CollectionComposer looks like:

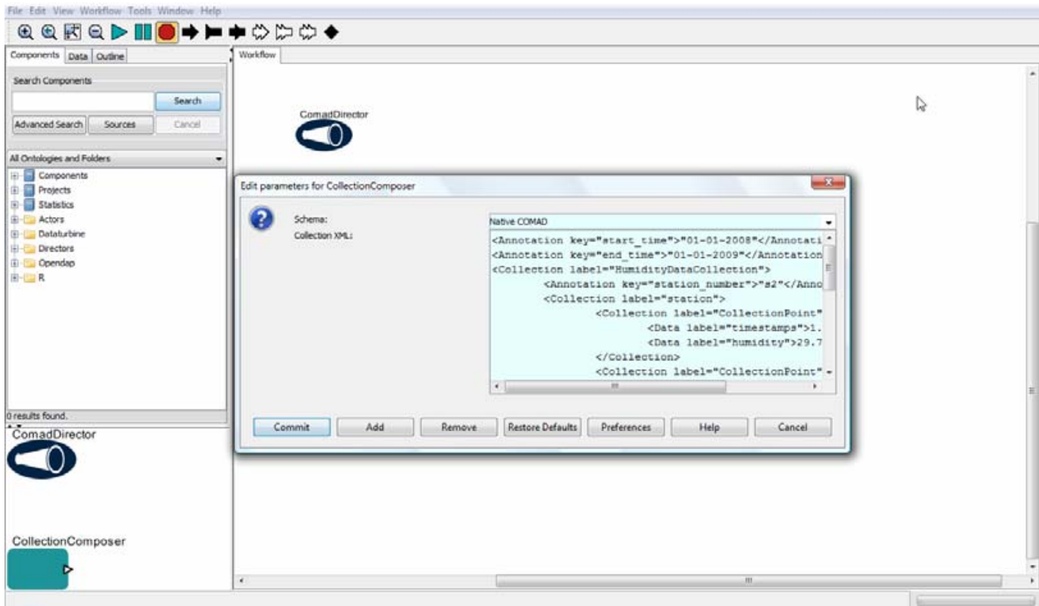

Figure8. CollectionComposer Configuration

For more information about input collection and input data format supported by COMAD, please refer to the chapter of “COMAD Data Stream Composition”.

3.3. *Step3. Add StatisticCalculator actor*

To implement specific function, we could choose to develop a composite actor by encapsulating a sub-workflow assembled from a group of existing actors. Or we could choose to develop an atomic actor from scratch, which is more flexible. For simplicity, we use the atomic actor in the Statistic workflow. You may check the Statistic_Composite workflow to see how the same function is implemented by using the composite actor. It’s under the same directory as the completed Statistic workflow.

1. Create the StatisticCalculator actor in the similar way as we create the CollectionComposer actor by typing “org.kepler.demo.simple.StatisticCalculator” for the class name.
2. Connect the StatisticCalculator with the CollectionComposer by dragging a line between the input port of StatisticCalculator and the output port of CollectionComposer.
3. Configure the input and output of StatisticCalculator in the configuration window by double clicking its icon. There’re six parameters:
   - Signature: This parameter is set by the actor developer to define the input and output of this actor, including port name, expected type and cardinality. This actor reads a list of DoubleToken from valueList port, and output statistic result DoubleToken from the avg, max and min port separately. The setting of the data binding must be consistent with the Signature requirement.
   - ReadScope: This parameter defines the visible range of the data stream to the actor to grab the input data or write the output data. To process data for each station, put “/HumidityDataCollection/station” here.

- valueList: This parameter defines how to pick up the input data from the data stream for the valueList port. To give the humidity data from all collecting point of a station in a list to the actor as required by the signature, put "//DoubleToken[@label=="humidity"]+" here.
- avg: This parameter defines how to write the output data from port avg to the data stream. Put "/DoubleToken[@label=="average"]" here to write the data labeled as "average" in the analyzed station collection.
- max: This parameter defines how to write the output data from port max to the data stream. Put "/DoubleToken[@label=="max"]" here to write the data labeled as "max" in the analyzed station collection.
- min: This parameter defines how to write the output data from port min to the data stream. Put "/DoubleToken[@label=="min"]" here to write the data labeled as "min" in the analyzed station collection.

The configuration of the StatisticCalculator actor looks like:

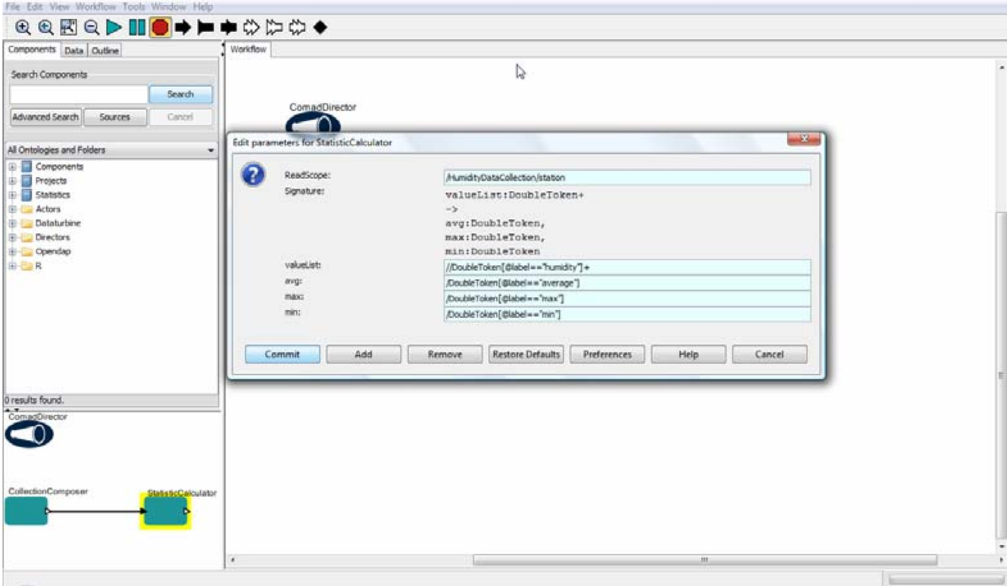

Figure9. StatisticCalculator Configuration

For more information about how the COMAD actor is configured and how they work, please refer to the chapter of "COMAD Actor". For more information about how the syntax of signature, read scope and data binding, please refer to the chapter of "COMAD Path Expression Syntax".

3.4. *Step4. Add CollectionDisplay actor*

To show the final result and the provenance of the workflow, the CollectionDisplay actor is added.

1. Create CollectionDisplay actor in the similar way to create the other actor by typing "org.kepler.coactors.CollectionDisplay" for the class name.
2. Connect the StatisticCalculator with the CollectionDisplay by dragging a line between the input port of CollectionDisplay and the output port of StatisticCalculator.

3. Configure the CollectionDisplay actor by double clicking its icon. The CollectionDisplay is a special actor without data binding. It simply shows the content of the data stream inside the read scope. Put "/" here to show the whole data stream. "/" represents the root collection which also equals to "/HumidityDataCollection".

The configuration of CollectionDisplay actor looks like:

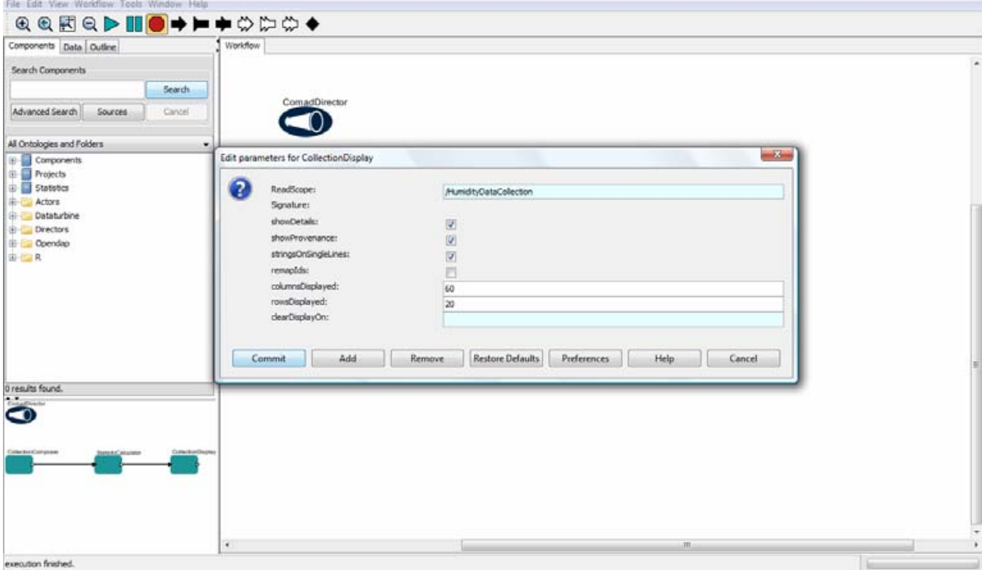

Figure10. CollectionDisplay Configuration

3.5. Step5. Run the workflow

The completed Statistic workflow is demonstrated in Figure 4. To run the workflow, click the big green triangle in the tool bar and then the final execution data stream got from the workflow is showed in the pop up text window as Figure 5 demonstrated. The data stream not only contains the input and output data, but also records the execution provenance information, including what actors and configurations are in the workflow and how the data stream is changed by each actor with insertion or deletion operation.

4. COMAD Data Stream Composition

Logically the COMAD data stream consists of three kinds of entities: collection, data and annotation. The structure of COMAD data stream is a tree, similar to XML.

- Collection: The collection is a container of a group of data. A collection could also be contained by another collection. The top collection corresponds to the whole data stream is called root collection. Each collection has an attribute of label and any number of annotations as needed. The label is optional. But we strongly suggest assigning a label for each collection. Therefore some advanced features could be used, like static analysis.
- Data: The data represents the concrete data value. Each data has attributes of label and type. It can also have any number of annotations as needed. Both label and type are optional. When type is absent, it will be inferred from the data value

automatically. COMAD support multiple type by default as well as user defined type. Please refer to the chapter of "Type System" for details.

- Annotation: The annotation is used to tag the collection or data to denote it has specific property. The annotation just appears before the target collection or data it annotates. Some annotations are used to annotate the root collection with the workflow system configuration information, including what actors and parameters are used. They're called run annotations and they're annotated by the system automatically. In spite of this kind of annotation, the user can use the normal annotation or a kind of special annotation, called metadata annotation, to annotate the collection or data. The only difference between these two is the metadata annotation is disallowed to be deleted. The normal annotation and metadata annotation have attributes of key and type. The newly added annotation will result in deletion of the previously existing annotation with the same key. The annotation is inheritable. That means all the collection or data inside one collection will inherit its annotation. The annotation with the same key as the inherited annotation will overwrite it.

The external input data of COMAD workflow could be in two formats:

- Native COMAD format: It's a XML format with special schema. The data in this format is directly organized in the COMAD logical data structure, with elements of Collection, Data and Annotation.
    - The geographical input data of the Statistic workflow in Figure 7 is in this format. The "start_time" and "end_time" annotations of the root collection "HumidityCollection" tags that the data is collected between 01-01-2008 and 01-01-2009. Inside the root collection, there're two "station" collections enclosing data from "s2" and "s5" stations separately. Each "station" collection has a "station_number" annotation to tag which station it represents and multiple "CollectionPoint" collections containing timestamp and the humidity data. The logical structure of this input is demonstrated in Figure 11. For simplicity, three CollectionPoint collections of station s2 are not fully expanded, which has the similar structure as the CollectonPoint collection of station s5.

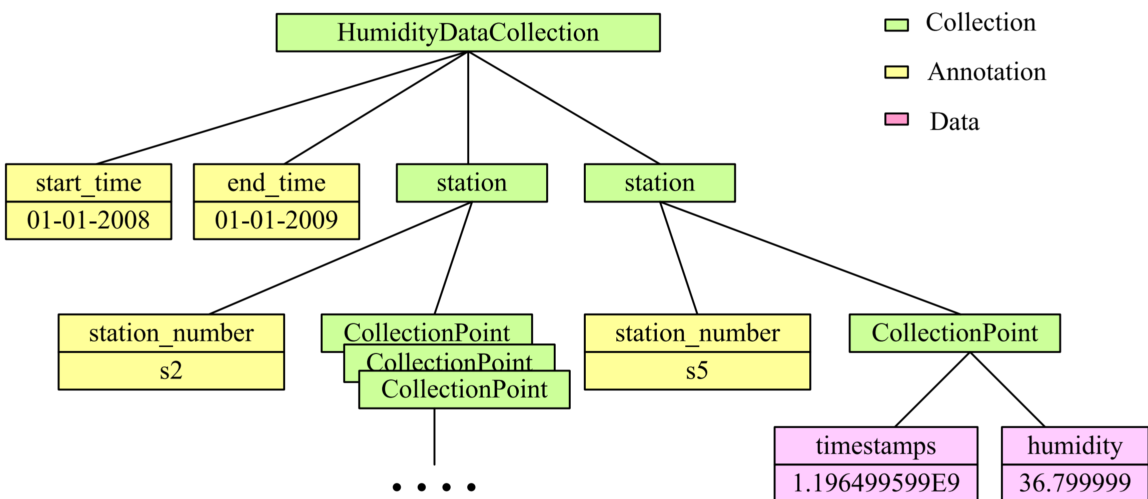

Figure11. Logical Structure of Statistic Workflow Input

- General XML format: The input data could be in any valid XML format. Inside the system, the general XML format is mapped to the native COMAD format with the following rules.
  - Element node
    - If the node has multiple child node, it's mapped to the COMAD Collection while its name is mapped to the label of the Collection.
    - If the node only has one Text node or CData node, it's mapped to the COMAD Data while its name is mapped to the label of the Data and value of the Text or CDadta node is mapped to the Data value. The difference between Text and CData mapping is, the CData is mapped to a Data with type of StringToken while the Text is mapped to a Data whose type is inferred from its value automatically.
  - Attribute is mapped to the annotation.
  - Text node or CData node inside an Element node together with the other nodes is mapped to the COMAD Data without label.

The input in general XML format equal to the input in native COMAD format in Figure 7 is demonstrated in the following. The Statistic workflow with such input format could be found in the "<*comad-exp_install_dir*>/workflow/demo/Simple/Statistic_XmlInputFormat.xml":

```
<HumidityDataCollection start_time="01-01-2008" end_time="01-01-2009">
        <station station_number="s2">
                <CollectionPoint>
                        <timestamps>1.196499599E9</timestamps>
                        <humidity>29.700001</humidity>
                </CollectionPoint>
                <CollectionPoint>
                        <timestamps>1.196503199E9</timestamps>
                        <humidity>28.799999</humidity>
```

```
            </CollectionPoint>
            <CollectionPoint>
                <timestamps>1.196510399E9</timestamps>
                <humidity>29.200001</humidity>
            </CollectionPoint>
        </station>
        <station station_number="s5">
            <CollectionPoint>
                <timestamps>1.196499599E9</timestamps>
                <humidity>36.799999</humidity>
            </CollectionPoint>
        </station>
    </HumidityDataCollection>
```

Figure12. Input in general XML format

During the workflow execution phase, all the information is mapped into a token stream, including all the logical entities of data and the recorded provenance information etc. The expert user who are familiar with the data stream composition, related data structure and processing mechanism, are provided with a group of API to access each token in the data stream. Therefore any fancy data processing function could be implemented. For more details, please refer to “Exteneded AtomicCoactor” chapter. The category of COMAD token is summarized in Table 1. Each token contains provenance about how it’s inserted.

Table1. COMAD Token Classification.

Token Name	Description
OpeningDelimiterToken	denote the start of a collection
ClosingDelimiterToken	denote the end of a collection
DataToken	represent the entity of data
AnnotationToken	represent the entity of annotation
ActorRegistrationToken	a special AnnotationToken recording what actor is involved in the workflow
ParameterToken	a special AnnotationToken representing parameter, like parameter of actor
MetadataToken	a special AnnotationToken which is not allowed to be deleted once it’s created
ExceptionToken	enclose the exception happened during workflow execution
LoopTerminationToken	denote the end of loop. This token together with StartLoop and EndLoop actors are used to implement loop structure in COMAD. Please refer to PI_Calculation workflow in demo.
DeletionRecordToken	enclose provenance information about how the collection, data or annotation is deleted
InvocationDependencyToken	enclose provenance information about how invocations of actors depend on each other

For example, the sub-tree representing “s5” collection of Statistic workflow in Figure 11 is mapped into the following token stream:

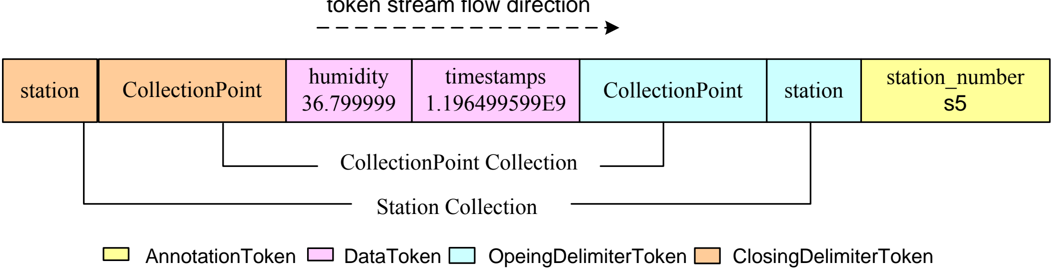

Figure13. Token stream mapped from s5 collection

5. COMAD Actor

Actor is the basic building block of the workflow. As a worker besides the assembly line, the COMAD actor of workflow sits on the data processing pipeline, picks up the interested data, processes it and puts the result back into the data stream.

5.1. Basic COMAD Actor Composition

A COMAD actor comprises two parts:

- Data processing logic: It defines how the actor processes the input data and produces the output. The COMAD actor could be implemented from scratch by inheriting specific actor and overwriting specific methods to implement the data processing logic. The actor can also be implemented as a CompositeCoactor to encapsulate a sub-workflow composed by a group of existing actors.
- Parameter: The actor is customized through the parameter. There’re two kinds of parameters.
    - Functional parameter: It’s used to customize the function of actor. For example, the actor generating multiple time windows has an interval parameter which defines the interval for each window.
    - Input and output binding configuration parameter: It defines how to grab the input data from the data stream and write the output data into the data stream. Three parameters belong to this category: read scope, signature, input and output port data binding.

The duty of the actor developer is:

- Implement the data processing logic
- Define the functional parameter if it exist
- Define the signature parameter to declare what data is expected to be input and output just as the signature of the function in the programming language.

The duty of the workflow assembler to use the actor is:

- Configure the functional parameter as needed if it exists
- Configure the parameter of read scope, input and output port data binding to bind the input and output port of the actor to the concrete data stream.

The actor is easily developed since the actor developer only needs to focus on the data processing logic. Meanwhile, the actor is easily reused since the expected input and output is clearly declared through the signature and the changing of binding configuration parameters make the actor easily adapt to the different data stream in different workflow.

In the rear of this chapter, the input and output binding configuration parameters are introduced firstly. Then it's explained how the actor is fired in case of different input data bindings. Finally the classification of the COMAD actor and its major APIs are elaborated.

5.2. Input and Output Binding Configuration

The signature, read scope, input and output data binding are defined through the COMAD path expression. Please refer to the chapter of "COMAD path expression syntax" for more details.

5.2.1. Signature

Signature is defined by the actor developer to declare the requirement on the input port and the expected data on the output port. The input and output port data binding defined by the workflow assembler must be consistent with the signature.

For each port, signature defines the expected type and cardinality. The type could be any system built-in type, like StringToken, IntegerToken etc. It can also be user defined type. There're four choices of cardinality: one; ? (zero or one); + (one or more) and * (zero or more). Since ? and * could be zero, they have optional meaning while one and + have none optional meaning. For example, the signature of input port valueList in SatisticCalculator actor is "DoubleToken+". It means that valueList port is a none optional port and in each firing it must provides a none empty list of DoubleToken as input.

The signature of actor is composed by the signature element of all input and output ports. Each signature element comprises port name and signature definition separated by semicolon. The input signature elements are separated from the output ones by arrow which indicates the direction from the input to the output. Multiple input or output signature elements are separated by comma. For example, the signature of SatisticCalculator actor is defined in figure 12. It declares that the actor receives a list of DoubleToken and outputs statistic value of one average, min and max in type of DoubleToken.

```
valueList:DoubleToken+
->
avg:DoubleToken,
max:DoubleToken,
min:DoubleToken
```

Figure14. Signature of StatisticCalculator Actor

Both input and output port data binding must be consistent to the corresponding signature. The consistency between the input port data binding and signature guarantee the actor get what it needs when it’s fired. The consistency between the output data binding and signature guarantee the output of actor is the data expected to be written into the data stream. If there’s any inconsistency, an exception is thrown out. The consistency test includes cardinality consistency test and type consistency test, as illustrated in table 2 and table 3 separately. The type of signature, input and output port data binding is actually decided together by the declared type and cardinality. The type declared in signature or data binding is denoted as basic-type while the type of signature or data binding is denoted as signature-type or binding-type. For data binding like “/DoubleToken+”, the basic-type is DoubleToken while the binding-type is List<DoubleToken>.

Table2. Cardinality Consistency Test

Data Binding Port		Signature Cardinality	Consistent
Category	Cardinality		
Input Port	? or *	One or +	No. It’s inconsistent because there might be no input data at this port when the actor is fired but the actor expects at least one input data.
	One or +	? or *	Yes.
Output Port	One or +	? or *	No. It’s inconsistent because there might be no output data at this port after the actor is fired but the binding expects at least one output data.
	? or *	One or +	Yes.

Table3. Type Consistency Test

Data Binding Port Category	Type Relations	Consistent
Input Port[*1]	binding-type ≤ signature-type[*3]	Yes.
	! binding-type ≤ signature-type	No.
Output Port[*2]	signature-type ≤ binding-type	Yes.
	! signature-type ≤ binding-type	No.

*1: Binding-type of input port, is the type of data prepared by this port to fire actor.
*2: Binding-type of output port, is the type of output data expected to be written into data stream by this port after actor is fired.
*3: “≤” means the right side type is compatible to the left side type. One type is compatible to another if the first type is allowed to be converted to the second one.

Sometimes, the type of input port data binding is not clear and fixed due to the complex selecting condition used in path expression. In this case, the type consistency test between input port data binding and signature is actually done by testing the consistency between the real bound input data and the signature. Such problem doesn’t exist for the output port data binding since the type for the newly created value is clearly declared in the output port data binding path expression.

Besides consistency between data binding and signature, the real output data must also be consistent to signature. They're consistent as long as the type of the output data is compatible to the type of signature.

A pair of data binding and signature with consistent type is possible to have different format. For example, in COMAD type system, ArrayToken(DoubleToken) (It's an ArrayToken with element type of DoubleToken) is consistent to List<DoubleToken> defined as DoubleToken+. If actor expects to receive a list of DoubleToken, the ArrayToken data bound through the input port data binding needs to be converted into a list before it's fed into the actor. The same thing happens between the output data and output port data binding. In conclusion, the bound input data or the output data needs to be converted into the target format defined by the input port signature or the output port data binding when their formats are different.

All the above consistency test and format conversion is done by COMAD system. Such automatic input and output data messaging for the actor makes the actor development and reuse easier. For more details about what type is supported, how the cardinality affects the type, what's the compatible relationship between two types, and how the data is converted between different formats, please refer to the chapter of "Type System".

5.2.2. Read scope

Read scope defines a collection inside the data stream where input data could be picked up from and output data could be written into.

Read scope actually defines a segment of data stream visible to the actor. When the data stream passes through, the actor firstly tries to locate the collection matches to the read scope. If it fails, the actor does nothing. As long as a collection is matched and read scope is entered, the actor is invoked to navigate in this collection to find the interested input data, make processing or write data into this collection. The collection matched to the read scope is the starting point from where the actor will look for the collection, data or annotation matched to the data binding path expression, while the data binding path expression is also declared relative to the read scope. How many times the read scope is entered, how many times the actor is invoked how many times.

The read scope is not entered in nested way. Once read scope is entered, the collection inside the read scope collection won't be treated as read scope anymore even if its path also matches to the read scope.

The read scope of StatisticCalculator actor in Statistic workflow is defined as "/HumidityDataCollection/station". Both s2 and s5 station collection will be matched to this read scope. The StatisticCalculator will be invoked twice.

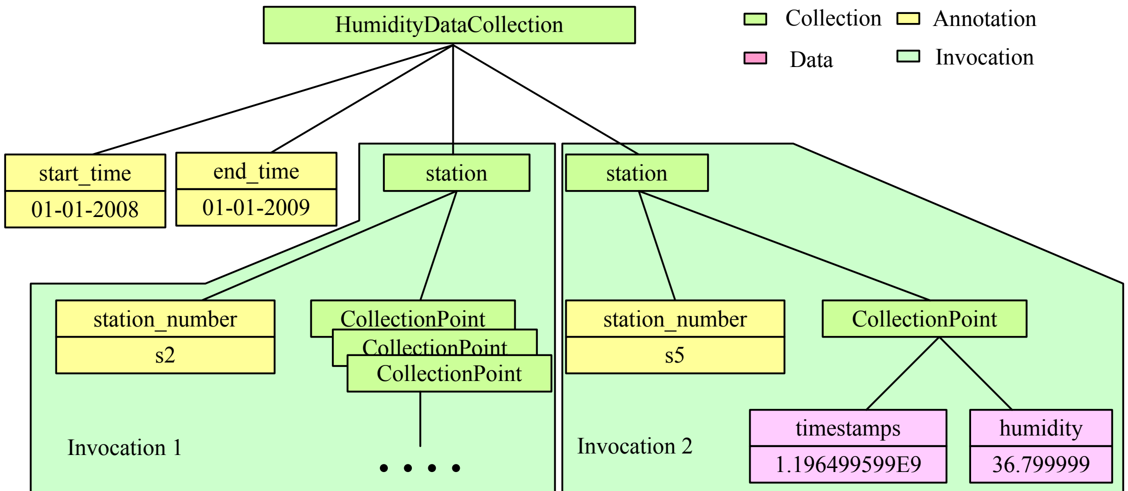

Figure15. Read Scope and Invocation of StatisticCalculator. Although the start_time and end_time annotations of root collection are not explicitly included in either invocation in the graph, yet they're inherited implicitly by each collection and data inside the root collection. Therefore they are also possible to match the input port data binding expression and fire the actor.

5.2.3. Input port data binding

The input port data binding declares where the input data of this port comes from. It could come from three places:

- Data stream: The input port data binding denotes to a data or annotation in the data stream through path expression, like "//DoubleToken". It's not an absolute path to the location where the interested data is in the whole data stream. It's a path relative to the read scope.
- Workflow parameter: The input port data binding denotes to an existing parameter of the workflow, like $interval. The "interval" is a workflow parameter.
- Literal: The input port data binding denotes to a literal using the Ptolemy expression. For example, the expression of "hello" represents a StringToken data with value of "hello". For more details, please refer to the Ptolemy tutorial at http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-28.html.

The input port data binding with const value by referring to workflow parameter or using literal is called const input port data binding. Otherwise, it's a none-const input port data binding. The const value of the const input port data binding is the input data of this port. Same as the input data picked up from the data stream through the none-const input port data binding, it must be consistent to the signature. Please refer to the "Signature" chapter for the consistency test.

Inside read scope collection, whenever a data or annotation arrives, the actor will firstly test whether it matches to any of the input port data binding and then put it into the buffer of the matched port. One data or annotation is possible to match to multiple data binding and put into the multiple buffers. After that, it's tested whether the input of actor is ready. Once it's ready, the actor is fired. Otherwise the actor just waits until the next data or annotation.

Whether the incoming data or annotation is matched or not is not affected by the cardinality of the input port data binding. The match process is only related to the type declared in the data binding. For data binding like "/DoubleToken+", all the data with type of DoubleToken satisfying all the other path requirement is matched. For input data, the cardinality of input port data binding only decides how to assemble the matched data for input and when the actor is ready to be fired, which is illustrated in table 4 and table 5.

Table4. Cardinality and Input Data Assembly

Cardinality	Assembled Input Data in Buffer	Description
One or ?	------	Each matched data or annotation is put singly in one cell[*1] to fire actor with only one input data.
+ or *	------	The matched data or annotation is assembled into a list and put into each cell. The actor will be fired with a list of input data from this port. The boundary of such assembly is the previous matched element just before the data or annotation in the path expression. Please refer to the "COMAD Path Expression Syntax" chapter for more details.

*1: Each cell of input buffer holds input data of this port for each firing

Table5. Cardinality and Firing Strategy

Input Port Data Binding and Cardinality Cases	Firing Times	Description
zero input port data binding	One time	Once the read scope is entered and the actor is invoked, the actor will be fired one time if there's no input port data binding. Such strategy is useful when the actor only generate output without input.

There's at least one required binding port[*1].	null Port1: ? Port2: 1 null Port3: * Port4: + f2 f1	The firing time in each invocation is the minimum number of buffered data in all required binding ports. Port2 and port4 are required binding ports. The minimum number of buffered data for them is 2. So the actor is fired twice. In each firing, if no data is buffered in the corresponding cell for the optional binding port[*2], the null value is inserted and used as input of this port.
There's no required binding port and there's at least one optional binding port	null null Port1: ? null Port2: ? null Port3: * f3 f2 f1	The firing time is the maximum number of buffered data (including null value) in all optional binding ports. All three ports are optional. And the maximum number of buffered data is 3. So the actor is fired three times
There's only one binding port and it's a const binding port	One time	The const input port data binding is treated as a parameter. The const value of this port could be used to fire the actor multiple times as long as the other input is ready. When there's only one const binding port, the actor is fired once with its const value.

There're one const binding port and at least one required binding port	Port1: ? Port2: 1 Port3: * Port4: + Port5: const f2 f1	The firing time in each invocation is the minimum number of buffered data in all required binding ports. In each firing, the const binding port provides its const value as input. Same as the second case, the actor is fired twice. Each time, const binding port5 provides its const value c.
There're one const binding port and at least one optional binding port	Port1: ? Port2: ? Port3: * Port4: const f3 f2 f1	The firing time is the maximum number of buffered data in all optional binding ports. In each firing, the const binding port provides its const value as input. Same as the third case, the actor is fired three times. Each time, const binding port5 provides its const value c.

*1: Required binding port is none-const input port data binding with cardinality of one or +.
*2: Optional binding port is none-const input port data binding with cardinality of ? or *.

In Statistic workflow, if read scope of StatisticCalculator actor is root collection and valueList input port data binding is "/station//DoubleToken[@label=="humidity"]+", the actor will be invoked once when the root collection is entered and fired twice with a list of humidity data from each station.

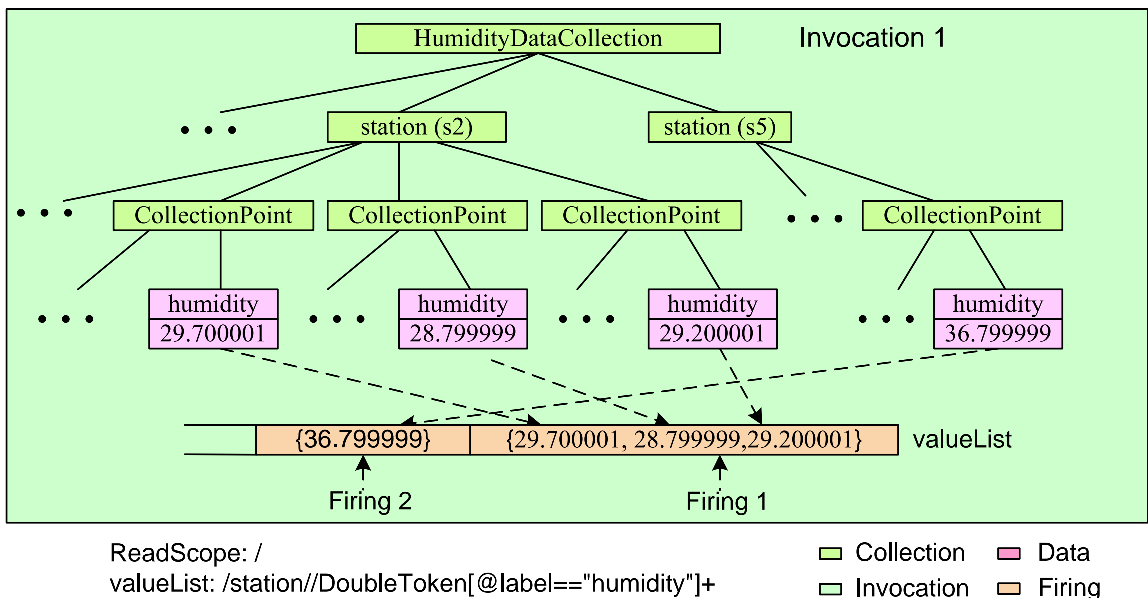

Figure16. Invocation and Firing of StatisticCalculator

For none-const input port data binding, the collection, data or annotation matched to the path expression could be deleted by using special decoration in the path expression. In this way, the filter functions could be easily implemented. The Statistic workflow that filters out the collection point with humidity data lower than 29.00 could be found under "*<comad-exp_install_dir>*/workflow/demo/Simple/Statistic_filter.xml". Please refer to "COMAD Path Expression Syntax" chapter for more details about deletion decoration.

5.2.4. Output port data binding

The output port data binding declares where the output of this port goes into data stream through a path expression. The last element in the path expression specifies what kind of data or annotation will be created at the target location defined by the other path elements using the output value from this port. The path is relative to the read scope. If there's only one element in the binding expression, the read scope collection is the target location. The newly created data goes to the last element of the target collection, while the newly created annotation goes to the last annotation in the annotation set of the target collection or data.

For example, "max" output port of the SatisticCalculator actor has binding expression of "/DoubleToken[@label=="max"]". The output value of this port is used to create a DoubleToken data with label of "max" under the read scope collection. If the binding expression is changed to "/@max[@type=="DoubleToken"]", then the output is created as an annotation of the read scope collection. The key of this annotation is max and the type is DoubleToken.

If the last element of the output data binding path expression denotes to a data without qualifier, like "/DoubleToken", then the data is created without label. A newly created data could be assigned a label with qualifier, like "/DoubleToken[@label=="hi"]". If the last element of the path denotes to an annotation without qualifier, like "/@max", then its

type is assumed to be the same type as declared in the corresponding signature. A qualifier could be used to explicitly declare the type for newly created annotation, like “/@max[@type=="DoubleToken"]”.

Output data must be consistent to the output port data binding. Such consistency is automatically satisfied, since it’s tested that the output data is consistent to signature while the signature is consistent to output port data binding. If the output data value is null, it equals empty output. In this case, an exception is thrown out if the cardinality of the data binding is one or + which expects at least one output data. Except converting the output data value into the format expected by the output port data binding, like converting a data with type of ArrayToken(DoubleToken) into a list of DoubleToken, the output won’t be converted to a value with type as exactly the same as defined by data binding. For example, an output value with type of “IntegerToken” is consistent to the data binding defined as “/StringToken”, since IntegerToken is consistent to StringToken in COMAD type system. But the IntegerToken is written into data stream instead of a StringToken.

By default the target location for the newly created data or annotation are all collections or data inside read scope matched to the output port data binding path expression. But such “write to all matched” strategy is not always useful. Sometimes, the output needs to be written to specific place related to where the input data comes from. And sometimes a group of output needs to be written to the same specific place. The port reference could be used in path expression to satisfy such reference requirement. The output port data binding may reference any path element of any input port and any other output port. The reference is declared from “#” with port name and path element index. It means the collection or data matched to that element now is the starting point to look for the target writing location for this port. In Statistic workflow, if the read scope of StatisticCalculator is root collection, the vlaueList port gets a list of humidity data from each station through “/station//DoubleToken[@label=="humidity"]+”. Without port reference, the avg output defined as “/station/DoubleToken[@label=="average"]” will write average statistic value of each station into all station collections because they are all matched target locations. To avoid such error and keep the corresponding relationships between input vlaueList port and output avg port, avg port should be defined with port reference as “#valueList[0]/DoubleToken[@label=="average"]”. “#valueList[0]/” refers to the collection currently matched to the first element of the valueList binding path. That’s the station collection where the current input data comes from. Similarly, max port is defined as “#avg[0]/DoubleToken[@label=="max"]” to output max value to the collection where the avg value is written. The cases without and with port reference are demonstrated in Figure 17 and Figure 18 separately. The complete Statistic workflow with port reference could be found at “<*comad-exp_install_dir*>/workflow/demo/Simple/Statistic_portReference.xml”.

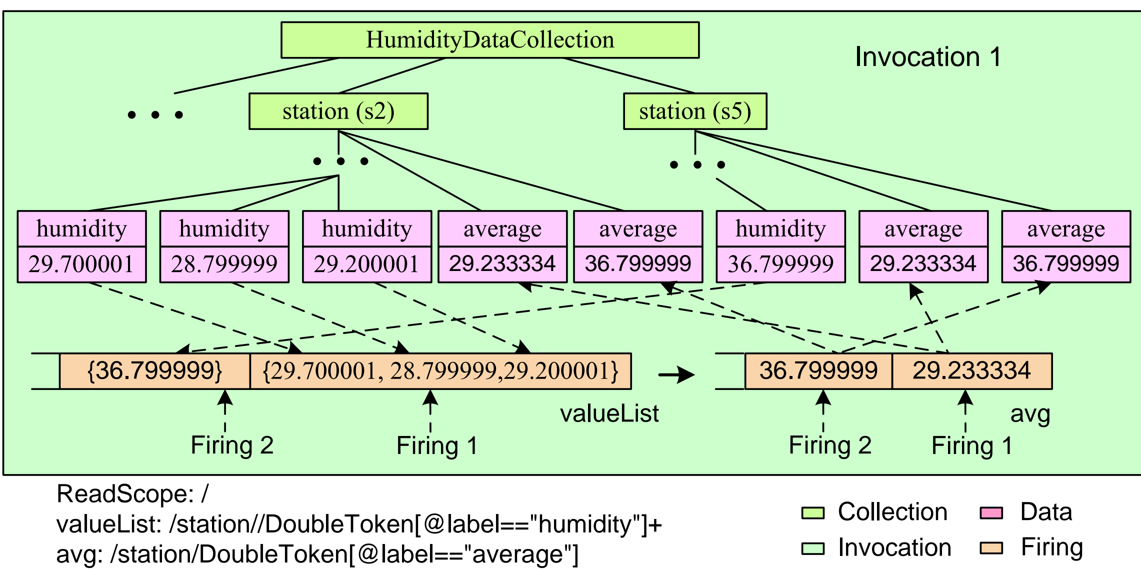

Figure17. Output Port Data Binding without Port Reference

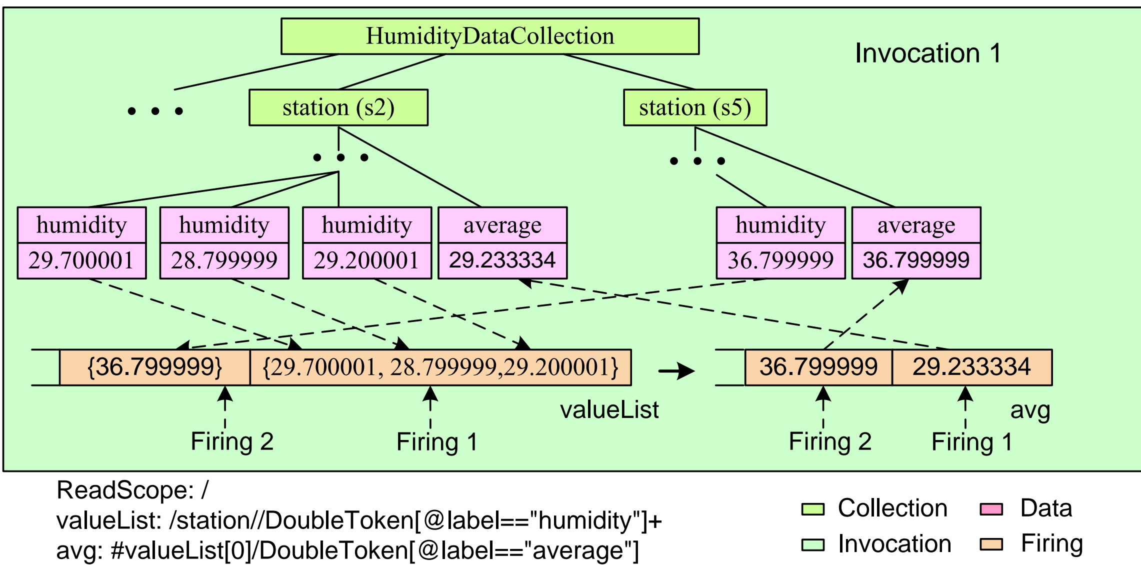

Figure18. Output Port Data Binding With Port Reference

Besides writing the output into an existing collection, the target collection could also be created to contain the written data or annotation. By using special decoration in the path expression, multiple collections could be created in each invocation, or in each firing, or as a peer of read scope collection. Also for the above Statistic workflow, if we tend to put all statistic value in a “statistic” collection clearly separated from raw data, the avg port could be defined as “#valueList[0]/{-f}statistic/DoubleToken[@label=="average"]”. Correspondingly, the max port is changed to “#avg[1]/DoubleToken[@label=="max"]”. “-f” means create collection in each firing. The creation of statistic collection is illustrated in Figure19. The completed workflow could be found at “<*comad-exp_install_dir*>/workflow/demo/Simple/Statistic_creation.xml”. For more details about creation support in the path expression, please refer to “COMAD Path Expression Syntax”.

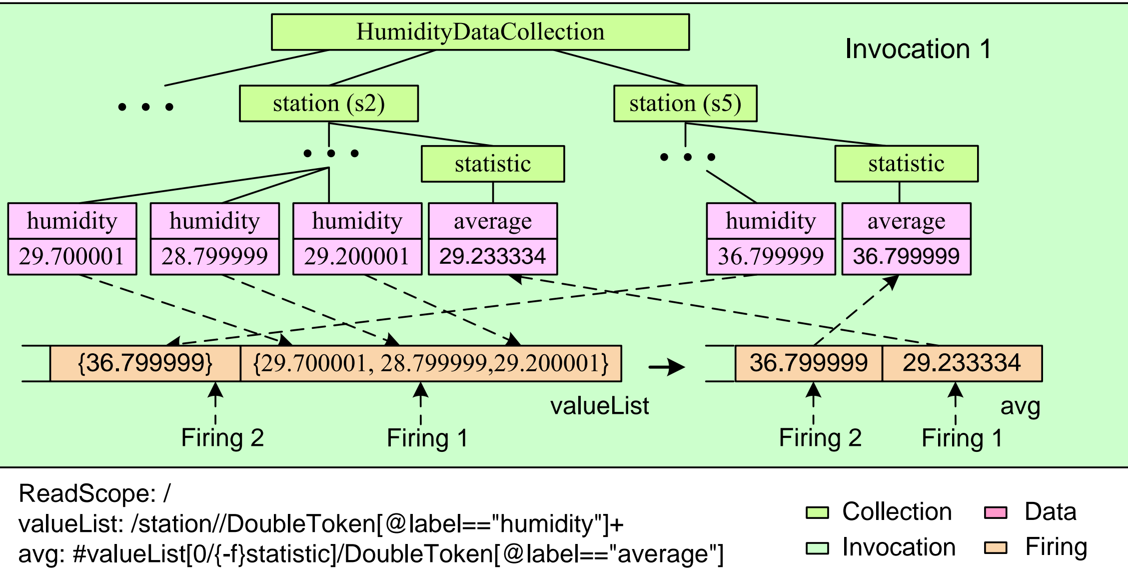

Figure19. Collection Creation in Output Port Data Binding

5.3. *COMAD* Actor Classification

Generally, COMAD actor is classified into AtomicCoactor and CompositeCoactor. CompositeCoactor is used to encapsulate a sub-workflow assembled from a group of pre-existing kepler actors by dragging and assembling them together, while AtomicCoactor is for development of new atomic actor. Usually, through data binding configuration, the input data is automatically prepared for the actor and the output is automatically written into specific place. But if such automatic data massaging can't satisfy the developer's requirement for some fancy function, they can access the raw token stream directly to do anything they like by using the ExtendedAtomicCoactor. Since it's required to be very familiar with the composition of the underlying data stream, related data structure and processing mechanism, so ExtendedAtomicCoactor is only suggested for the expert.

5.3.1. AtomicCoactor

Based on the general role the actor plays in the workflow, there're three kinds of AtomicCoactor. The base class for each kind is:

- CollectionSource: It's an actor to convert external input into inside data stream of COMAD workflow. It's always the first actor in workflow. To deal with different external input, there're multiple subclasses of this actor. For example, CollectionComposer can receive input in text window, while CollectionReader can handle input in file. As the data stream composer of workflow, CollectionSource and its subclasses are very special. They have no read scope and data binding and only have one port to output the data stream.
- CollectionTransformer: It's an actor in the middle of the workflow. The data stream flows in and out through the input and output port. There're several call back methods inside the actor which is invoked at specific time, like when the input data is ready. By rewriting these methods in different way, actors with different data processing functions are developed.

- CollectionSink: It's an actor at the end of the workflow, with only one input port to let the data stream flow in. Except this, it's the same as the CollectionTransformer.

To develop your own source actor, firstly it's required to inherit the CollectionSource. And then you need to overwrite the fire method to parse external input and create the corresponding data stream through API of CollectionManager or AnnotationSet. As a manager of collection or annotation, the CollectionManager and AnnotationSet class provide creation, deletion and the other management functions for collection, data inside collection and annotation. Besides fire, there're five other methods you may overwrite: preinitialize(), initialize(), prefire(), postfire(), and wrapup(). These methods will be invoked by the system at different time point during the workflow execution to for initialization or wrap up etc. For more details about these methods, please refer to https://kepler-project.org/.

To develop your own actor other than source actor, you need to inherit from either CollectionTransformer or CollectionSink, define the signature, and then overwrite the methods as defined in table 6. Usually only the fireActor needs to be overwritten.

Table6. Callback Methods in Actor of CollectionTransformer or CollectionSink Kind

Method	Description
void fireActorAfterEnterScope()	It's invoked for initializing the actor status once the read scope is entered.
DataBindingValueMap fireActor(DataBindingValueMap inputDataMap)	It's invoked when actor is ready to be fired. The data processing logic is implemented here. The prepared input data is provided through the inputDataMap parameter, while the output is given back to system through the return value.
DataBindingValueMap fireActorBeforeLeaveScope()	It's invoked before the leave the read scope collection for wrap up. What needs to be written into data stream is given back to system through the return value.

Take StatisticCalculator actor for example to see generally how AtomicCoactor is developed. The source code of StatisticCalculator actor could be found at "<comad-exp_install_dir>/src/org/kepler/demo/simple/StatisticCalculator.java".

Firstly the actor inherits from CollectionTransformer.

```
public class StatisticCalculator extends CollectionTransformer
```

Secondly the signature of actor is defined in constructor. Actually the signature could be defined in any places as long as it's defined before the workflow is initialized with it which is done in the initialize method. Through createSignatureElementParameter method, one input and three output data ports are declared. For each port, the port name, signature definition and whether it's an input or output port are defined.

```
createSignatureElementParameter("valueList", "DoubleToken+", true);

createSignatureElementParameter("avg", "DoubleToken", false);
createSignatureElementParameter("max", "DoubleToken", false);
createSignatureElementParameter("min", "DoubleToken", false);
```

Finally, the data processing logic is implemented by overwriting the fireActor method. The input data prepared automatically by system is provided through inputDataMap parameter. It's actually a map of port name and value. If no data is bound at the port, the value is null. According to the signature definition, the value could be an object like StringToken, or could be a list of object like List<DoubleToken>. After making statistic with input data, the return object to keep output value for each port is constructed which will be written into the data stream automatically by system as defined by data binding. The result DataBindingValueMap object can also be used to declare specific dependency of the output data. Please refer to "Provenance Recording" for more details about dependency. For each port, a single object or a list object could be a valid output depending on the signature definition. For any none optional output port whose signature has cardinality of one or +, to output null or an empty list result in exception. If the port is optional and nothing is output, you can either put null as the output value or just don't put this port in the returned DataBindingValueMap object. If all the output ports are optional and nothing is output, you can directly return null.

```
//get input data
Object inputValueListData = inputDataMap.get("valueList");
List<?> inputValueList = (List<?>)inputValueListData;

//make statistic
....

//output
DataBindingValueMap outputData = new DataBindingValueMap();

outputData.put("avg", new DoubleToken(avg));
outputData.put("max", new DoubleToken(max));
outputData.put("min", new DoubleToken(min));

return outputData;
```

5.3.2. CompositeCoactor

CompositeCoactor is used to encapsulate a sub-workflow assembled from a group of kepler actors.

We take the StatisticCalculator actor in the Statistic_Composite workflow as an example to show how the CompositeCoactor is used. Statistic_Composite workflow could be found at "<*comad-exp_install_dir*>/workflow/demo/Simple/Statistic_Composite.xml".

Step1: Create CompositeCoactor and rename it as StatisticCalculator

1) Go to tool menu and click the "Instantiate Component" item to open the "Instantiate Component" window. In the text field for the "class name", type "org.kepler.coactors.CompositeCoactor" and click "ok".
2) Right click the icon and choose "Customize Name" to open the rename window. Type "StatisticCalculator" for both name and display name, then click "Commit" button.

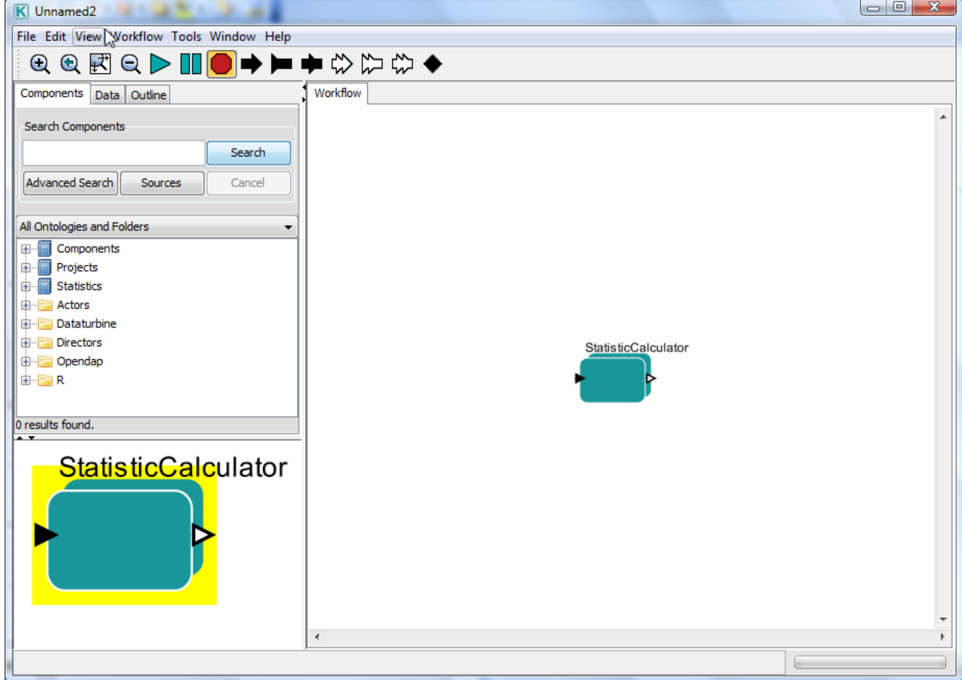

Figure 20. CompositeCoactor

Step2: Open the sub-workflow design panel and add SDF director. For more information about SDF director, please refer to https://kepler-project.org/.

1) Right click the actor icon and choose "Open Actor" from the pop-up menu
2) In the right side search field, type "sdf" and click search. In the result shown below, drag "sdf Director" to the left side workflow design panel. It's only allowed to use SDFDirector in CompositeCoactor. The ports with name "input" and "output" appearing on the canvas are actually the input and output port of the actor from where the data stream flows in and out. They're not data binding ports.

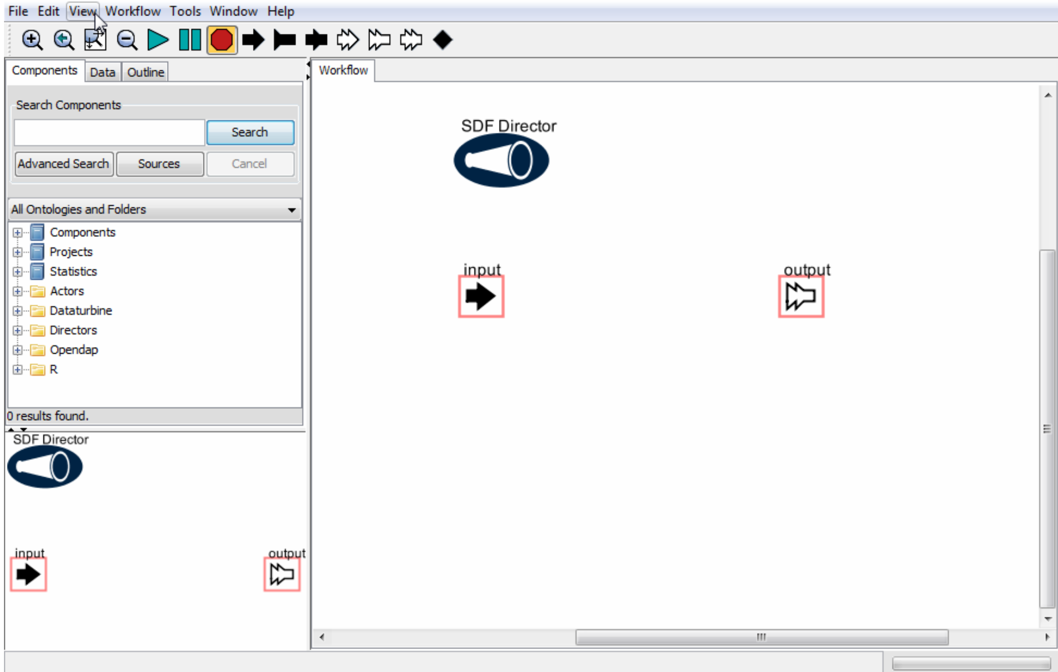

Figure21. Design panel of CompositeCoactor with SDFDirector

Step3: Add kepler actor, RExpression actor, configure it to implement statistic computation. For more information about RExpression actor, Please refer to https://kepler-project.org/.

1) Go to tool menu and click the “Instantiate Component” item to open the “Instantiate Component” window. In the text field for the “class name”, type “org.ecoinformatics.seek.R.RExpression” and click “ok”.
2) Add input and output ports for RExpression actor which will be used as variable in the R script configured later.
    a. Right click icon of RExpression actor and choose “Configure Ports” to open the port configuration window.
    b. To add “valueList” input port, click “Add” button, type “valueList” in the “name” column, check checkbox in “input” and “show name” column. Except check checkbox in “output” column, the output port of “average”, “max” and “min” could be created in the same way. Finally click “Commit” button to finish port configuration.
3) Implement function by writing R script
    a. Double click the RExpression icon to open the parameters configuration window
    b. In the text field for “R function or script”, paste the following script and then click “Commit” button to finish configuration. “valueList” represents the input data from valueList input port. “mean”, “max” and “min” are built-in function of R. By assigning value to variable with the same name as the output port, the value is actually output from that output.

```
average <- mean(valueList)
max <- max(valueList)
min <- min(valueList)
```

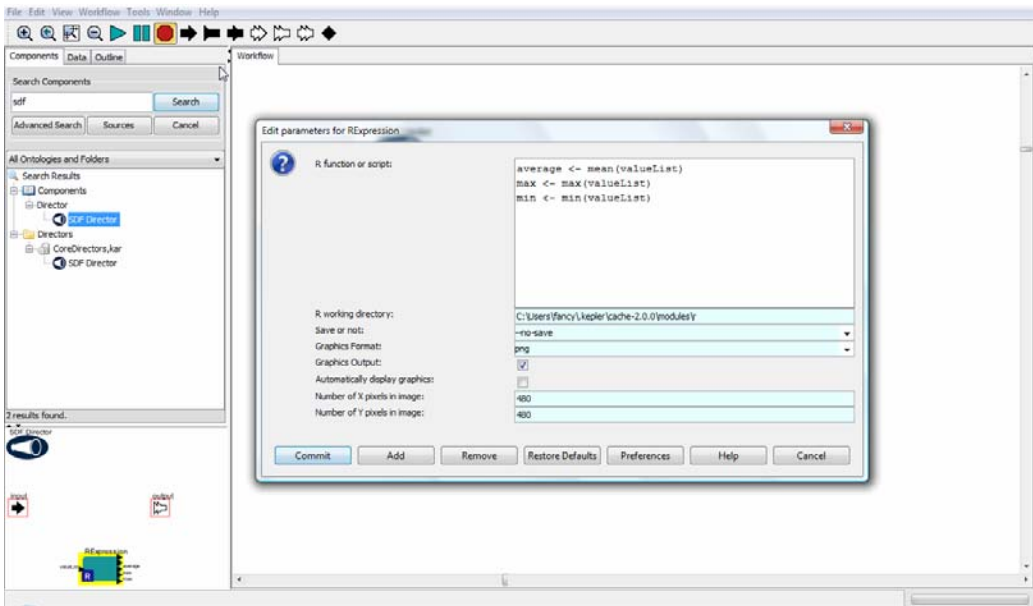

Figure22. RExpression Actor

Step4: Add input and output data binding port for the StatisticCalculator actor which server as the input and output port for the inside sub-workflow. The final nested workflow is demonstrated in Figure 23.

1) Go to tool bar and click “New input port” icon, both a new input port with blank triangle and a signature element parameter with blue circle for this port are added onto canvas. Drag them to proper location. The port can also be created through copy of an existing data binding port or through the port configuration window of this CompositeCoactor.
2) Connect the newly added input port with the “valueList” port of RExpression actor.
3) Double click the signature element parameter of the newly added port to open the parameter configure window. Type “DoubleToken+” in the text field and commit.
4) Go to the outside workflow design canvas and open the port configuration window of StatisticCalculator, change the name of the newly created port to “valueList” which makes the meaning of this port more clearly.
5) Go to tool bar and click “New output port” icon to add output data binding ports of avg, min and max in the same way.
6) The nested workflow is done and close the design panel.

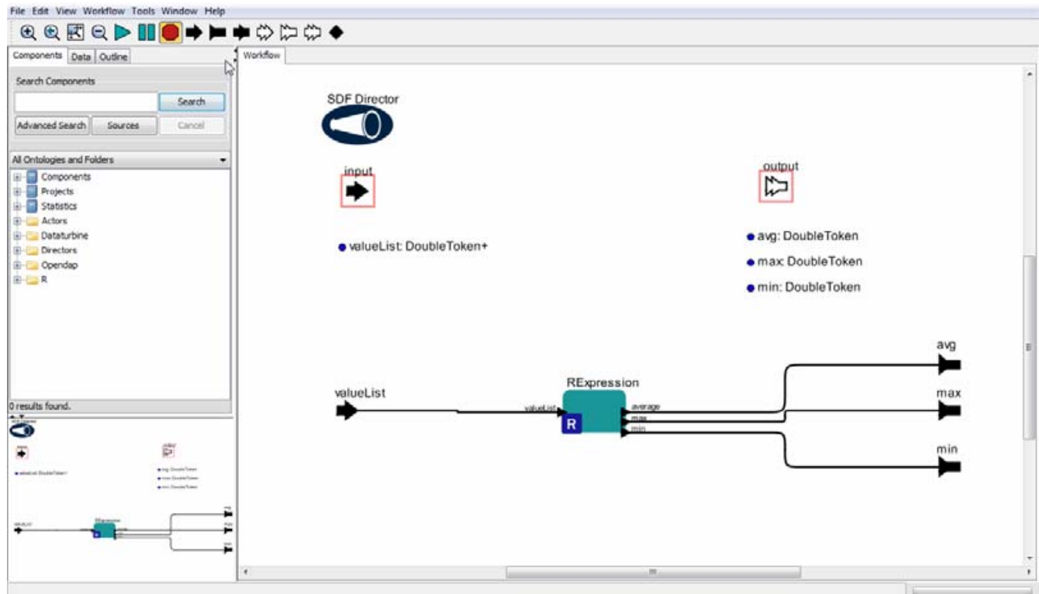

Figure23. Nested Workflow of Composite StatisticCalculator

Step 5. Design the top workflow as described in chapter 3. The final finished workflow is demonstrated in Figure 24. It has the same function as the Statistic workflow with the atomic StatisticCalculator actor.

1) In the top workflow design canvas, add COMAD director, add and configure the CollectionComposer and CollectionDisplay actor, connect all the actors in order.
2) Double click the StatisticCalculator icon and set the read scope and data binding path expression for each port.

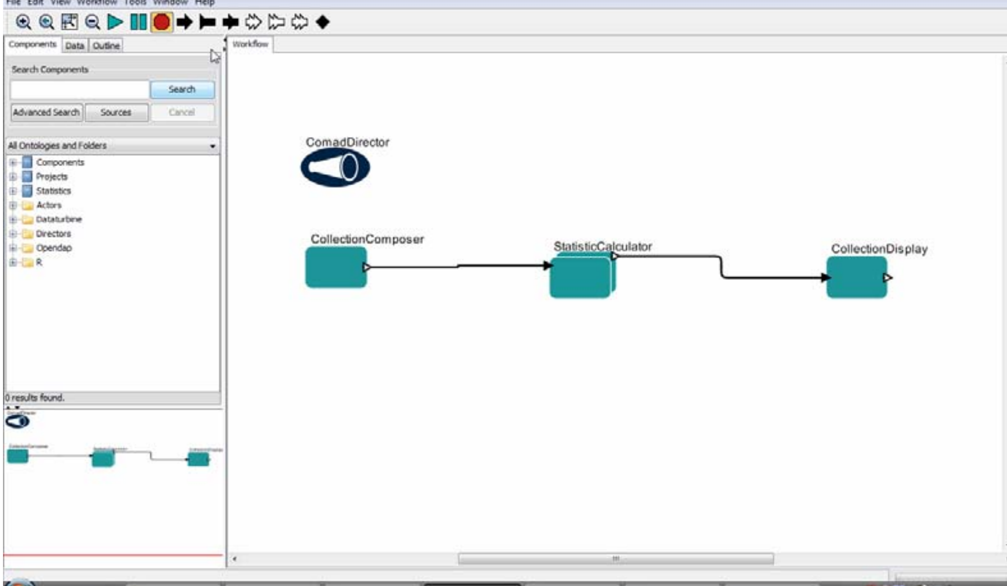

Figure24. StatisticCalculator_Composite Workflow

5.3.3. *Extended AtomicCoactor*

This kind of actor actually works on event-driven mode. When data stream flows through the actor, once a token arrives at actor an event is triggered. And then the corresponding

event handler method is invoked. In this way, the actor could access everything in the raw data stream and any fancy data manipulation could be implemented.

Through ExtendedCollectionTransformer or ExtendedCollectionSink, all the event handler methods as defined in Table 7 are inherited. And the actor can choose to overwrite some of them as needed. In each handler method, the corresponding token and the objects managing the related collection or annotations, like CollectionManager or AnnotationSet, are passed to the actor. Therefore the actor can buffer data or operate on Collections or Annotations freely. For more detailed API definition, please refer to the API documents.

Table7. COMAD Actor Event Handler

Handler Method Name	Description
handleToken	Invoked once any token arrives
handleScopeStart	Invoked once an OpeningDelimiterToken arrives and result in entering the read scope. The invocation initialization could be done here.
handleCollectionStart	Invoked once an OpeningDelimiterToken arrives. If this event invokes both handleScopeStart and handleCollectionStart, the handleScopeStart is invoked before handleCollectionStart.
handleAnnotation	Invoked once an AnnotationToken arrives
handleData	Invoked once a DataToken arrives
handleException	Invoked once a DataToken arrives
handleLoopTermination	Invoked once an ExceptionToken arrives
handleInvocationDependencyToken	Invoked once a InvocationDependencyToken arrives
handleCollectionEnd	Invoked once a ClosingDelimiterToken arrives
handleScopeEnd	Invoked once a ClosingDelimiterToken arrives and result in leaving the read scope. The invocation wrap up could be done here. If the arrival of ClosingDelimiterToken invoke both handleCollectionEnd and handleScopeEnd, then handleScopeEnd is invoked after handleCollectionEnd.

5.4. Provenance Recording

During COMAD workflow execution, the following provenance information is recorded therefore the data lineage could be fully tracked and the dependency between actors is clearly showed:

- Insertion: Whenever a token is created and inserted into data stream, it also contains the provenance information about how it's created, including invocation and dependency. Invocation declares the token is created in which invocation of which actor, while the dependency declares which data or annotation contributes this insertion operation.

- Deletion: Whenever a token is deleted, a DeletionToken is inserted into data stream to record this token is deleted in which invocation or which actor.
- Invocation dependency: The dependency between invocations is actually inferred from data dependency.
    - Once a token is inserted in invocation A while the insertion depends on another token inserted in invocation B, then A depends on B.
    - Once a token is deleted in invocation A while it's inserted in invocation B, then A depends on B.

For the normal AtomicCoactor and CompositeCoactor, the data dependency for each newly created data or annotation is generated automatically by system. There're three dependency modes for choose:

- Depend on invocation: Depend on all data or annotation used to fire actor ever since the read scope is entered. This is the default mode.
- Depend on firing: Depend on data or annotation used to fire actor only in the same firing in which the token is inserted.
- Depend on specified elements: Depend on the specified list of data or annotation used to fire actor.

The AtomicCoactor could set the dependency mode through the DataBindingValueMap object returned in fireActor or fireActorBeforeLeaveScope method. In Statistic workflow, if the read scope of StatisticCalculator is root collection, the vlaueList port gets a list of humidity data from each station through "/station//DoubleToken[@label=="humidity"]+". The actor is invoked once while fired twice as demonstrated in Figure 16. Each time when the actor is fired, a list of data from one station is input and the statistic value of this station is output. To make the statistic value only depend on the input data of the same station instead of input data from all stations, the latter two dependency modes could be used.

```
//depend on firing

//output
DataBindingValueMap outputData = new DataBindingValueMap();

outputData.put("avg", new DoubleToken(avg));
outputData.put("max", new DoubleToken(max));
outputData.put("min", new DoubleToken(min));

//set dependency of input elements in each firing
outputData.setDependOnlyOnCurrentFiring();

return outputData;
```

```
//depend on specified elements

//get input data
List<?> inputValueList =
(List<?>)inputDataMap.get("valueList");
DependingObjectSet dependingObjects = new DependingObjectSet();

//make statistic
for(int i=0; i<inputValueList.size(); i++){
      dependingObjects.add(inputValueList.get(i));
      ...
}

//output
DataBindingValueMap outputData = new DataBindingValueMap();
...
outputData.setDependingObjectSet(dependingObjects);

return outputData;
```

The CompositeCoactor only support the former two dependency modes. To use the second mode, the user needs to check the "Depend on Firing" checkbox in its parameter configuration window. The extended AtomicCoactor needs to create the dependency set according to what's really depended by calling the corresponding API.

The trace file recording all the provenance information of one workflow execution could be generated by putting TraceWriter actor at the end of the workflow. Then it could be parsed and visualized by "Provenance Brower" which is an interactive tool for visualizing and querying data dependency (lineage) graphs produced by scientific workflow. The browser allows users to explore different views of provenance as well as to express complex and recursive graph queries through a high-level query language (QLP). By combining provenance visualization, navigation, and query, the provenance browser can enable scientists to more easily access and explore scientific workflow provenance information. The data dependency view and collection structure and invocation dependency view provided by provenance browser is demonstrated in Figure 25 and Figure 26.

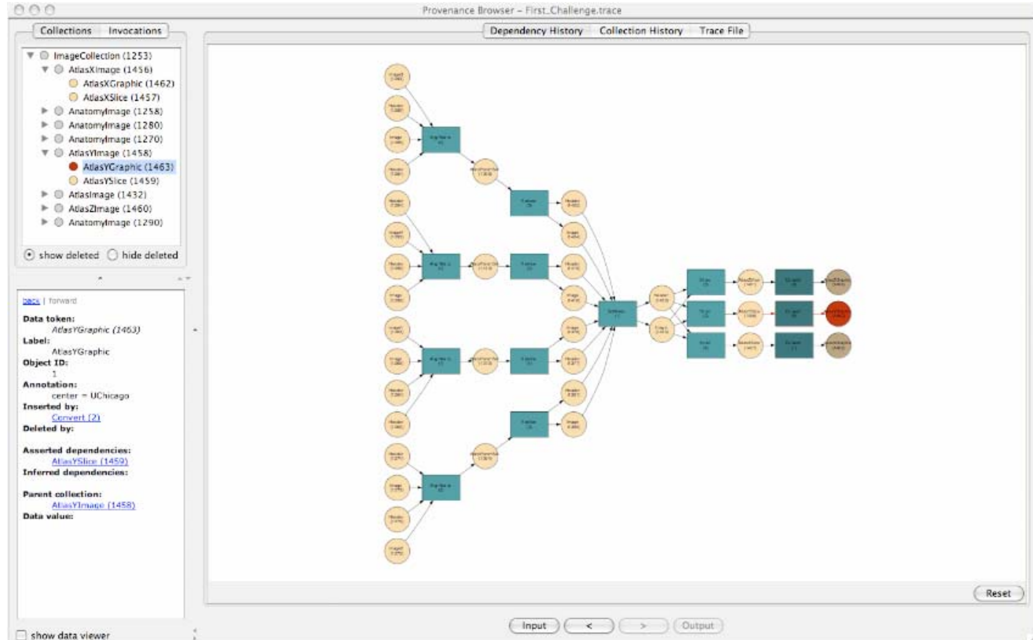

Figure25. Data Dependency View

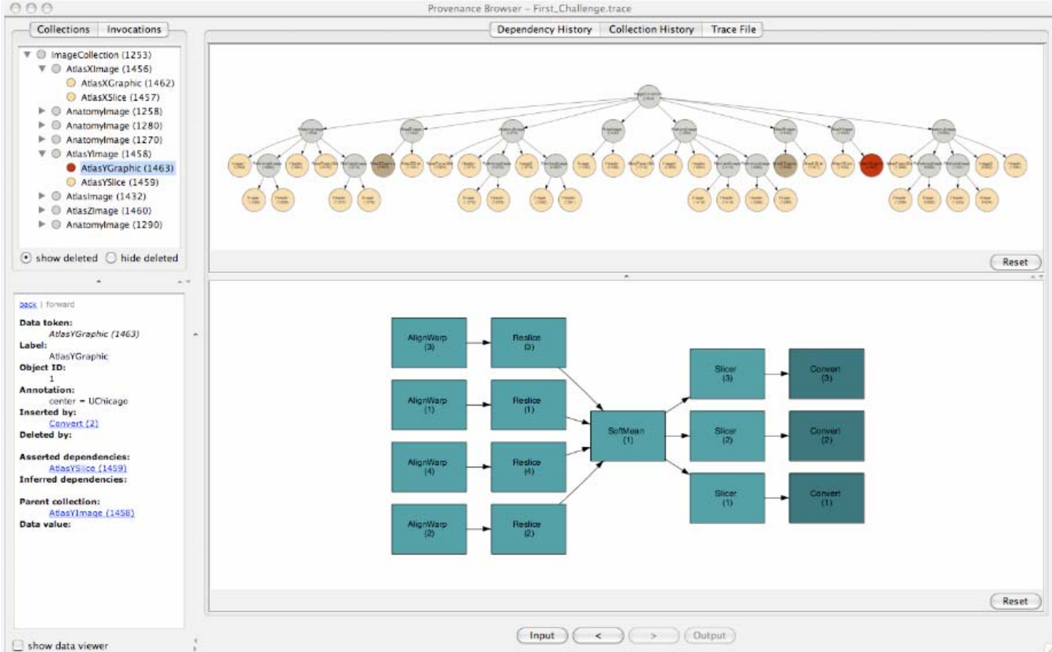

Figure26. Collection Structure and Invocation Dependency View

6. COMAD Path Expression Syntax

This chapter elaborates the syntax of COMAD path expression used to declare the read scope, signature and data binding, including path element, qualifier, cardinality, decoration and port reference

6.1. Path Element

The collection, data or annotation is selected through the path expression. In the path expression, collection is denoted by its label. Data is denoted by its type. Annotation is denoted by its key. Wildcard of "*" could represent any label, any type or any key depending on the context.

Expression	Description
/	Match items that are children of the context item. The context of the read scope is the root collection. The context of the data binding is the read scope. The single "/" of the read scope matches the root collection.
//	Match items nested in the context item
collection_label	Match the collection with the specified label
data_type	Match the data with type compatible to the specified type. Please refer to the chapter of "Type System" about supported type and their compatible relations.
*	Match all collections or data. Which one it denotes to depends on its path expression context.
@annotation_key	Match annotation with the specified key
*@**	Match all annotations

ReadScope expression example:

Expression	Description
/ /*	Match the root collection
/Project	Match the root Project collection
/Project/Nexus	Match all the Nexus collections as child of the root collection with label of Project
//GeneSeq	Match all the GeneSeq collections
/ Project// GeneSeq	Match all the GeneSeq collections as descendant of the root Project collection.
//Project// GeneSeq	Match all the GeneSeq collections as descendant of the Project collection.
/Project/* /Project//*	Match all the collections as child of the root Project collection. These two expressions are equal since the read scope is not matched in a nested way.

Data binding expression example:

Expression	Description
/@cutoff	Match all the cutoff annotations of the read scope collection. Since the annotation with specific key is unique, so actually at most one annotation will be matched here.
//@cutoff	Match all cutoff annotations for the read scope collection and all its descendant collections and data.

	Notice: The path element with annotation behind double slash only matches the annotation uniquely. The duplicated cutoff annotation due to annotation inheritance won't be matched repeatedly. Once the annotation is overwritten, even with the same value, it becomes a different annotation and then will be matched. If read scope collection has one cutoff annotation, and it's not overwritten, then only one cutoff annotation is matched. If cutoff is overwritten for n times, plus the original one, n+1 annotations are matched.
/@*	Match all the annotations of the read scope collection
/Nexus/@cutoff	Match the cutoff annotation of the Nexus collection who is a child of the read scope collection
//Nexus//@cutoff	Match all the cutoff annotations uniquely of each Nexus collection and all its descendant collection and data while Nexus collection is the descendant of the read scope collection
//StringToken/@cutoff	Match cutoff annotation of each data with type compatible to StringToken inside read scope collection

6.2. Qualifier

Qualifier declares qualification for the item to be matched. The qualifier is a Boolean expression consisting of constant, variable, function and operator. Qualifier could be applied to the path element denoting to the collection, data or annotation.

The following constants are supported:

Constant	description	example
PI, pi, E, e, true, false		
String constant	Anything between quotes	"hello","10"
Integers constant	Numerical values without decimal points	10 , -3
Double constant	Numerical values with decimal points	10.0 , 3.14159
Long integer constant	Integers followed by the character l (el) or L	10L, -3L
Complex constant	A complex is defined by appending an "i" or	1+2i , 3+4j

	a "j" to a double for the imaginary part.	
Array constant	Arrays are specified with curly brackets	{1, 2, 3} {"x", "y", "z"} {1, 2.3} {2*pi, 3*pi} {{1, 2}, {3, 4, 5}}
Record constant	Records are delimited by curly braces, with each element given a name.	{length=1, name="foo"} {value={1,2}, name="foo"} {value={width=1,length=2}, name="foo"}

The following variables are supported:

Variable name	description
@label	Return the label of the element this qualifier is applied to, which might be a collection or data.
@type	Return the type of the element this qualifier is applied to, which might be a data or annotation.
@value	Return the value of the element this qualifier is applied to, which might be a data or annotation. This variable should be used carefully. If the value is expected to be a scalar in the qualifier expression, then any value with type other than scalar, like string, will result in an expression evaluation exception. This variable can't be used for the data with user defined type.
@annotation_key	Return the value of the annotation with the specified key. This variable should be used carefully. If the value is expected to be a scalar in the qualifier expression, then any value with type other than scalar, like string, will result in an expression evaluation exception. This variable can't be used for the data with user defined type.

The following functions are supported:

function	Argument type(s)	Return type	description
exist	*@annotation_key*	boolean	Test whether the specified annotation exists.
type	*@annotation_key*	string	Return the type of the specified annotation.
abs	double or complex	double or int or long (complex returns double)	absolute value complex case: abs(a + ib) = |z| = sqrt(a^2 + b^2)
ceil	double	double	ceiling function, which returns the smallest (closest to negative infinity) double value that is not less than the argument and is an integer.
floor	double	double	floor function, which is the largest

			(closest to positive infinity) value not greater than the argument that is an integer.
max	double, double or {double}	a scalar of the same type as the arguments	maximum
min	double, double or {double}	a scalar of the same type as the arguments	minimum
pow	double, double	double	first argument to the power of the second
round	double	long	round to the nearest long, choosing the next greater integer when exactly in between. If the argument is out of range, the result is either MaxLong or MinLong, depending on the sign.

The following operators are supported:

Operator Class	Operator Expression	Description
Logical Operator	!	not
	||	or
	&&	and
Relational Operator	==	Equal
	!=	Not equal
	>	Greater than
	>=	Greater than or equal to
	<	Less than
	<=	Less than or equal to
Arithmetic Operator	+	addition
	-	subtraction
	*	multiplication
	/	division
	%	Modulus (division remainder)

Qualifier example:

Expression	Description
/DoubleToken[@label=="temperature"]	Match the DoubleToken data with the label of temperature.
/*[@label=="temperature"]	Match collection or data with label of temperature.
/DoubleToken [@value==10.5]	Match the DoubleToken data with the value of 10.5
/@cutoff[@value>1]	Match the cutoff annotation with the value greater than 1.
/Station[@name=="station1"]	Match the Station collection with the name annotation and the value of this annotation

<table>
<tr><td></td><td>is station1</td></tr>
<tr><td>/StringToken
/*[@type=="StringToken"]</td><td>Match data with type compatible to StringToken.</td></tr>
<tr><td>/@name[@type=="StringToken"]</td><td>Match the name annotation with type compatible to StringToken.</td></tr>
<tr><td>/Station[exist(@name)]</td><td>Match the Station collection with name annotation</td></tr>
<tr><td>/StringToken[exist(@*)]</td><td>Match the StringToken data with any annotation. RunAnnotaiton, including Coactor annotations and its parameter annotations, are not counted here.</td></tr>
<tr><td>/Station[type(@name) =="StringToken"]</td><td>Match the Station collection with the name annotation and the type of this annotation is compatible to StringToken.</td></tr>
<tr><td>/DoubleToken[@label!="temperature"]</td><td>Match the data with type compatible to DoubleToken and with no label or with label other than temperature.</td></tr>
<tr><td>/Station[!exist(@name)]</td><td>Match the Station collection with no annotation or with annotation other than name.</td></tr>
<tr><td>/Station[exist(@cutoff) ||
abs(@temperature+10)>1]</td><td>Match the Station collection when it has cutoff annotation or the absolute value of temperature annotation after adding 10 is greater than 1</td></tr>
</table>

6.3. Cardinality

Cardinality is used for the data or annotation to express the requirement on the number of the input or output data of the actor. The following cardinality is supported:

Cardinality	Description		
	signature	Input port data binding	output port data binding
One (default)	One data is expected to be input or output	In each firing, this port must provide one bound data or annotation as the input	In each firing, this port must write one output data or annotation into data stream.
?	Zero or one data is expected to be input or output	In each firing, this port can either provide one bound data or annotation as input or provide nothing.	In each firing, this port can either write one output data or annotation into data stream or write nothing.
*	Zero or multiple data is expected to be input or output	In each firing, this port can either provide one none empty list of bound data or annotation as	In each firing, this port can either write one none empty list of output data or

		input or provide nothing.	annotation into data stream or write nothing.
+	Multiple data is expected to be input or output	In each firing, this port must provide one none empty list of bound data or annotation as input.	In each firing, this port must write one none empty list of output data or annotation into data stream.

Cardinality can be used in the signature and data binding. When it's in the data binding, it can only be used for the last element in the data binding path to declare the number of data or annotation to bind.

When cardinality of * or + is used in the data binding, it means multiple data or annotation will be accumulated and fed into the actor as a list. The boundary of such accumulation is the collection or data matched to the second to last element of the data binding path. If the path only has one element, then the boundary is the read scope collection. For example in Statistic workflow, if the read scope of StaticCalculator actor is "/HumidityDataCollection" and the data binding of valueList port is "/station//DoubleToken[@label=="humidity"]+", then the humidity data will be grouped by the station. But if the read scope is "/HumidityDataCollection" and the data binding is "//DoubleToken[@label=="humidity"]+", then the humidity data from all stations will be accumulated together into one big list.

6.4. Decoration

Decoration is used before the element to declare special operation.

Decoration is declared as a switch inside a pair of curly brackets, like {-op}. Multiple decoration could be declared as {-op1 -op2}. The order of the multiple decorations for one element doesn't matter.

Three kinds of decoration are supported:

- Life management of the element, including creation and deletion
- Exact type match requirement: By default the type compatible to the target type is treated as a match type.
- Split of the data in structural type

6.4.1. Life management

This decoration is applied to collection, data and annotation to manage their lives, including creation and deletion.

6.4.1.1. Creation

The collection, data and annotation can only be created when an actor outputs new value. Therefore such decoration can only appear in the output data binding. The new output data and annotation will be created automatically. But a decoration must be used to declare a collection needs to be created.

There're three decorations to create collection at different time or at specific location:

- {-i} or {-invocation}: create the collection in each invocation.
- {-f} or {-firing}: create the collection in each firing
- {-p} or {-peer}:
    - create the collection as the peer of the read scope collection. By default such creation happens once in each invocation. So {-p} equals to {-p -i} by default. But if use the combination of {-p -f}, such creation happens once in each firing.
    - create the data as the peer of the read scope collection whenever it needs to be created.

The rules to use and parse the creation decoration:

- For a collection without any creation decoration, the system will try to find the matched collection that already exists and then write the data or annotation into it. If there's no matched collection, nothing will be written into the data stream.
- The creation only happens when the output data binding port has an output data or annotation.
    - If the actor is invoked while the port has no output in the whole invocation, then no path will be created in this invocation even if there's {-i} or {-p} or {-p -i}.
    - If the actor is fired while the port has no output in this firing, then no path will be created in this firing even if there's {-f} or {-p -f}
- Multiple occurrence of {-i} and {-f}
    - Multiple occurrence of {-i} equals to only keep the first occurrence of the {-i}
    - Multiple occurrence of {-f} equals to only keep the first occurrence of the {-f}
    - If both {-i} and {-f} occurs, {-f} can only appears after the element decorated by {-i}.
    - The output binding expression with {-i} and {-f} could be divided into three segments.
        - The first segment is the sub-path before the first occurrence of the {-i}. This sub-path will match to the existed collection.
        - The second segment is the sub-path between the first occurrence of the {-i} and {-f}. This sub-path will be created one time for each invocation.
        - The third segment is the sub-path after the first occurrence of the {-f}. This sub-path will be created in each firing.
- Rule for {-p}
    - {-p} can only decorate the first element in the path. Then the whole path will be created as a peer of the read scope collection. The {-i} or {-f} inside the path keep the same operation semantics.
    - {-p} is disallowed to be used when the read scope collection is the root collection, since it's impossible to create a collection as a peer of the root

collection. All the collection, data or annotation must be inside the root collection.
    - {-p} is disallowed to decorate the annotation since the annotation can't be a peer of the read scope collection.
- The relative path delimiter "//" is forbidden to be used for the newly created elements, including collection, data and annotation
    - The "//" is forbidden for the element decorated by {-i}, {-f} or {-p}
    - The "//" is forbidden to appear after the element decorated by {-i}, {-f} or {-p}

Example of the creation decoration in the output data binding:

Expression	Valid	Description
/Station/DoubleToken	yes	Create a DoubleToken data in each firing as a child of any matched Station collection
/{-i}Station/DoubleToken	yes	Create a Station collection as a child of the read scope collection in each invocation. And then create a DoubleToken data in each firing as a child of this newly created Station collection.
/{-i}Station/{-i}Month/DoubleToken /{-i}Station/Month/DoubleToken	yes	Create a subtree of /Station/Month as a child of the read scope collection in each invocation. And then create a DoubleToken data in each firing as a child of this newly created Month collection.
/{-f}Station/DoubleToken	yes	Create a Station collection as a child of the read scope collection in each firing. And then create a DoubleToken data in each firing as a child of this newly created Station collection.
/{-p}StringToken	yes	Create the StringToken as a peer of the read scope collection in the data stream
/{-p}Station/DoubleToken /{-p -i}Station/DoubleToken	yes	Create the Station collection as a peer of the read scope collection in each invocation. And then create the DoubleToken data in each firing as a child of this newly created Station collection.
/{-p -f}Station/DoubleToken	yes	Create the Station collection as a peer of the read scope collection in each firing. And then create the DoubleToken data in each firing as a child of this newly created Station collection.

/{-p}Temperature/{-i}Station/{-f}Month/DoubleToken	yes	Create the Temperature collection as a peer of the read scope collection in each invocation. Create the Station collection as a child of the newly created Temperature collection in each invocation. Create the Month collection as a child of the newly created Station collection in each firing. Create the DoubleToken data in each firing as a child of this newly created Station collection
/{-i}StringToken /{-f}StringToken /{-i}@cutoff /{-f}}@cutoff	no	Data and annotation is created automatically when it's output. {-i} and {-f} can only decorate the collection needs to be created.
//{-p}StringToken //{-i}Station/StringToken /{-f}Station//Month/DoubleToken	no	relative path delimiter "//" is forbidden to be used at the element or after the element decorated by {-i}, {-f} or {-p}
/{-f}Station/{-i}Month/DoubleToken	no	{-f} is disallowed to appear before the {-i}
/Station/{-p}Month/DoubleToken	no	{-p} can only decorate the first element
/{-p}@cutoff	no	{-p} can't decorate the annotation

6.4.1.2. Deletion

This decoration is used to delete the collection, data or annotation from the data stream.

The element with such decoration is visible to the actor where it's deleted. But once it leaves the actor, the deletion takes effect. It's invisible to the successive actor generally except in some special cases when the deleted token needs to be visible.

When the collection is deleted, the collection, its annotations and all its descendants, including collection, data and annotation, are all deleted. When the data is deleted, the data and its annotations are deleted. When the annotation is deleted, the annotation itself is deleted.

Deletion decoration is declared as {-d} or {-delete}. The rules to use and parse this decoration are:

- {-d} can only be used in the read scope or input data binding.
    - For read scope: {-d} is only allowed to decorate the read scope collection. So it's only allowed to appear in front of the last element of the read scope expression.
    - For input data binding: {-d} is allowed to be in front of any element in the path expression.

- If it tends to output new element inside the deleted collection or as an annotation to a deleted collection or data, then an exception will be thrown out.
- If the path in the data binding with delete decoration is matched once the read scope collection is entered, the deletion will take effect even if the actor is not fired since the input data is not ready.

Example of deletion decoration in read scope:

Expression	Valid	Description
/Station//{-d}Month	Yes	Delete all the Month collections as descendants of the Station collection
/{-d}Station//Month	No	It's disallowed to delete the collection other than the read scope collection in the read scope

Example of deletion decoration in input data binding:

Expression	Valid	Description
/Station//{-d}StringToken	Yes	Delete the StringToken data as descendants of the Station collection.
/{-d}Station//StringToken	Yes	Delete the Station collection that is an ancestor of the StringToken.
/{-d}@cutoff[@type="DoubleToken"]	Yes	Delete the cutoff annotation of the read scope collection with type of DoubleToken

6.4.2. Exact type match

The match based on the type is "compatible type match" instead of "exact type match" by default. When the StringToken is expected, the token with type compatible to StringToken is matched. For the definition of the "compatible type", please refer to the type system chapter. To support declaration of match to a type exactly, this decoration is introduced.

The exact type match decoration is declared as {-e} or {-exact}. The rules to use and parse this decoration are:

- {-e} is used just before the type declaration, including type constant, type variable or type function.
- Since {-e} is used for the element match and selection, this decoration can only be used in the read scope, input data binding, input signature, while can't be used for the output data binding and output signature.

Example of exact type match decoration:

Expression	Description
/StringToken	Match all the data with type compatible to StringToken, might includes IntegerToken, DoubleToken etc.
/{-e}StringToken	Match all the data with the exact type of

	StringToken.
/@name[{-e}@type=="StringToken"]	Match the name annotation whose type is exactly StringToken.
/StringToken[{-e}type(@name) =="StringToken"]	Match all the data with type compatible to StringToken and has name annotation whose type is exactly StringToken.

6.4.3. Split of structural type

ArrayToken and RecordToken are two structural types supported in COMAD. Following the COMAD design idea, if the inside element, instead of the whole data, of such structural type will be consumed by the actor individually, the data should be tore open with each data representing each element. In this way, the actor doesn't need to tear open and wrap up the data repeatedly. It's easier for the actor to pick up the interested data. Therefore, split decoration is introduced for such requirement. Obviously, such split is not needed while the data in such structural type is consumed by the actor as a whole. In this case, the use of the atomic data in structural type instead of a collection, is more neat and efficient.

Split operation is only available to the output data. And the split is only tear open the top level while not handling the deeply nested level. The way to declare split requirement is different for the type of RecordToken and ArrayToken.

- RecordToken
  - Split decoration is declared as {-s} or {-split} before the output data. It's not used for the collection or annotation.
  - {-s} can't be used together with cardinality of "*" or "+"
- ArrayToken
  - The output data in type of ArrayToken is split when the output data binding is declared as the element type plus cardinality of * or +.

Example of the split operation:

Output data	Output data binding	Description
{name="Jim", value=1}	/RecordToken (name=StringToken, value=IntegerToken)	One RecordToken data with value of {name="Jim",value=1} is written into the data stream.
	/{-s}RecordToken (name=StringToken, value=IntegerToken)	Two data are written into the data stream. One is the StringToken data with label of name and value of Jim. Another is the IntegerToken data with label of value and value of 1.
{name="Jim", value={1,2}}	/{-s}RecordToken (name=StringToken, value=ArrayToken(IntegerToken))	Two data are written into the data stream. One is the StringToken data with label of

		name and value of Jim. Another is the ArrayToken data with label of value and value of {1,2}.
{1,2}	/ArrayToken(IntegerToken)	One ArrayToken data with value of {1,2} is written into the data stream.
	/IntegerToken+	Two IntegerToken data are written into the data stream. One is the 1 and another is 2.

6.5. Port Reference

By default, the output data will be written at all the places matched to the data binding expression inside the read scope collection. The port reference is used to declare the output data will only be written into the specific location related to some input or output port in the same firing.

The port reference is only used in the output data binding. Such data binding expression starts from port reference expression with the normal path expression followed. The port reference expression is defined as: #*referenced_port*[*path_element_idx*] :

- *referenced_port*: It's the name of the referenced port. It could be any input or output port of the same actor. The system will parse the port in order according to such dependency. But once the circle reference is found, an exception will be thrown out.
- *path_element_idx*: It's the element index of data binding expression of the referenced port. It means the sub path of the referenced data binding expression from the beginning to the specified index will be used as the starting point to search for the writing place of this output data. The specified sub-path might change in each firing since it's possible to match different element in different firing. If the whole path will be referenced, the *path_element_idx* could be omitted.

The output data binding with reference to other data binding can also be referenced by another output data binding.

If the output data binding is declared as none optional through its cardinality, then its referenced data binding should also have the cardinality with the none optional semantics.

If the output data binding reference a path in an input data binding and the referenced path will be deleted since the decoration {-d} is used, an exception will be thrown out because it's meaningless to output anything under a path will be deleted.

When an output annotation references a data, the rules to write the annotation into the data stream in case of various combinations of the cardinalities are:

Cardinality of the output	Cardinality of the	rule

annotation	referenced output data	
default cardinality or ?	default cardinality or ?	The one output annotation will be written as the annotation of the one output data in the same firing.
default cardinality or ?	* or +	The one output annotation will be written as the annotation of each of the output data in the same firing
* or +	default cardinality or ?	The multiple output annotations will be written as the annotations for the one output data in the same firing. Since the key of the annotation for the same data or collection is not allowed to be the same. An automatically increased number will be append to the key.
* or +	* or +	The pairing semantics is implemented in the case. The multiple output annotations will be paired to the multiple output data. So it requires the number of the output data and annotations must be the same.

Example of the port reference:

Data Binding	Description
Input1: /Station/DoubleToken **Output1**: #Input1/@name	Output a name annotation to the input DoubleToken in the same firing
Input1: /Station/DoubleToken **Output1**: #Input1[0]/StationOne/{-i}Month/DoubleToken	The output DoubleToken is written under the Station collection where the input StringToken comes from in the same firing. Firstly a Month collection will be created in each invocation as a child of each existed StationOne collection which is child of the Station collection. And the DoubleToken will be written inside this collection as its child.
Input1: /Station/DoubleToken **Output1**: #Input1/@month	Output month annotation for the input DoubleToken in the same firing
Input1: /Station/DoubleToken **Output1**: #Input1/@month+	Output multiple month annotations to the input DoubleToken in the same firing. The key of the

	annotation has a automatically increased number in the end, like month, month1, month2...
Input1: /Station/DoubleToken+ **Output1**: #Input1/@month	Output month annotation for each input DoubleToken in the same firing
Input1: /Station/DoubleToken+ **Output1**: #Input1/@month+	Pairing the output month annotation for each input DoubleToken in the same firing. The number of the output month annotation and the number of the input DoubleToken should be the same.
Input1: /Station/DoubleToken **Output1**: #Output2/@name **Output2**: #Input1[0]/StationOne/{-i}Month/DoubleToken	The Output2 outputs DoubleToken under the Station collection where the input StringToken comes from in the same firing. Firstly a Month collection will be created in each invocation as a child of each existed StationOne collection which is child of the Station collection. And the DoubleToken will be written inside this collection as its child. The Output1 outputs the name annotation for the DoubleToken output by Output2.

7. Type System

Each data or annotation token in COMAD stream encapsulates a data object with specific type. There're several built-in types in COMAD. User-defined type is also supported. The type lattice defines the compatible relations between different types, while the type match is defined based on such relations. Usually one type is matched to another, when this type is compatible to it. The match of type determines whether the data or annotation token is bound. It also determines whether signature, data binding and the input/output data are consistent with each other. In some cases, the input/output data needs to be converted to be of the expected type.

7.1. Type Category

7.1.1. System Built-in Type

The COMAD system built-in type is defined in Table 7.

Table7. COMAD Built-in Type

Type Name	Example Data Value	Description
Object or *	-	Base type of all types
Token	-	Base type of all token types
DoubleToken	1.233e03, -3.34, 3.14159	The object of this type encapsulates double value
IntegerToken	833, 34, 56	The object of this type encapsulates integer value
LongToken	43L, -345443211, 43343332233	The object of this type encapsulates long value
ComplexToken	1+3i, 22.1i, 22.1+0i	The object of this type encapsulates complex value

ScalarToken	-	Base type for all numeric token types and BolleanToken
StringToken	"Greetings!"	The object of this type encapsulates string value
BooleanToken	true, false	The object of this type encapsulates boolean value
ArrayToken(*element_type*) e.g. ArrayToken(IntegerToken) ArrayToken(StringToken)	{1,2,3}, {"Tom","Jack"}	The object of this type encapsulates an array with element type as defined in the type name.
RecordToken(element_name=elemen_type, ….) e.g. RecordToken(name=StringToken, age=IntegerToken)	{name="Jack", age=30}	The object of this type encapsulates a record with element name and type as defined in the type name.
DomainObject	-	Base type of all user defined types

Token is a structural type encapsulating concrete data value. For example, "Hello", a string of alpha-numeric characters, is encapsulated as a string token. While 3, an integer is encapsulated as an integer token. For more details, please refer to Kepler documents.

RecordToken and ArrayToken are structural types encapsulating data of the other token types. The concrete RecordToken and ArrayToken type are decided by the inside element name and types. The element name is a string of alpha-numeric characters. The element type could be any built-in token type, even a type of RecordToken or ArrayToken.

In the declaration of the workflow input, if the data or annotation token is declared without type, then its type is inferred automatically from the value. Otherwise, the data object encapsulated in the token is created as an object with specified type. During this process, an exception will be thrown out if the type of data value is actually not as specified. For a data or annotation token without type, if its value is a quoted string of characters, it's parsed into a StringToken. But once a data or annotation token declares its type as StringToken, the pair of outside quotes is not needed anymore.

The following is some examples to declare data or annotation token with various system built-in types.

```
//StringToken
<Annotation key="station_number">"s2"</Annotation>
<Annotation key="station_number" type ="StringToken">s2</Annotation>
```

```
//DoubleToken
<Data>29.700001</Data>
<Data type="DoubleToken">29.700001</Data>

//ArrayToken
<Data>{1,2,3}</Data>
<Data>{"1","2","3"}</Data>
<Data type="ArrayToken(ScalarToken)">{1, 24.43, -3.2, true}</Data>

//RecordToken
<Data>{name="Jack", age=30}</Data>
<Data type="RecordToken(name=StringToken, gae=IntegerToken)">
        {name="Jack", age=30}
</Data>
<Data{name="arrayOne", value={1,2,3}}</Data>
<Data type="RecordToken(name=StringToken, value=ArrayToken(IntegerToken))">
        {name="arrayOne", value={1,2,3}}
</Data>

//Invalid Declaration
<Data type="IntegerToken">29.700001</Data>
<Data type="DoubleToken">hello</Data>
```

Sometimes the type of ArrayToken or RecordToken is too long to be used conveniently. Sometimes, the user expects to use a domain-specific name for the type to make its meaning clearer. The type alias could be used in these cases. Any built-in type except Object and DomainObject, could be assigned with an alias which is registered through “TypeSystem” component of the workflow. After it’s registered, the type alias could be used everywhere as the same as the normal type. The use of type alias should be careful to avoid side-effect, because once a type alias is defined, all the appearance of the original type will be replaced by the alias.

To demonstrate how type alias is defined and used, a complex RecordToken is used in Statistic workflow to encapsulate all the data inside each station collection. It has two elements. The timestamps and humidity elements contain timestamp and humidity value separately of all collection points in the same order. Both of them are an array of DoubleToken. For example, the s2 station is changed to:

```
<Collection label="station">
        <Data>
            { timestamps={1.196499599E9,1.196503199E9,1.196510399E9},
              humidity={29.700001,28.799999,29.200001}}
        </Data>
</Collection>
```

The complete type definition of this data is RecordToken(timestamps=ArrayToken(DoubleToken), humidity=ArrayToken(DoubleToken)). Without type alias, this long and complex string needs to be used repeatedly in both data binding and signature. To make it simple to be

referenced and make its meaning clearer, CollectonPoints is registered as its type alias in "TypeSystem" component with the following steps:

1. Go to tool menu and click the "Instantiate Attribute" item to open the "Instantiate Attribute" window. In the text field for the "class name", type "org.kepler.types.TypeSystem" and click "ok".
2. Double click TypeSystem icon to open parameter configuration window and click "Add" button to add "CollectionPoints" type alias definition in the opened window:
   1) Type "CollectionPoints" in the "name" field
   2) Type "RecordToken(timestamps=ArrayToken(DoubleToken), humidity=ArrayToken(DoubleToken))" for the "default value" field
   3) Click "OK" and finally commit the configuration

The added TypeSystem component and the registry for type alias of CollectionPoints are demonstrated in Figure 27. While the usage of CollectionPoints in data binding and signature is shown in Figure 28. The complete workflow could be found at "<*comad-exp_install_dir*>/workflow/demo/Simple/Statistic_TypeAlias.xml".

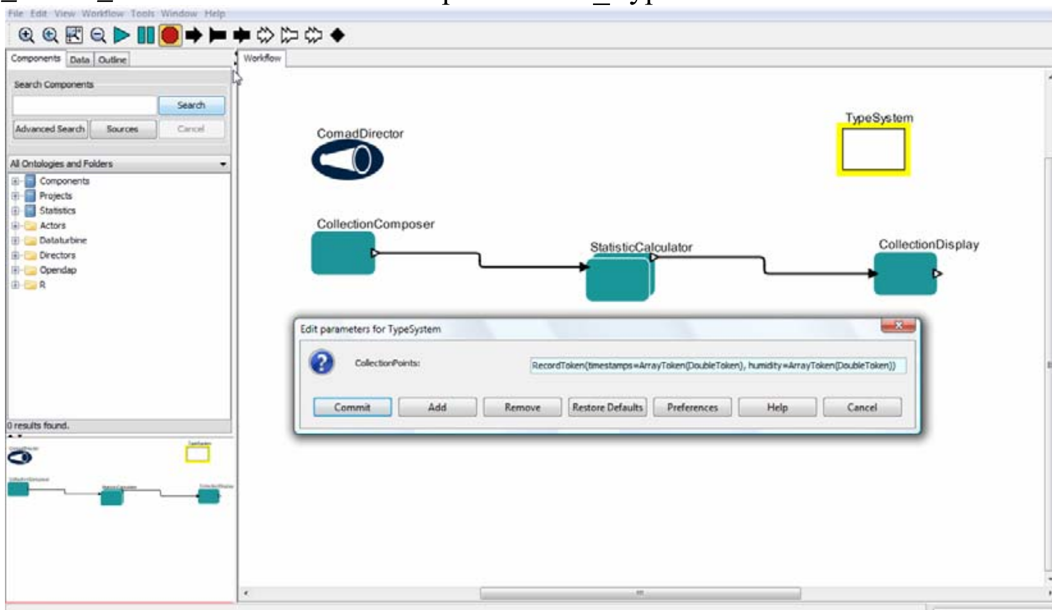

Figure27. Type Alias Registered in TypeSystem

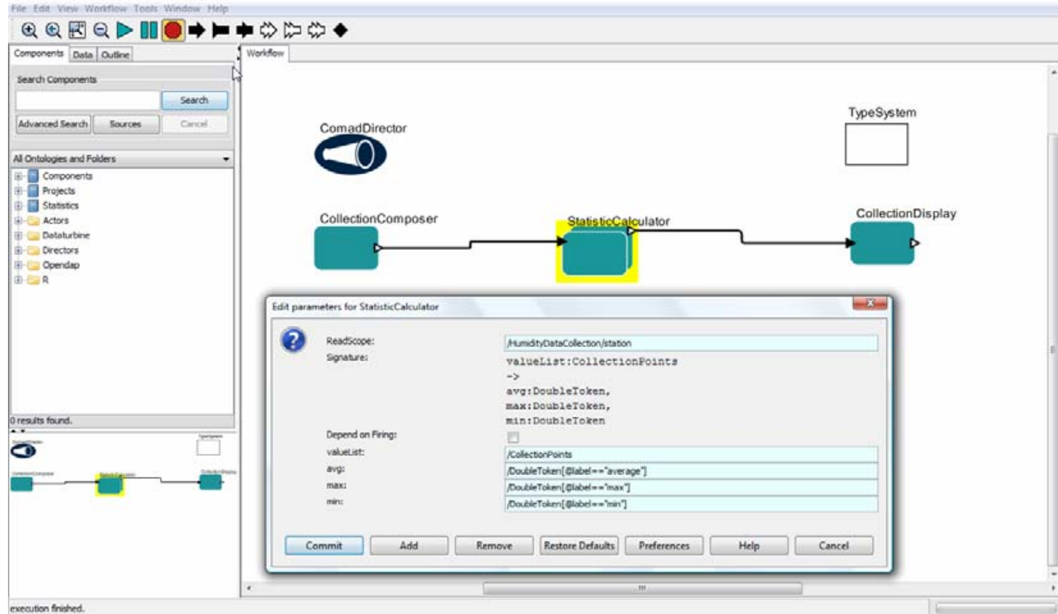

Figure28. Type Alias in Data Binding and Signature

7.1.2. User-Defined Type

For domain-specific data, sometimes user-defined type is more convenient to encapsulate and operate data.

Similar to type alias, user-defined type is registered in "TypeSystem" component with type name and the class representing domain-specific data object. A data object in user-defined type can't be encapsulated by an annotation token. For data token encapsulating data value in user-defined type, the type can't be inferred from the value and therefore must be declared explicitly. Except these, the user-defined type is used in the same way as the system built-in type once it's registered successfully.

Each domain object must provide a constructor to enable it be initialized from an xml node. This constructor is used to create the object from the external workflow input in either native COMAD forma or general XML format. The key feature of a domain object is that its state must not change once the data token encapsulating the object is inserted into data stream. Thus, a domain object must implement methods for write-locking the object, and verifying that it is indeed locked. The object must also be able to represent its state as XML. These methods elaborated in table 8 are defined in LockableDomainObject interface which extends from the DomainObject interface. Each class of domain object is required to implement LockableDomainObject interface and overwrite these methods.

Table8. Mandatory Methods for Domain Object Class

Method	Description
void setWriteLock()	Write-locks the object for immutability. This method is invoked automatically by system when the

	data token encapsulating this object is created and inserted into data stream.
boolean isWriteLocked()	Verifies that the object is write-locked. This method should be invoked whenever before the state of object is to be changed.
String getXmlContentString(Xml.Indentation indentation)	Renders the state of the object as XML element content which may be a single line of text, or one or more XML elements. The indentation parameter is used to indent the return value to an appropriate level of tabulation

To demonstrate how user-defined type is defined and used, a CollectionPoint type is used in Statistic workflow to encapsulate a pair of timestamp and humidity data. Figure 29 demonstrates how CollectionPoint type is registered in TypeSystem component. It’s similar to register type alias. The only difference is that the full class name of the domain object should be put as the “default value” of the type instead of the original type name of type alias. The source code of CollectionPoint class and the actor of StatisticCalculatorOnCollectionPoint to consume objects in such type could be found at “*<comad-exp_install_dir>*/src/org/kepler/demo/simple”. The complete workflow could be found at “*<comad-exp_install_dir>*/workflow/demo/Simple/Statistic_UserDefinedType.xml”.

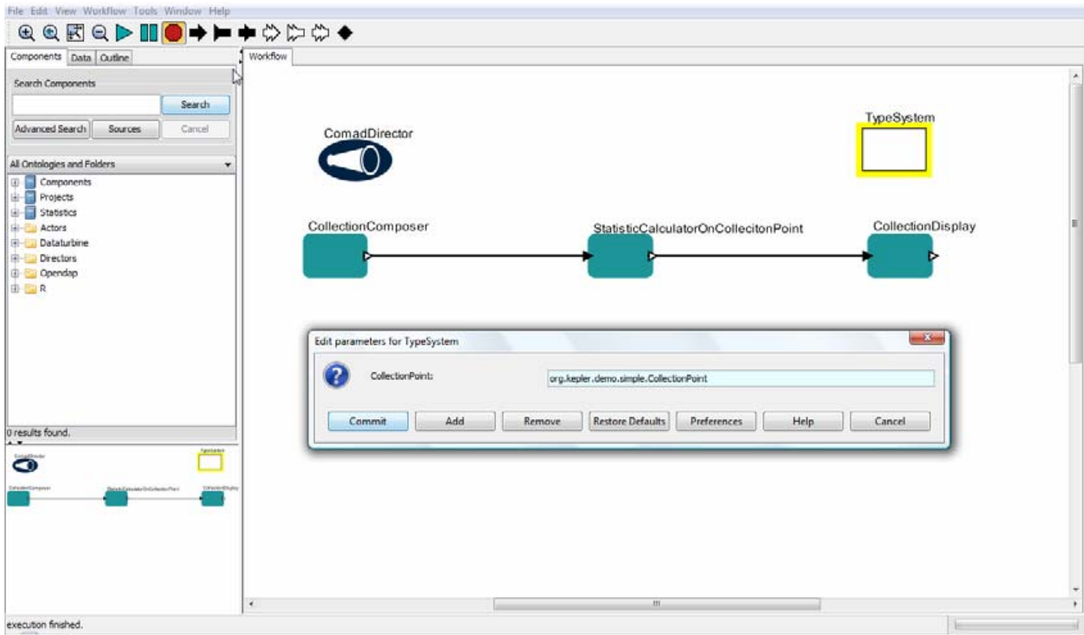

Figure29. User-Defined Type Registered in TypeSystem

7.2. Type Compatibility and Data Format conversion

The compatible relations between COMAD types are defined as type lattice in Figure 30. One type is compatible to another if the first type is allowed to be converted to the second type. In type lattice, the first type is below the second type.

Generally one type is matched to another if the first type is compatible to the second one. But if the second type is decorated with the “exact type match” tag, then the first type is matched to the second type only if it’s exactly the same type as the second one. For example, “IntegerToken” type is matched to “LongToken” type while “IntegerToken” type is not matched to “{-e}LongToken” type. For more details about “exact type match” used in the path expression, please refer to the “COMAD PATH Expression Syntax” chapter.

The compatible relation between two ArrayToken is decided by their element types. If their element types are compatible, then they’re compatible. Otherwise, they’re not. For example, ArrayToken(IntegerToken) is compatible to ArrayToken(ScalarToken).

The compatible relation between two RecordToken is decided by both the element name and type. If they have the same group of element name and for each pair of element with the same name their types are compatible with the same direction (all element types of one RecordToken are compatible to the types of another, or vice versa), they’re compatible. Otherwise they’re not. For example,
RecordToken(name=IntegerToken,value=DoubleToken) is compatible to
RecordToken(name=StringToken,value=ScalarToken).

Besides implementing DomainObject interface, the user-defined type can also inherit the other token type. In this case, the user-defined type is compatible to the token type which it inherits. One user-defined type is compatible to another if the first one is subclass of the second one.

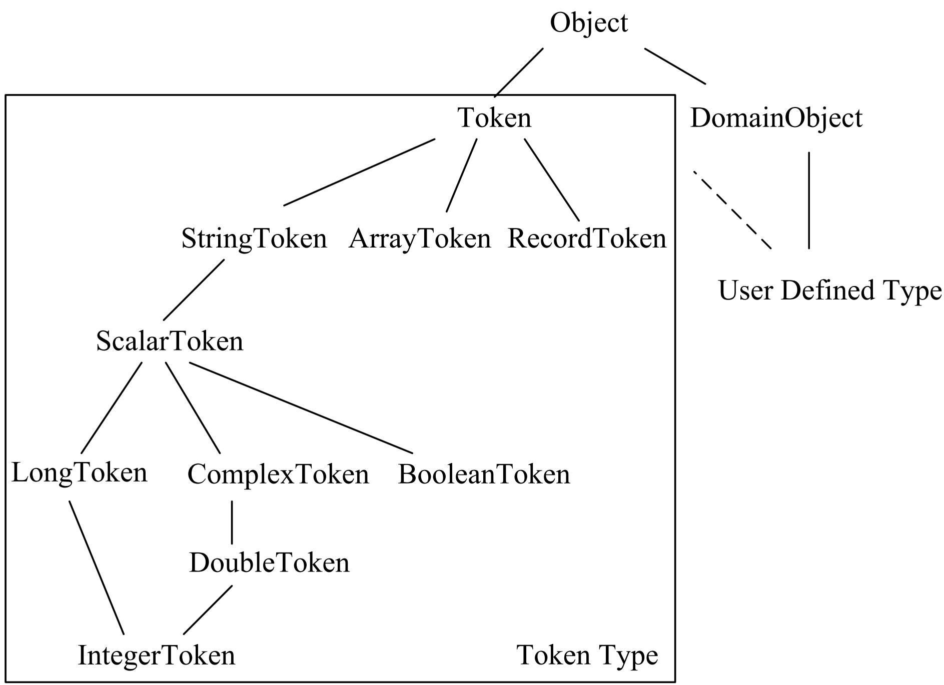

Figure30. COMAD Type Lattice

The type compatibility is used to test the consistency between data, data binding and signature. But COMAD doesn't do type conversion. For example, it's valid to output an IntegerToken while the output signature is "StringToken" since IntegerToken is compatible to StringToken. But we won't really convert the IntegerToken into a StringToken. And finally it's the IntegerToken written into data stream instead of a StringToken.

Although COMAD doesn't do any type conversion, yet it does do data format conversion automatically if possible to provide better data messaging service for actor. Therefore actor could focus on data processing logic and then be easily reused. The workflow can also benefit from this to present neat structure. For these convertible data formats, their types are treated as compatible in the consistency test although originally they're not.

The data format conversion could be done to convert the buffered input data to the format as expected by the input port signature, or to convert the output data to the format as expected by the output port data binding. There're two kinds of data format conversion:

- List vs. ArrayToken: No List type is directly supported by COMAD. But, if the cardinality of signature or data binding is * or +, the type of the signature or data binding is actually a List while its element type is the type defined in signature or data binding. For example, the type of signature "/DoubleToken+" is List<DoubleToken>. COMAD supports data format conversion between List and ArrayToken if and only if their element types are compatible. So whether a List type is compatible to an ArrayToken type or vice versa is also decided by their element types. For example, the data could be converted between the formats of ArrayToken(DoubleToken) and List<DoubleToken >. The

ArrayToken(DoubleToken) type and DoubleToken+ are compatible to each other. The data can only be converted from the format of ArrayToken(DoubleToken) to a list of scalarToken. The ArrayToken(DoubleToken) type is compatible to scalarToken+ but the opposite direction is not true.

- singleton case: In mathematical definition, singleton is a set with one element. Similarly, the singleton case of COMAD is to convert one single data into a List or an ArrayToken with that data as its only element. The precondition is the type of the single data must be compatible to the element type of the List or ArrayToken. For example, in singleton case, an data could be converted from an IntegerToken to an ArrayToken(LongToken) or List<ScalarToken>.

Besides the above general cases of data format conversion, special data format conversion is done at CompositeCoactor. The workflow nested inside CompositeCoactor is a normal kepler workflow which can only receive token type. Therefore, all the data with user-defined type given into or output from the nested workflow is encapsulated in an ObjectToken. The CompositeCoactor assembles data into an ObjectToken and disassembles ObjectToken to get the encapsulated data automatically before and after firing inside workflow.

In COMAD, the type compatibility relations among data, data binding and signature at each port are concluded in table 9.The type declared in signature or data binding is denoted as basic-type while the type of signature or data binding is denoted as signature-type or binding-type. For example, in data binding like "/DoubleToken+", the basic-type is DoubleToken while the binding-type is List<DoubleToken>. The left side type of "≤" is compatible to the right side type.

Table9. Type Compatibility Relations of Data, Data Binding and Signature

Port Category	Type Relations	Example	Description
Input	data type ≤ basic-type of data binding	IntegerToken: ScalarToken IntegerToken: ScalarToken +	The incoming data is bound only if its type is compatible to the basic-type of input data binding.
	binding-type ≤ signature-type	IntegerToken:ScalarToken IntegerToken+: ScalarToken+ ArrayToken(IntegerToken): ScalarToken+ IntegerToken+:ArrayToken(ScalarToken) IntegerToken: ArrayToken(ScalarToken) IntegerToken: ScalarToken+	Binding-type must be compatible to signature-type. Otherwise, an exception is thrown out. Sometimes, the input data prepared by data binding needs to be converted to target format expected by signature.

Output	Data type ≤ signature-type	IntegerToken: ScalarToken ArrayToken(IntegerToken): ScalarToken+ List<IntegerToken>:ScalarToken+ List<IntegerToken>:ArrayToken(ScalarToken)	The type of output data must be compatible to the signature-type. Otherwise, an exception is thrown out.
	signature-type ≤ binding-type	IntegerToken:ScalarToken IntegerToken+: ScalarToken+ ArrayToken(IntegerToken): ScalarToken+ IntegerToken+:ArrayToken(ScalarToken) IntegerToken: ArrayToken(ScalarToken) IntegerToken: ScalarToken+	Signature-type must be compatible to the binding-type. Otherwise, an exception is thrown out. Sometimes, the output data needs to be converted to the target format expected by data binding.

8. Demo

Except for the Statistic workflow introduced through this manual, three other demos are included in this install package to show multiple features of COMAD.

All the workflows could be found under the corresponding directory at "<*comad-exp_install_dir*>/workflow/demo". All the source code could be found under the corresponding directory at "*comad-exp_install_dir*>/src/org/kepler/demo".

8.1. Comet

Comet workflow analyzes meteorological data coming from comet project (http://comet.ucdavis.edu/wiki/index.php/Main_Page). The main analysis steps are:

- Collect hourly humidity data in ten days from the weather stations
- Aggregate the hourly data based on a group of time window and calculate basic statistic data, like min, max etc, for the data aggregated in each window.
- Make further analysis of the data in each window, like calculating the Growing Degree Day, drawing time-based trend graph or making some comparison

Basically the workflow is composed by six actors:

- CollectionComposer: convert external input data from one station into COMAD data stream.
- WindowsGenerator: generate a group of time windows to aggregate data according to specific parameters, including number of generated windows, the start time of the first window, the interval of each window, and the slide time between adjacent windows.

- Chunker: aggregate the data based on the time windows and make basic statistics for the group of data in each window.
- GddCalculator: compute gdd (growing degree day) according to the statistic data of min and max for each window
- RPlotter: draw the average and gdd time-based trend graph for each window by using RExpression.
- TraceWriter: write the whole data collection and all related provenance information into a file for the future provenance analysis.

This demo shows the following features of COMAD through four workflows.

Linear structure: Usually COMAD workflow is linear. The Figure24 shows the comet workflow built with COMAD while Figure25 shows the same workflow built with PN director. Obviously, COMAD workflow with linear structure demonstrates the original data analysis process more clearly.

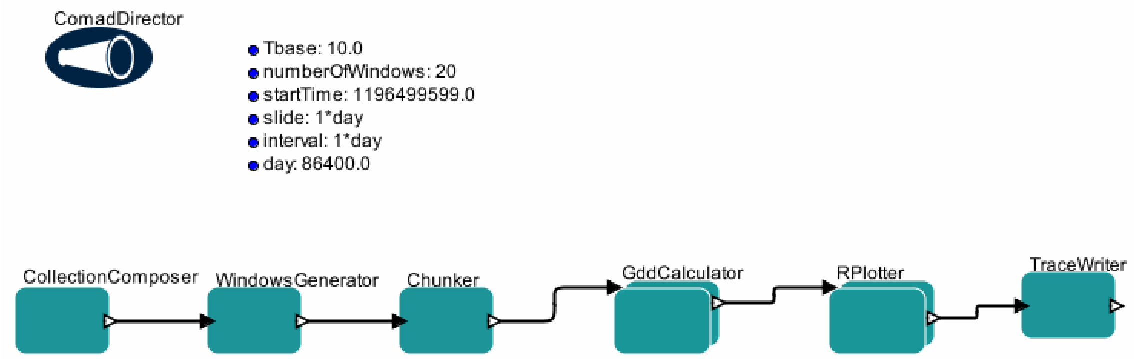

Figure31. Comet workflow with COMAD Director

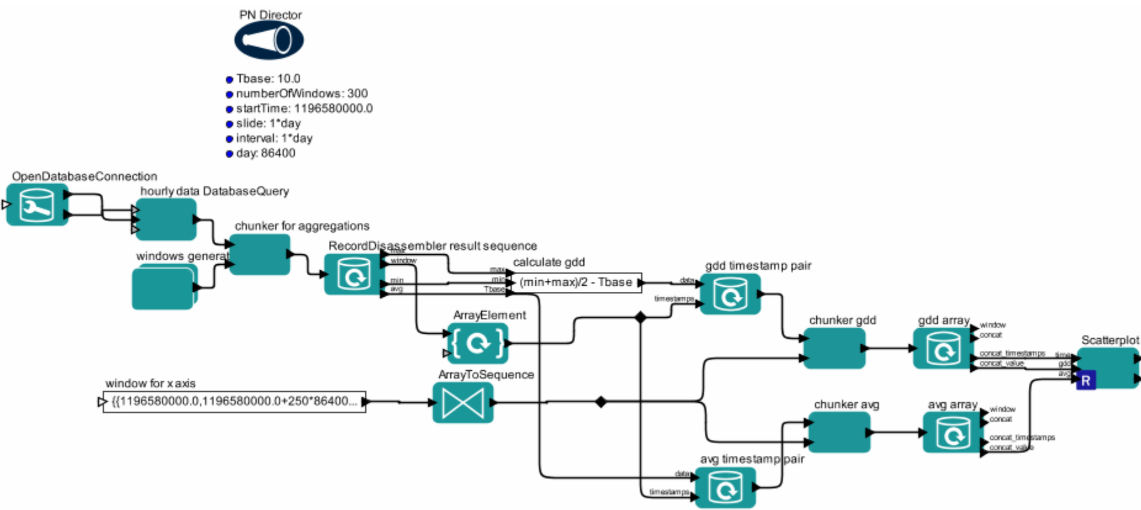

Figure32. Comet workflow with PN Director

Streaming mode: The data inside COMAD is organized into a data stream. Different actors are possible to work on different parts of data stream at the same time. The system

performance could be improved through such Parallelism. The data streaming feature is showed in one_station_SequencePlotter workflow. Based on one_station_RPlotter workflow, the one_station_SequencePlotter is constructed by adding a SequencePlotter actor after the RPlotter actor. And the streaming feature could be seen clearly in the following ways:

- The SequencePlotter draws the graph based on the received sequence of data. Once it receives a data, it draws the point accordingly. By setting the "delay" parameter of the SequencePlotter to increase the time distance to draw the graph between each accepted data, we can clearly see the data comes one by one in a streaming way and the graph is drawn one point after another.
- Although SequencePlotter is behind RPlotter, yet it begins to plot before the RPlotter because RPlotter needs to wait until all data arrives but it doesn't block the data stream. Therefore the SequencePlotter could still receive data and begins to plot before RPlotter.

High reusability and adaptability of the actor: The two_station_RPlotter and three_station_RPlotter workflows implement the same function as the one_station_RPlotter workflow. The only difference is the source data comes from more stations which might belong to different county. The change of source data structure is showed in Figure26. By changing binding configuration, the COMAD actor is easily reused and adapted to the change of the data structure and workflow organization:

- To adapt to the data structure change, the only change from one_station_RPlotter to two_station_RPlotter is the read scope of Chunker actor. It's changed from "/" to "//station".
- In two_station_RPlotter, to draw graph for one station each time instead of two, just need to change the read scope of RPlotter from "//AggregationResult" to "//AggregationResult[@station_number=="s2"]"
- Nothing needs to be changed from two_station_RPlotter to Three_station_RPlotter to deal with data with additional added hierarchy.

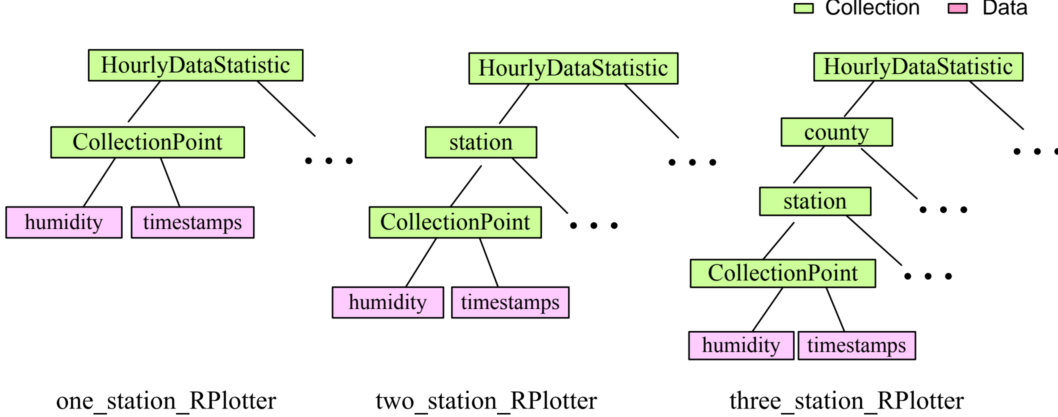

Figure33. Source Data Structure of Comet workflows

This demo could also help user to understand how easy it's to develop an actor and how powerful the path expression syntax is to express various binding requirement. By

browsing the trace file generated by TraceWriter or use the "Provenance Browser" to browse and query the data lineage, the user could also see how great support provided by COMAD for provenance capturing.

8.2. PICalculation

Usually COMAD workflow is linear. But comad can also support "do-while" loop structure. PICalculation workflow demonstrates how to implement such "do-while" loop in COMAD.

PICalculation workflow calculates PI through multiple iterations. The formula implemented in this workflow to calculate PI is: PI = 4-4/3+4/5-4/7…..

PICalculation consists of five actors:

- CollectionComposer: compose the initial data stream consists of initial data value and iteration number used for PI calculation
- StartLoop and EndLoop define the loop boundary. In both actors, the target collection which is looped over is defined through loopCollectionLabel parameter. The loop termination condition is defined in EndLoop. In this workflow, the termination condition is the iteration times.
- PICalculator: calculate the new PI value according to the iteration number and the current PI with the above formula.
- CollectionDisplay: display the final PI calculation result and the iteration number.

Besides loop structure, PICalculation workflow also shows how to delete data or annotation through input binding path expression. By checking "show provenance" choice in CollectionDisplay actor, the fully collection modification traces is shown, including all deleted items, instead of showing only the final result.

8.3. MobyService

COMAD actor is configurable, including the number of input/output port, corresponding signatures, and even the data processing logic. Benefiting from this ability, a group of configurable data processing services could be simply modeled as one COMAD actor instead of a large number of actors with similar behavior.

BioMoby service is for biological data analysis. All BioMoby services are registered in several registries. Similar to web service, there's standard API to search BioMoby service, get metadata of each service, including input/output number and type etc, and access the service.

As an abstract BioMoby service client, the SimpleMobyServiceAccessor actor could be instantiated to encapsulate specific BioMoby service according to the service name configuration. In SimpleMobyServiceAccessor, the service name is configured through MobyServiceName parameter by typing or choosing the name from the drop list. After commit to make the configuration take effect, the actor is instantiated to be the specific service client. Each input or output of the service corresponds to an input or output data binding port in the actor, while the type of input or output is used to generate the

signature correspondingly. In this way, thousands of BioMoby services could be modeled by one COMAD actor.

Two workflows composed by BioMoby services are demonstrated. Blast workflow gets blast analysis result for a gene or protein denoted by the input identifier. PhylogeneticTree workflow returns a phylo-genetic tree for a group of input gene or protein in Fasta format. All BioMoby services inside the workflows are configured and instantiated from SimpleMobyServiceAccessor actor. The PhylogeneticTree workflow is showed in Figure 34.

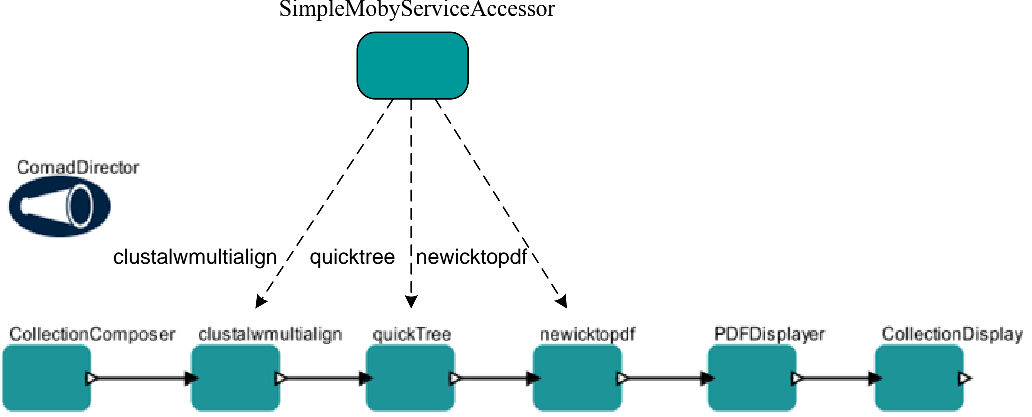

Figure34. PhylogeneticTree workflow. Three actors encapsulating three BioMoby services are actually instantiated from the same SimpleMobyServiceAccessor actor by configuring the service name.

Notice: It takes some time to open a workflow composed by BioMoby service, because the actor needs to be instantiated with metadata of the service got from remote service registry. So internet access is required to run this kind of workflow.

9. Appendix: Actor Reference

COMAD system provides some ready to use actors summarized in Table10. User is also supported to develop new actor as introduced in "COMAD Actor" chapter.

Table10. Ready-to-use COMAD Actors

Category	Description	Actor Name
Data Stream Composer	Import the external data into workflow by composing it into data stream	CollectionComposer
		CollectionReader
		CreateRootCollection
Data Stream Renderer	Render the data stream of workflow	CollectionDisplay
		XmlDisplay
		TraceWriter
Loop Structure Controller	Controls the loop boundary	StartLoop
		EndLoop
Data stream Filter	Filters the elements out from data stream	Filter

Workflow Composer	Encapsulate nested workflow	CompositeCoactor

9.1. CollectionComposer

The CollectionComposer actor imports external data into COMAD workflow. The external input data is actually losslessly converted into a data stream composed by tokens.

Ports:

- output: The output port from where the data stream flows out of the actor.

Parameters:

- Schema: The format of external data. Two choices are provided, native COMAD or General XML.
- Collection XML: The text window to put the input data.

9.2. CollectionReader

The CollectionReader actor imports external data from a file into the COMAD workflow. The external input data is actually losslessly converted into a data stream composed by tokens.

Ports:

- output: The output port from where the data stream flows out of the actor.

Parameters:

- Schema: format of external data. Two choices are provided, native COMAD or General XML.
- File: full path of the input file.

9.3. CreateRootCollection

The CreateRootCollection actor creates an empty root collection as the source data stream of the workflow.

Ports:

- output: The output port from where the data stream flows out of the actor.

Parameters:

- Root label: label of the root collection.

9.4. CollectionDisplay

The CollectionDisplay actor renders a string representation of its input stream inside the read scope in native COMAD format to a GUI display window.

Ports:

- input: The input port from where the data stream flows into the actor.

- output: The output port from where the data stream flows out of the actor.

Parameters:

- ReadScope: read scope path expression
- showDetails: Element IDs, and object IDs and Java classes of data items, are included if set to true.
- showProvenance: Insertion records, deletion records, invocation dependency records, and deleted elements are included if set to true
- stringsOnSingleLines: CR and LF characters are removed from String data values if set to true
- remapIds: All element ids and object ids are separately mapped to a group of integer starting from 1 if set to true
- columnsDisplayed: Width of the display in characters
- rowsDisplayed: Height of the display in characters.
- clearDisplayOn: Display will be cleared on arrival of collections with this label.

9.5. XmlDisplay

The CollectionDisplay actor renders a string representation of its input stream inside the read scope in general XML format to a GUI display window.

Ports:

- input: The input port from where the data stream flows into the actor.
- output: The output port from where the data stream flows out of the actor.

Parameters:

- ReadScope: read scope path expression
- useDataTypesAsTags: indicate whether the name of element in XML representation created from data token is its data type name
- columnsDisplayed: Width of the display in characters
- rowsDisplayed: Height of the display in characters.
- clearDisplayOn: Display will be cleared on arrival of collections with this label.

9.6. TraceWriter

The TranceWriter actor serializes an XML representation of its input stream to a file. The trace file is output to the directory specified by "output directory root" parameter of COMAD director. And the file is named as "*workflowName_timeStamp_traceId*.trace".

Ports:

- input: The input port from where the data stream flows into the actor.
- output: The output port from where the data stream flows out of the actor.

Parameters:

- remapIds: All element ids and object ids are separately mapped to a group of integer starting from 1 if set to true

9.7. StartLoop

The StartLoop actor defines the start boundary of a sub-workflow that loops over a specified collection.

Ports:

- input: The input port from where the data stream flows into the actor.
- output: The output port from where the data stream flows out of the actor.
- loopback: The input port for the collection looped back from the EndLoop actor.

Parameters:

- ReadScope: read scope path expression
- loopCollectionLabel: The first collection with this label inside the read scope is the loop collection. It must be the same as loopCollectionLabel set in EndLoop actor.

9.8. EndLoop

The EndLoop actor defines the end boundary of a sub-workflow that loops over a specified collection.

Ports:

- input: The input port from where the data stream flows into the actor.
- output: The output port from where the data stream flows out of the actor.
- loopback: The output port for the collection looped back to the StartLoop actor.

Parameters:

- ReadScope: read scope path expression
- loopCollectionLabel: The first collection with this label inside the read scope is the loop collection. It must be the same as loopCollectionLabel set in StartLoop actor.
- whileExpression: continue loop when it's evaluated as true. The expression is based on the annotation of the looped collection. The annotation is denoted as M{*annotation_name*}. During evaluation, the value of the specified annotation is used.
- maxLoopCount: the maximum loop times. The loop will stop if the max loop number has been reached, no matter whether the "while" condition is true or not
- exitOnException: exit loop if any exception happens.

9.9. Filter

This actor filters out specified collection, data or annotation from data by using delete decoration in the read scope or data binding path expression. The actor simply deletes all the elements decorated with deletion tag.

- If the target deleted element is a collection
  - put the target collection path as the read scope with deletion decoration
  - or put path for the DeletedElement data binding port which involves the target collection and use the deletion decoration for that collection.

- If the target deleted element is a data or annotation token
    - put the path to the target data or annotation with deletion decoration

If there's no deletion decoration in both read scope or DeletedElement data binding expressions, an exception is thrown out.

Ports:

- input: The input port from where the data stream flows into the actor.
- output: The output port from where the data stream flows out of the actor.

Parameters:

- ReadScope: read scope path expression. Meanwhile, it could also be used to declare the deleted collections.
- Signature: displays the signature of the actor
- DeletedElement: The optional input port data binding to declare the deleted element.

9.10. CompositeCoactor

The CompositeCoactor actor is used to encapsulate a sub-workflow assembled from a group of kepler actors. The director for the sub-workflow can only be SDFDirector.

The input and output port of the nested workflow are actually the input and output data binding port of the CompositeCoactor. From these ports, the input data is bound and prepared to fire the inside workflow and the output is written into the data stream. Please refer to chapter "COMAD Actor" for more details about how to use this actor.

Ports:

- input: The input port from where the data stream flows into the actor.
- output: The output port from where the data stream flows out of the actor.

Parameters:

- ReadScope: read scope path expression. Meanwhile, it could also be used to declare the deleted collections.
- Signature: displays the signature of the actor
- Depend on Firing: If it's set true, then the dependency is generated only from the input data of the same firing. Otherwise, the dependency is generated from all the input data in the invocation. Please refer to "Provenance Recording" in "COMAD Actor" chapter for more details.