

# **Software Requirements Specification**

for

## **Workflow Editor module of the FlowOpt project**

Prepared by: Vladimír Rovenský

22.4.2010

## **Purpose**

This document is a part of the FlowOpt project specification – it specifies software requirements on the workflow editor module which will be used to create, manipulate and visualize workflows and subsequently pass them to other FlowOpt modules. The FlowOpt project is a software suite designed for working with workflows and schedules. It will be integrated into the MAKE project.

## **Used terms and remarks**

This section describes the terminology used throughout this document, font conventions and other meta-information relating to this document as a whole.

Throughout this document, I sometimes refer to information contained in other parts of the FlowOpt project specification (for example the declarative format for workflows). I therefore assume that the reader is familiar with the rest of the FlowOpt project specification, or at least its parts relevant to this module.

### **Font conventions**

In the text, whenever I mention some term that will be referred to later in the document, it is written in *italics*. Fragments of declarative format code will be written in `courier`. In some cases, I'm not yet certain on the solution of the specific problem – these cases are marked by (TBD) mark and will be resolved later (advice on such problems is especially welcome).


### **Declarative format**

Examples in this specification use simple declarative pseudocode to show how various workflow transformations are done by this module. This pseudocode is not used by the application, it is here merely to illustrate the process done by the application in a way that is relatively easy to read.

The pseudocode consists of predicates, which represent various data elements used in the FlowOpt application (see *Data model* chapter for details). For internal data representation, we chose an XML form of the (same) data model, rather than this declarative one for obvious reasons – XML is easily parsed. However, they represent the same semantic data model, only in a slightly different form (XML in elements, declarative pseudocode in predicates). I still use the declarative format for examples, as it seems to be easier to read and understand quickly.

### **Various remarks**

For the sake of this document, clicking refers to left clicking; dragging refers to dragging the mouse while holding the left mouse button.

The symbol  represents a user's mouse cursor in the examples of GUI usage.

## **Application overview**

This application will be a part of the FlowOpt project – specifically it will be a graphical workflow editor module, which will allow the user to create or import a workflow, edit it, save it and use it for scheduling by other FlowOpt modules. Emphasis will be on user friendliness, usability and completeness with regard to this specification.

## Hardware and software requirements

The application will be written in C# language and therefore, the user will be required to have .NET framework installed to run this application (mono or similar environments will not be supported). As the application will be integrated into MAKE, it will share all of its hardware and software requirements – place consult MAKE documentation.

## Operating Environment

The application will run under the MS Windows OS within the .NET runtime. It will be a component of the MAKE project – for further reference see the MAKE project specification.

## Workflow model

All workflows created by this module have to conform to a specific workflow model. This model is based on the Nested TNA model, but there are some differences. The user will be able to perform all operations relevant to the Nested TNA model (parallel / alternative / serial decomposition), plus he will be able to arbitrarily specify precedence, logical and synchronization constraints. This means that the user can choose any two vertices in the workflow graph and specify a precedence, logical or synchronization constraint for them. All this will be thoroughly specified later in this document.

## Nested TNA model description

Our workflow model is based on the Nested TNA workflow model introduced in <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.69.5684&rep=rep1&type=pdf>. Any information on this model not provided by this document can be found there.

## Domain model

The application will serve as graphical workflow editor; hence the main object of interest will be a workflow. Workflow in this case can be considered a directed graph  $G(V, E)$ . Vertices ( $V$ ) will be called *tasks* and *activities*. Arcs ( $E$ ) will be of three kinds: *precedence*, *logical* and *synchronization* arcs. All of these will be further described below.

## Tasks and Activities

*Activities* and *tasks* form the core of the workflow. Activities effectively say what should be done – they are the elementary components which build the workflow and they cannot be further decomposed. Tasks on the other hand serve as placeholders for activities or other tasks and store the structure of the workflow. Main difference between an activity and a task is the fact that activities are scheduled for execution when using the scheduler module of Flow Opt project, whereas tasks are not – they only inform the scheduler of the workflow structure. Also, an activity has a number of properties that the user can fill – for example cost, duration etc. Tasks have no such properties (only an identifier).

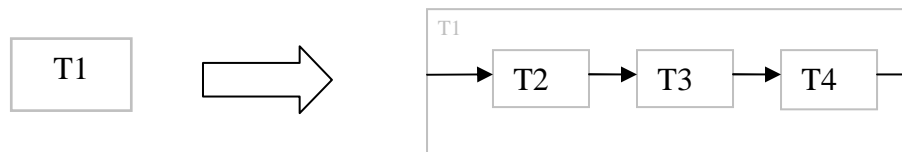
One task can be thought of as a slot that can contain either a single activity, or a *nest*. A nest is created by decomposing a task (which is then called a *parent task*) using parallel, alternative or serial decomposition, which will be thoroughly specified later in the document. For sake of this section, a nest is a set of tasks which are to be executed in place of the parent task.

If a task doesn't contain any activity or task, it is an *empty* task. Note that an empty task means an incomplete workflow – it is not the same as say an activity that doesn't do anything, the user will have to assign an activity to all empty tasks before the workflow is considered complete (ready for scheduling).

When creating a new workflow, the user will start with a single task (called *initial task*), representing the whole workflow process. He will then decompose this initial task into more

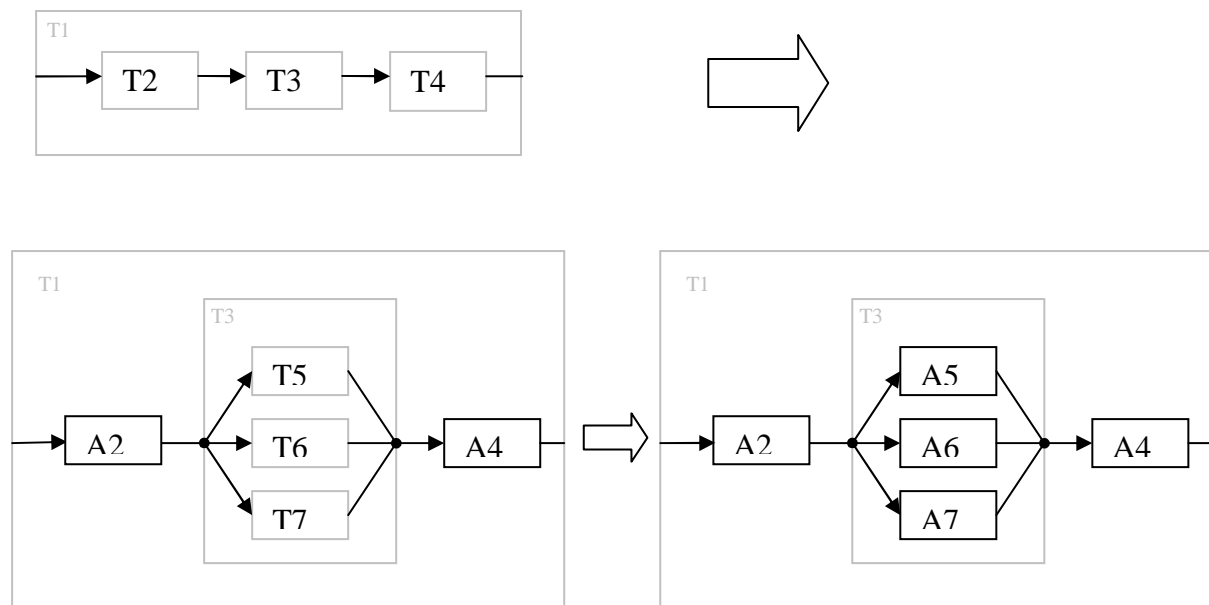
tasks, forming the structure of the workflow. To complete the workflow, the user will assign activities to empty slots.

The process is illustrated by the following picture (empty tasks are gray rectangles; tasks with activities are black rectangles):



Here the user started with one initial task T1 – this task is now empty. Then the user decomposed T1 into three tasks T2, T3 and T4. These three tasks now form T1's nest – T1 is no longer an empty task, however T2, T3 and T4 are empty tasks. The user may now either decompose them to fill them with their own nests, or he can assign them an activity to fill them.

In the following figure, T2 and T4 are filled with an activity (A2 and A4 respectively), while T3 is decomposed into three more tasks T5, T6 and T7, which are subsequently filled with activities (A5, A6 and A7) to complete the workflow.



The resulting workflow has 7 tasks and 5 activities. Initial task T1 is filled with its nest composed of T2 (filled with activity A2), T4 (filled with activity A4) and T3 filled with its own nest composed of tasks T5, T6 and T7 (filled with activities A5, A6 and A7 respectively).

The nature of decomposition operation is not relevant at this point, only the difference and use of activities and tasks is relevant.

The user will only be able to use activities previously defined by the MAKE application – workflow module itself will not allow the user to create / edit / delete activities. It may however invoke specific parts of the MAKE application that provide this functionality (TBD).

**Note:** For user convenience and for sake of intuitive use, tasks containing only a single activity may also be referred to as activities.

## **Precedence, Logical and Synchronization arcs**

Arcs in the workflow graph represent various constraints on tasks. Some arcs are created automatically through decomposition (these are only precedence arcs), but the user can arbitrarily create arcs of all kinds.

All arcs can be only connected to tasks (remember, activities are not considered vertices in the workflow graph). Note that tasks have their start and end times specified implicitly – start time of a task is the start time of its first (time wise) scheduled activity, while end time of a task is the time of its last (time wise) scheduled activity. Therefore, if a task only has one activity within itself, the start and end times of this task and its activity are identical and the arc effectively connects to this activity.

Of the three kinds of arcs, precedence arcs are probably the most common – they simply represent order within the workflow: when task T1 is connected to task T2 using a precedence arc, it means that T1 has to be completed before T2 can start.

Logical arcs serve to establish some logical constraints on tasks. There are three kinds of logical arcs: equivalence, implication and mutex. Suppose we have tasks T1 and T2. Then a logical arc (T1, T2) with equivalence constraint means that T1 is to be scheduled for execution if and only if T2 is scheduled for execution (time doesn't matter, only the fact that they are either both scheduled, or none of them is ). A logical arc (T1, T2) with an implication constraint means that if T1 is scheduled for execution, then T2 also has to be scheduled for execution. Finally, a logical arc (T1, T2) with a mutex means that if T1 is scheduled for execution, T2 cannot be scheduled for execution and vice versa – the two tasks are to be mutually exclusive for the scheduler. Finally, a logical arc (T1, T2) with inclusive or constraint means, that at least one of the tasks T1, T2 has to be scheduled for execution.

Last kind of arcs is the synchronization arcs. These form a synchronization constraint between two tasks and are of three kinds: start-start (SS), end-end (EE) and end-start (ES) synchronization. Suppose we have tasks T1 and T2, then a synchronization arc (T1, T2) with SS constraint means that T1 and T2 must start at the same time. A synchronization arc (T1, T2) with EE constraint means that T1 and T2 must end at the same time. Finally, a synchronization arc (T1, T2) with ES constraint means that T1 must end at the same time that T2 starts.

## User interface

Below is a very basic overview of the main form.

Main menu – File   Edit   ...   Help	
Toolbar – icon shortcuts for aligning, orientation switch etc.	
Drawing area – where the workflow is displayed and edited	Activity properties – ID, duration, cost, due date etc, When an activity is selected, its data can be seen and edited here
Hint box – miscellaneous relevant information for current operation	

## Zooming

The user will be able to zoom in and out. There won't be a magnifying glass functionality (i.e. the possibility to enlarge just a selected part of the screen, without the rest changing size). Zooming distance will be limited.

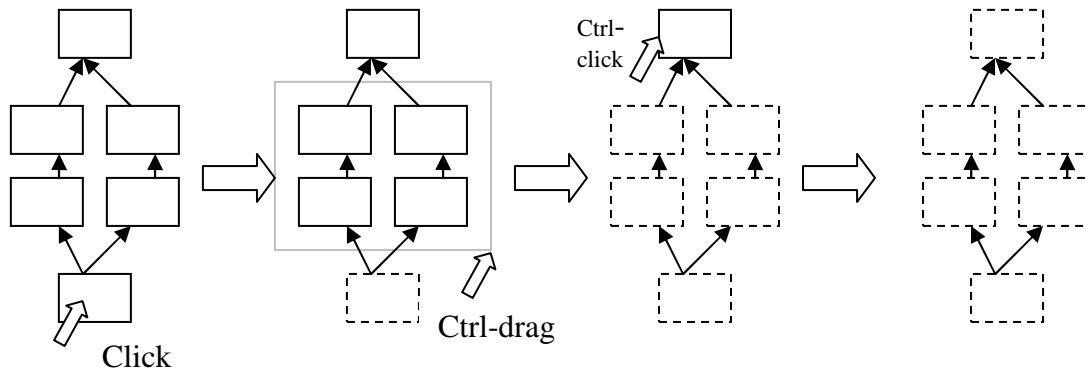
## Undo

The user will be able to perform an undo operation for all main operations. Its depth will not be unlimited. In some cases where undo operation is not well defined, it may not be available (TBD).

## Selection

The user will be able to select a single activity by left clicking on it (or more precisely on its task). By doing so, its properties will be displayed in the activity property form, where they can be viewed and edited.

By ctrl-clicking on activities, they are added/removed to/from current selection. By dragging the mouse when nothing is selected, a rectangle appears indicating selection – upon releasing the mouse button, all activities within the rectangle will be selected. By ctrl-dragging the mouse, multiple activities can be added to selection. When multiple activities are selected, their properties cannot be edited; the activity property form is only active when a single activity is selected. Selected activities are highlighted (dashed line). Same applies for arc selection.



## Activity and arc properties

Both activities and arcs will have certain properties, which the user will be able to see. In case of activities, these properties include for example ID, duration, cost, due date, late cost, resource requirements etc. There will be a special panel for viewing and these properties (see main form overview). Upon selecting a single activity, its properties are loaded into this form where the user may view them. Note that all activities will be loaded from the MAKE database, the user will have to use other parts of the MAKE application to create and edit the activities.

As for arcs, they have various properties based on the type of an arc – precedence arcs have no properties, logical and synchronization arcs have a single property: type of their operation. The user may input this information by double clicking on the arc, which will cause an input field to appear, allowing user to enter the property.

## Resources

One of activity properties to take special note of are its resource requirements. Every activity can have arbitrary number of required *resources* assigned to it. A resource is some artifact that the activity reserves when it starts and releases when it ends. An activity can only start when it has all required resources.

Resources themselves are defined within the MAKE application (see MAKE documentation). The user will be able to assign them in the workflow editor module through the activity property panel. A single activity can be assigned 0 to n (predefined) resources. The activity can then only be scheduled when all specified resources are available.

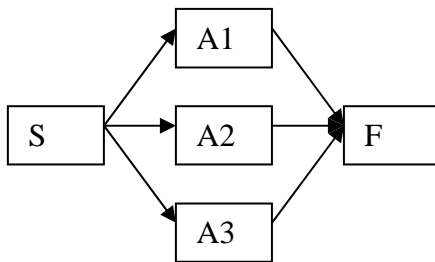
The user will also be able to specify *alternative resources* in the following way. If an activity is to have several alternative resources, the user will mark one of them as *primary resource*. For this primary resource it will be possible to add arbitrary number of *alternative resources*. The activity can then be executed when the primary or any of the alternative resources are available.

Note that the user may specify arbitrary number of primary resources for an activity plus for each primary resource arbitrary number of alternative resources.

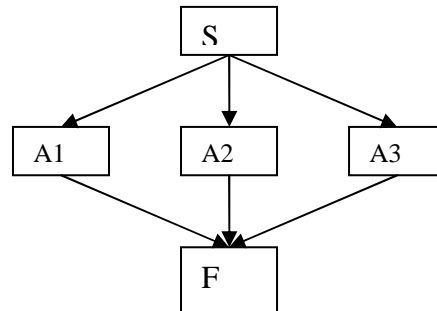
## Choice of orientation

In any moment, the user will be able to switch between two orientations – horizontal (left to right) and vertical (top to bottom). The following figure illustrates both cases:

Horizontal orientation



Vertical orientation



## Editing

In the beginning, the user starts with a workflow consisting of a single task (vertex)<sup>1</sup>. For sake of this specification, this task will be called the *initial task* and workflow consisting of this single task will be referred to as *initial workflow*. On this elementary workflow, the user can apply several editing actions.

To change the workflow structure, the user must perform decomposition – one task is decomposed into several new tasks, which are to be executed in a certain way in place of the original task. Upon decomposition, a task becomes a *parent* task. All tasks created by the decomposition belong to the parent task's *nest*. Tasks that are not parent, i.e. they have not been decomposed, will be referred to as *empty* tasks.

Every nest has two special points to take note of – the *starting point* and the *end point*. The former represents the point at which this nest starts; the latter represents the point at which the nest ends (time wise). These points are marked by a black dot on edges of the nest (see figures below). These points serve to connect tasks within the nest to the parent composite task – the start point corresponds to start of the parent task and the end point corresponds to end of the parent task.

Besides decomposition, the user can also arbitrarily specify binary precedence, logical and synchronization constraints, as described earlier in this document.

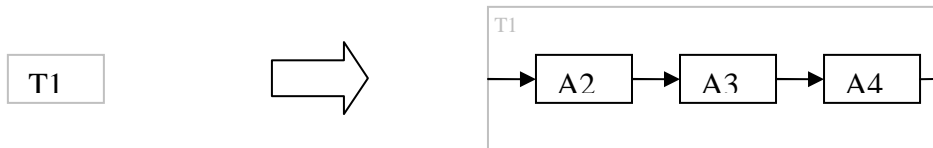
All editing options are further described here:

- **Serial decomposition** – the user can split a single task into multiple tasks, which will be executed one after another in place of the original task. Here T1 is decomposed into tasks T2, T3 and T4 (filled with activities A2, A3 and A4 respectively). T1 thus becomes a parent task, with T2, T3 and T4 in its nest. The editor will visualize the nest like in the figure – as a grey rectangle over all the nested tasks, with a name of the parent task. It will be possible to turn this visualization off.

---

<sup>1</sup> In the formal Nested TNA model, the user starts with two tasks connected by an arc, however unlike the formal model, we decompose the vertices rather than arcs, hence the slight change of model.

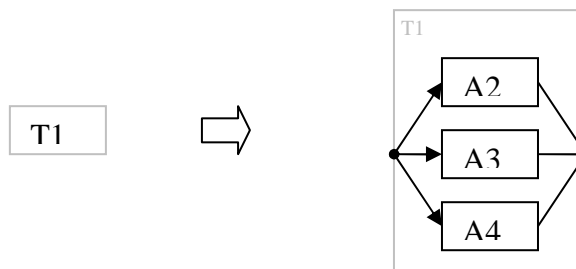




**Declarative format transcription:**

```
Task(T1)
=>
Task(T1)
Activity(A2,A3,A4)
Decomposition(T1,[A2,A3,A4],AND)
Precedence(A2,A3)
Precedence(A3,A4)
```

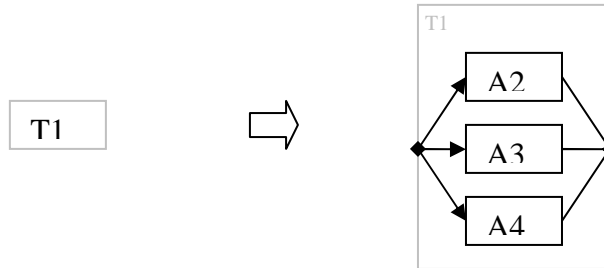
- **Parallel decomposition** – the user can split a single task (vertex) into multiple tasks, which will be executed in parallel in place of the original task (produces AND split and join). In this example, T1 is decomposed into T2, T3 and T4 (filled with activities A2, A3 and A4 respectively). Arcs leading to and from these tasks are marked with a filled circle – this indicates the parallel (AND) relation.



**Declarative format transcription:**

```
Task(T1)
=>
Task(T1)
Activity(A2,A3,A4)
Decomposition(T1,[A2,A3,A4],AND)
```

- **Alternative decomposition** – the user can split a single task into multiple tasks, out of which a single task will be executed in place of the original task (produces XOR split and join). In the following figure, T1 is decomposed into T2, T3 and T4 (filled with activities A2, A3 and A4 respectively). In this case, arcs leading to and from the nested tasks are marked by a filled diamond – this means alternative decomposition.

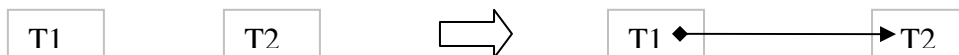


**Declarative format transcription:**

```
Task(T1)
=>
Task(T1)
Activity(A2,A3,A4)
Decomposition(T1,[A2,A3,A4],XOR)
```

- **Adding a precedence arc between two tasks** – the user can create a new precedence arc between two tasks. In this example we specify a precedence arc for tasks T1 and T2.

Note that the arc's starting point is different from the precedence arcs which are created through decomposition and it is also marked (diamond) to indicate that this precedence arc was created arbitrarily, not through decomposition.



**Declarative format transcription:**

```
Task(T1,T2)
=>
Task(T1,T2)
Precedence(T1,T2)
```

- **Adding a logical arc between two tasks** – the user can also specify a logical arc for a couple of tasks T1 and T2. These constraints will be visualized similarly to the precedence constraints – as arcs. These arcs will not be directed and may be drawn with different color. To prevent confusion, the user will be able to only show one kind of arcs (precedence / logical / synchronization), show one and draw the others in the background grayed out, or show both kinds of arcs normally.

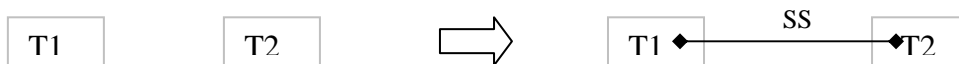
In this example, we specify an implication constraint for tasks T1 and T2. As in the case of arbitrary precedence arcs, arbitrary logical arcs also start in a different point and their start is also marked (diamond) in order to distinguish them from arcs created by decomposition.



**Declarative format transcription:**

```
Task(T1,T2)
=>
Task(T1,T2)
Logical(T1, T2, =>)
```

- **Adding a synchronization arc between two tasks** - The user can add a synchronization arc between two tasks. These are again marked differently from arcs created through decomposition, as illustrated by following figure. Here we specify a start-start synchronization constraint for tasks T1 and T2.



**Declarative format transcription:**

```
Task(T1,T2)
=>
Task(T1,T2)
Synchronization(T1, T2, SS)
```

**Description of editing process**

Here is a step-by-step example of creation of a simple workflow to illustrate the process. It assumes the horizontal orientation (for vertical orientation, the left-right dragging is replaced by up-down dragging and vice versa).

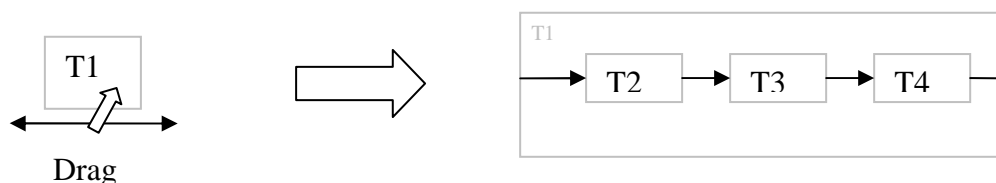
- 1) The user starts with a single task (the initial task), which represents the entire workflow.



**Declarative format transcription:**

```
Task(T1)
```

- 2) Serial decomposition application – the user clicks on the task, which he wants to decompose (in this case T1) and drags the mouse while holding the left mouse button. Dragging to the right adds more tasks to the decomposition, dragging to the left removes tasks from the decomposition (in this case, the user splits the T1 task into three serially executed tasks T2,T3 and T4):

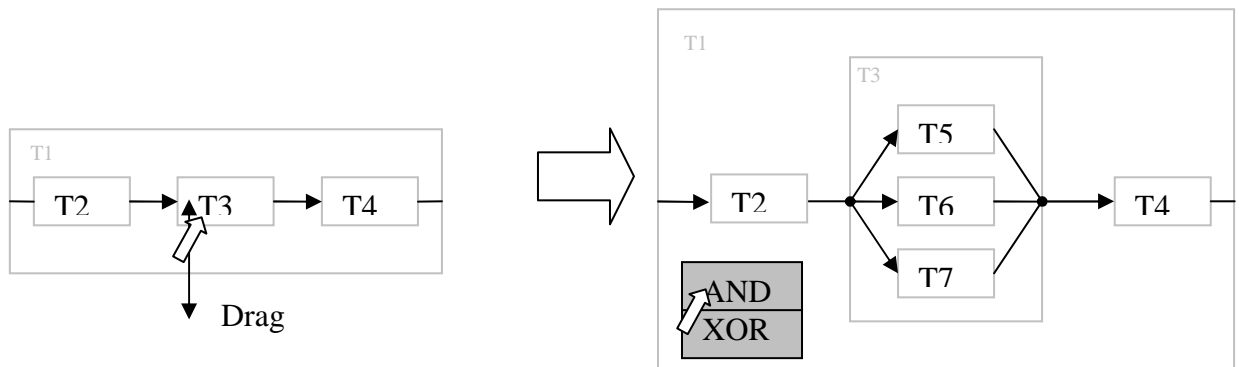


Declarative format transcription:

```
Task(T1,T2,T3,T4)2  
Decomposition(T1,[T2,T3,T4],AND)  
Precedence(T2,T3)  
Precedence(T3,T4)
```

- 3) Parallel decomposition application – The user clicks on the task, which should be decomposed (in this case T3) and by holding the left mouse button and dragging the mouse he specifies the number of tasks in the decomposition similarly as in the serial decomposition case – dragging up adds more tasks into the decomposition, dragging down removes tasks from the decomposition.

Here the user decomposed the T3 task into three tasks T5, T6 and T7, which should be executed in parallel in place of the original T3 task. When the user releases the mouse button, a pop-up menu automatically appears with the choice of either AND split (parallel decomposition) or a XOR split (alternative decomposition) – when the user wishes to perform an alternative decomposition, the process is the same as with parallel, except for selecting XOR split in the pop-up menu.

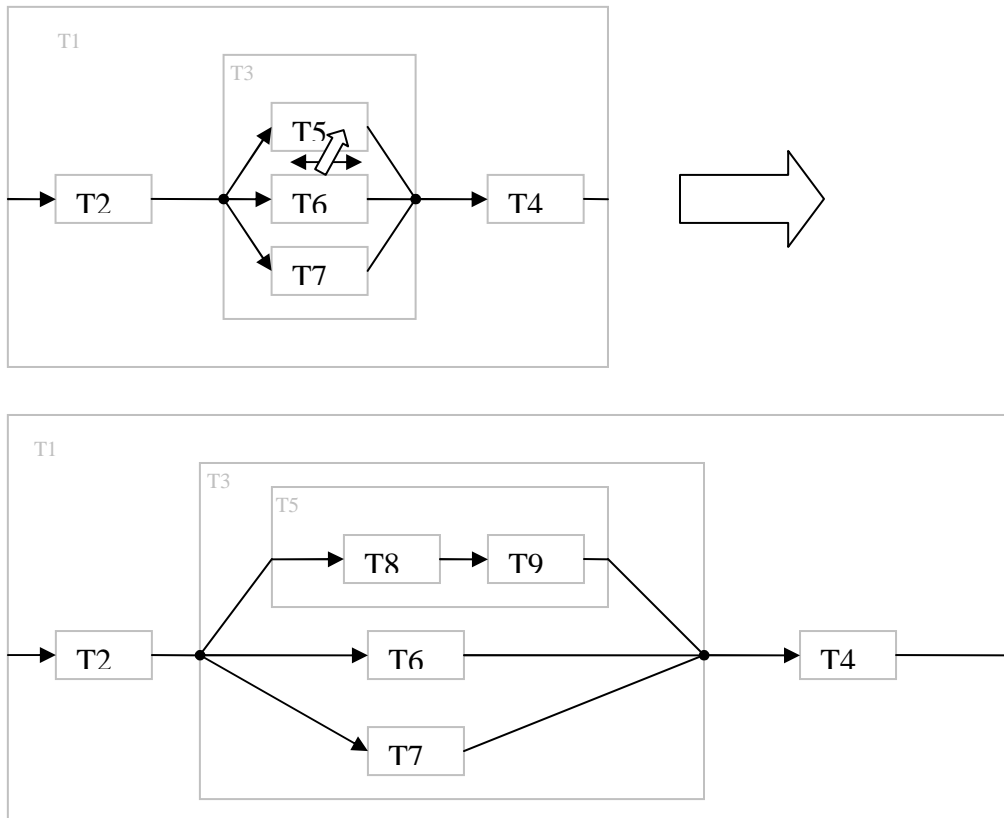


Declarative format transcription:

```
Task(T1,T2,T3,T4,T5,T6,T7)  
Decomposition(T1,[T2,T3,T4],AND)  
Precedence(T2,T3)  
Precedence(T3,T4)  
Decomposition(T3,[T5,T6,T7],AND)
```

- 4) Further serial decomposition. This time the user chooses to decompose T5 into two tasks T8 and T9:

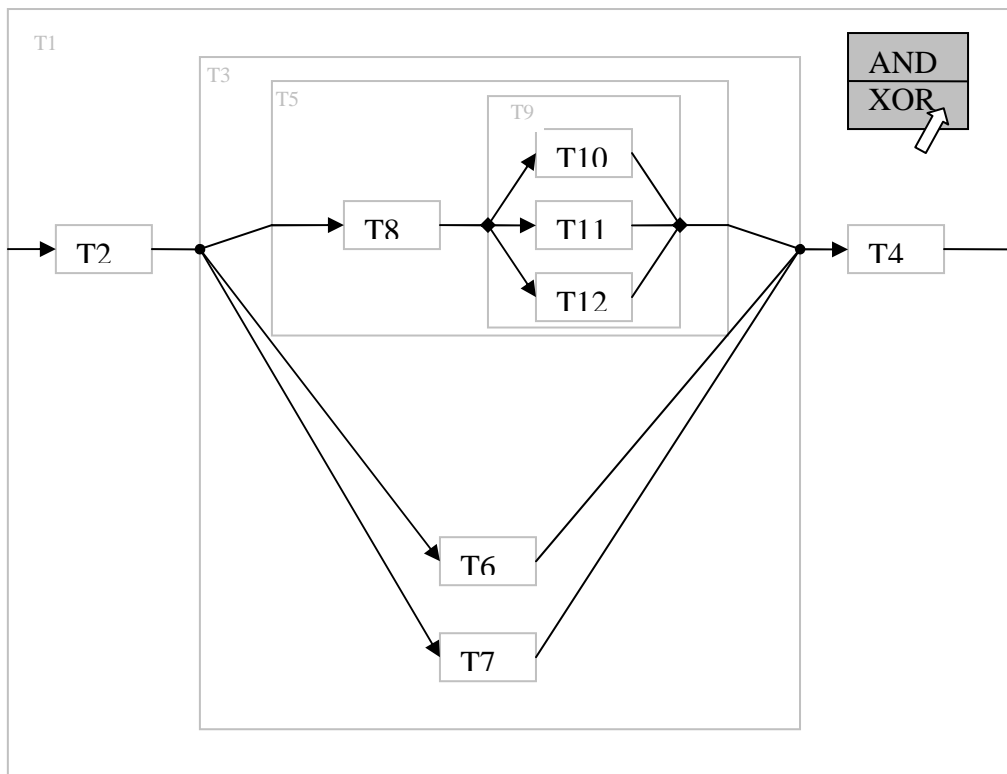
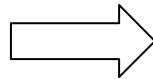
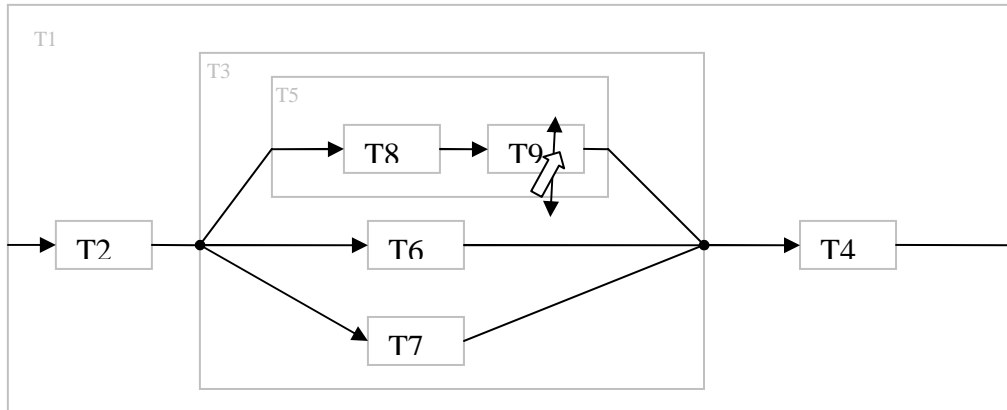
<sup>2</sup> To save space, I use n-ary predicate instead of unary for specifying activities and tasks.



Declarative format transcription:

```
Task(T1,T2,T3,T4,T5,T6,T7,T8, T9)
Decomposition(T1,[T2,T3,T4],AND)
Precedence(T2,T3)
Precedence(T3,T4)
Decomposition(T3,[T5,T6,T7],AND)
Decomposition(T5,[T8,T9],AND)
Precedence(T8,T9)
```

- 5) Alternative decomposition – As stated earlier, the process is similar to that of parallel decomposition, except in the pop-up menu XOR is selected indicating that the user wishes for an alternative decomposition, not parallel. Here the user chose to decompose the T9 task into three tasks T10, T11 and T12. When the T9 task should be performed in the original workflow, exactly one of the tasks T10, T11, T12 will be executed in the new workflow:



Declarative format transcription:

**Task(T1,T2,T3,T4,T5,T6,T7,T8, T9, T10, T11, T12)**

**Decomposition(T1,[T2,T3,T4],AND)**

**Precedence(T2,T3)**

**Precedence(T3,T4)**

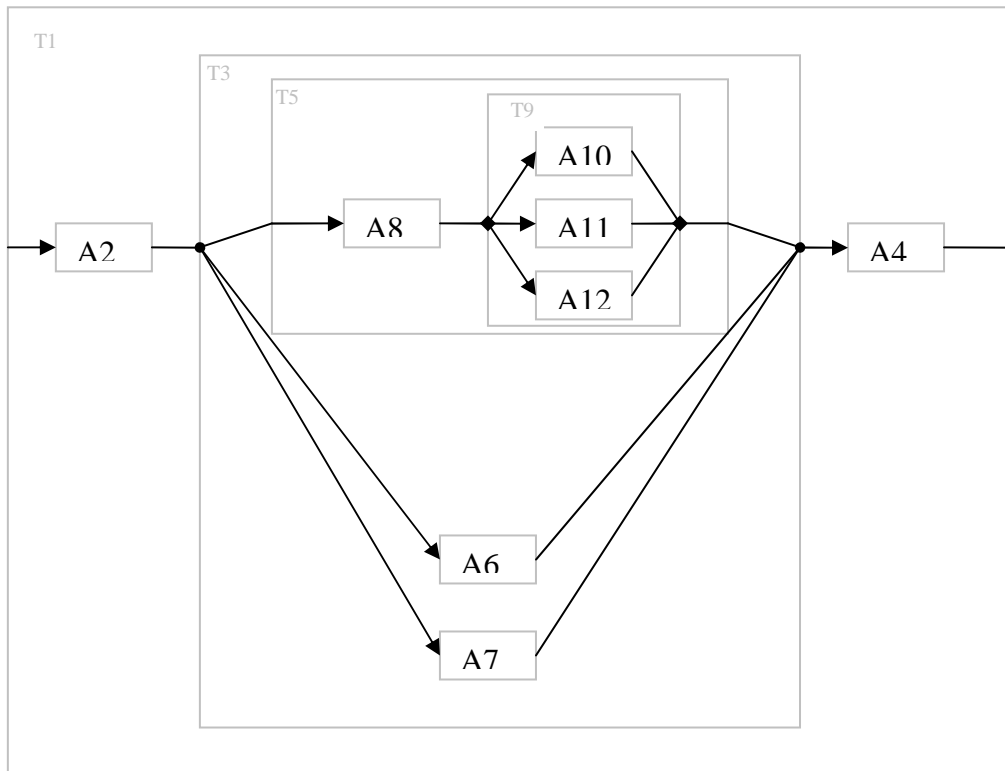
**Decomposition(T3,[T5,T6,T7],AND)**

**Decomposition(T5,[T8,T9],AND)**

**Precedence(T8,T9)**

**Decomposition(T9,[T10,T11,T12],XOR)**

After this, the user would still have to fill the empty tasks (in this case T2, T4, T6, T7, T8, T10, T11, T12) with activities to be able to schedule this workflow. When he does so, it will look like this:



Declarative format transcription:

```

Task(T1,T3,T5,T9)
Activity(A2,A4,A6,A7,A8,A10,A11,A12)
Decomposition(T1,[A2,T3,A4],AND)
Precedence(A2,T3)
Precedence(T3,A4)
Decomposition(T3,[T5,A6,A7],AND)
Decomposition(T5,[A8,T9],AND)
Precedence(A8,T9)
Decomposition(T9,[A10,A11,A12],XOR)
  
```

In other words: whenever the user fills a task with a single activity (as opposed to decomposing the task), the identifier of the task will be replaced in declarative format transcription wherever it appears by the identifier of the inserted activity.

### Another way to do editing actions

An alternative way to perform decomposition will be through right clicking on a task the user wishes to decompose and selecting “Decompose this task” option from the pop-up menu that appears. This will produce a dialog allowing the user to enter the number of tasks that should be in the nest and type of decomposition. This option is mainly intended for large-scale decomposition, where dragging would not be convenient for the user.

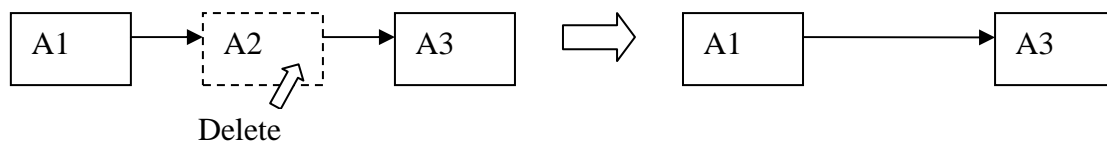
For the same reason, there will also be a special dialog for creating precedence, logical and synchronization constraints. This dialog will allow the user the enter IDs of two tasks that should be connected by an arc (and of course type of the arc) to make connecting arcs that are far away from each other more convenient.

### Removal of activities and tasks

The user can remove the selected activities within one task by pressing the delete key or by selecting the right option from right-click pop-up menu. Once an activity is deleted, the task which contained it becomes empty again – it can be filled by another activity or decomposed further.

Tasks can also be deleted – this deletes everything within them (all tasks, activities) – a dialog will appear requesting user confirmation. For every deleted task, all incident arcs are automatically deleted too.

In case the user removes a task within a sequence, it is removed together with (two) incident arcs. If the removed activity had two neighbors, a new precedence arc is created to connect them as in the following figure.



### Hierarchy

The user will be able to work with the natural hierarchy induced by the Nested TNA model. Since every decomposition creates a nest within decomposed task, we can see a tree-like hierarchy with the initial task in its root and elementary tasks in its leaves. The user will be able to show or hide any subset of the nests he created, as well as expand a single nest into new window as a separate workflow, edit it and save it either into the original workflow in place of the expanded nest, or as a completely new workflow.

### Process description

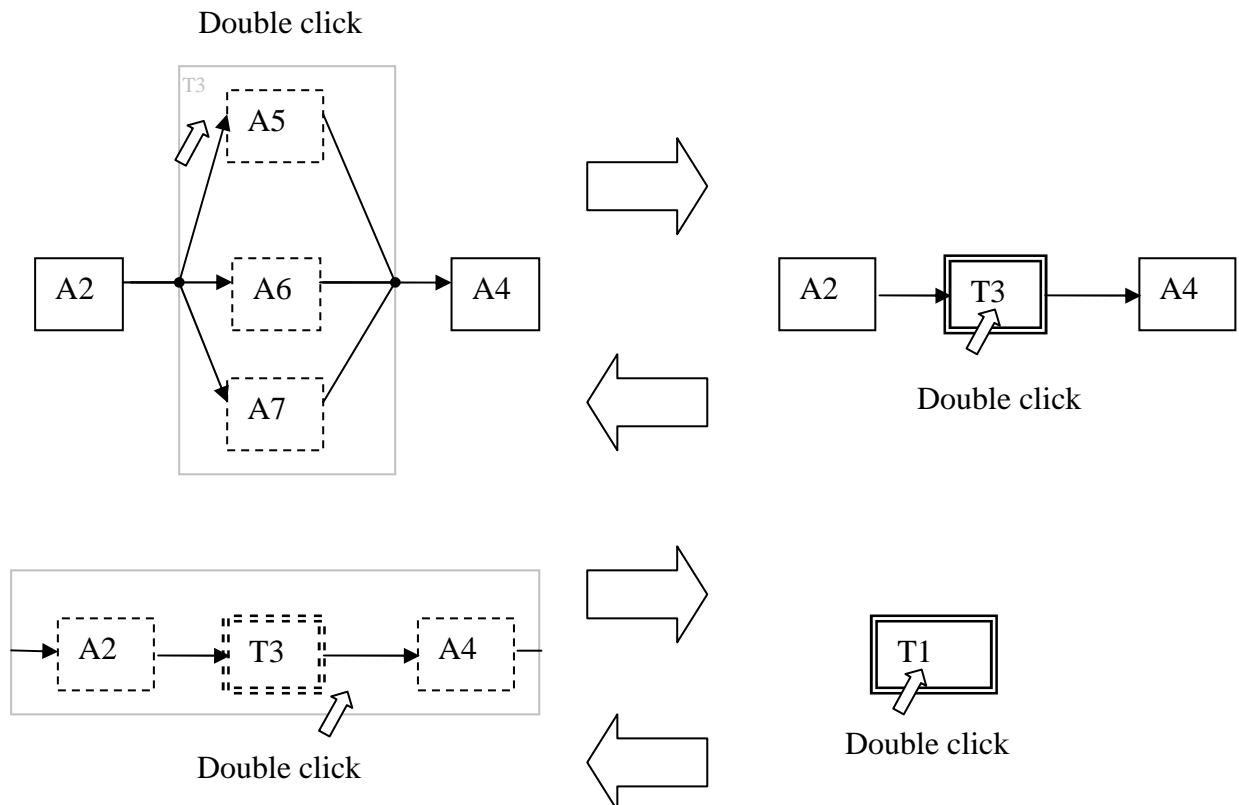
When the user moves mouse pointer over any task, all other tasks belonging into the same nest are highlighted automatically. Upon double clicking on the parent task, the nest is collapsed into the parent task, marked in a special way indicating that it is not an empty task, but a composite one containing a collapsed nest of tasks. Upon double clicking on a collapsed composite task, it is once again replaced by the nest of tasks within it. This process can be applied arbitrary number of times, giving two extremes: fully expanded workflow, where every activity and every task is visible and fully collapsed workflow, where only the initial task is displayed.

The user can also right click anywhere in a nest (or on a composite task) and select “Expand in a new window” option to open a separate window containing just the selected nest.



### Example

In this example case, the user created a network consisting of 5 activities A2, A4, A5, A6 and A7. Activities A5, A6 and A7 form a nest decomposed from task T3, so upon double clicking anywhere in T3's nest, it is collapsed as seen in the first figure (arrow to the right). On the other hand, when T3 is collapsed, it can be expanded again by double clicking on it. The second figure illustrates the same thing one hierarchy level higher – since A2, T3 and A4 are also a nest created from T1 (the initial task), the user can collapse this nest too. Upon double clicking within T1's nest, it is collapsed and only the initial (now also composite) task T1 is displayed. Double clicking on it will now expand this task's nest.



### Manipulating tasks

It will also be possible to do the following:

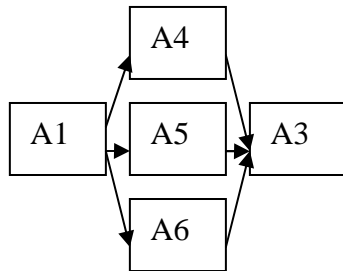
- Add more tasks to a nest – right click + “Add more tasks to this nest”. A dialogue appears where the user may enter a number of tasks to be added.
- Change the order of tasks in a nest (“swap” tasks) – changes the execution order in case of sequence, for parallel / alternative decomposition it is only for user's convenience. It will probably be done by using a special “cursor mode” for swapping tasks within the same nest.

### Padding

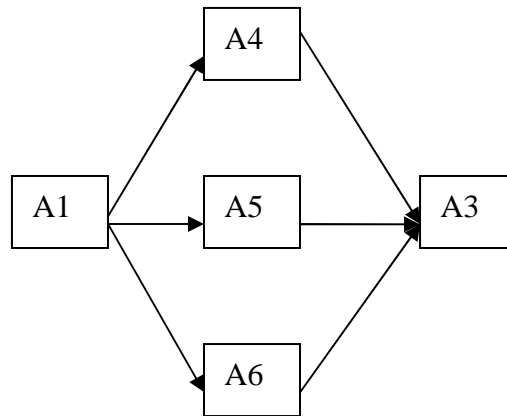
The user will be able to set the horizontal and vertical padding between tasks, i.e. how much space is there between tasks on the screen.

Example:

Low padding

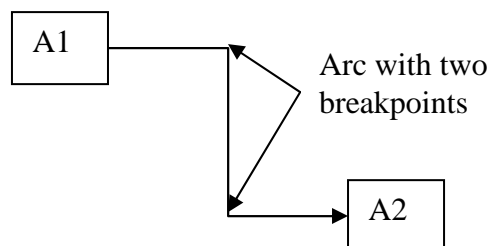
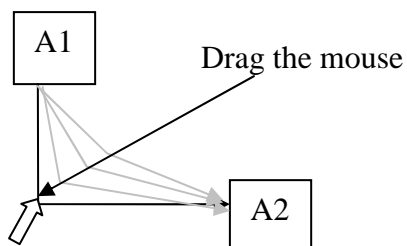
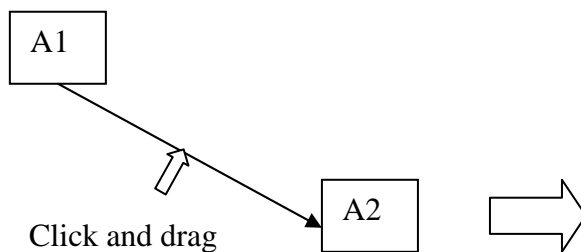


Higher padding



### Transforming arcs

The user will be able to transform arcs which were not added through decomposition by adding active breakpoints on them like so:



## ***Import/Export of the workflow***

The user will be able to import a workflow created in the freelance mode of the MAKE project, as long as it satisfies the Nested TNA model (this will be checked by this module).

## ***Data model***

This module will pass data only to and from the MAKE database. Said data will be workflows saved in specific format described below. It will be an XML format specified by a XSD schema (provided in a separate file).

Note that the data model is used throughout the whole FlowOpt project, not only by the workflow editor module.

I will list all data objects used by the workflow editor module below, each of them will translate into an XML element. This is a description of the format on semantic level, for complete and exact syntactic description of this format, please refer to its XSD schema. For sake of this definition, let the term “Item” refer to either an activity, or a task. Also, the symbol  $\infty$  means unconstrained upper bound – this means that semantically, there is no upper limit on the value.

- **Workflow** – represents entire workflow (root element of the XML file) with its ID and name. Within a workflow, following objects can be defined:
  - **Activity** – represents a single activity with its properties – an ID, name, duration and cost. There can be  $0 \dots \infty$  activities within a workflow.
  - **Task** – represents a task, which only holds its own ID and name. There can be  $1 \dots \infty$  tasks defined within a workflow.
  - **Decomposition** – represents a decomposition of one task into a nest of items. There has to be at least one item in the nest. The decomposition is of given type, which can be one of AND or XOR.
  - **Precedence arc** – represents a precedence constraint between two items.
  - **Logical arc** – represents a logical constraint between two items. The logical relation is of given type, which can be one of implication, equivalence, mutual exclusion or inclusive or.
  - **Synchronization arc** – represents a synchronization constraint between two items. The synchronization is of given type, which can be one of start-start, end-end or end-start.
  - **Resource dependency** – represents a requirement an activity has on resources – a single primary one and arbitrary number ( $0 \dots \infty$ ) of alternative ones. There can be ( $0 \dots \infty$ ) resource dependencies in a workflow.

## ***Other non-functional requirements***

For sake of this section, let *application* refer to the Workflow editor of the FlowOpt project.

## ***Security***

The application won't provide any form of security. Any data used in the application may be read by third parties. No authentication or authorization will be provided – the application will not distinguish between users and all users will have the same rights within the application.

## User Documentation and localization

All textual messages of this application will be in English, same as all the documentation, which will include developer documentation, user documentation (manual), installation guide and context hints within the application itself.

## Robustness

Robustness will be guaranteed in the sense that no action performed within the application will corrupt data it works with, provided that this data was valid in the first place. However, the application will not try to recover any data that isn't valid (if an error is detected in an input XML file, it will only be announced to the user, not repaired or modified in any way).

## Testability

Along with the application, a test suite will be provided in a standard framework. This test suite will cover all the public interfaces of the application. Graphics' functionality will be verified by providing graphical prototypes that will demonstrate this functionality.

## Extensibility

The application won't provide any SPI for plug-in purposes. It will only be extensible on the source-code level. No part of the application is to be further used as a separate library.

## Maintainability

The application will provide its own automatic installer, which will allow the user to install both the MAKE application and all the modules of the FlowOpt project at once. The application will not create any non-temporary files on the user's computer, nor will it edit the registry in any way.

## Performance and scalability

The application has to be scalable so that it functions well even for large workflows. For workflows containing less than 1000 activities and arbitrary number of arcs, it has to be usable on a PC with HW requirements specified earlier in this document.

In any point, the user will not have to wait for application's response for more than 10 seconds (except in case of HW failures, DB connection failures or other errors which can't be influenced by the application).(TBD).

In case the database is accessed remotely over a WAN, there will be no guarantees on speed.

## Usability

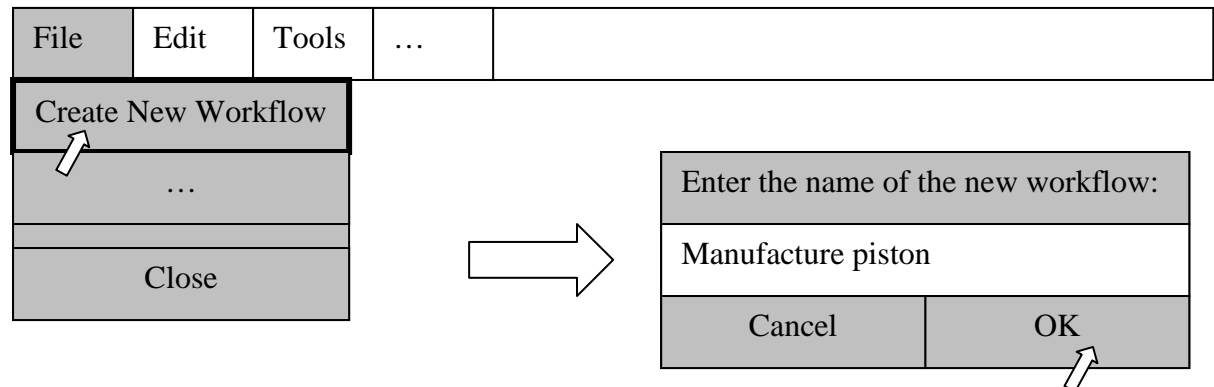
The application will provide the following mechanisms to achieve user-friendliness and ease of use:

- A user manual accessible directly from the application
- Context guides for all the GUI controls, which will appear in tooltips after a short mouse-hover
- Several examples of workflows and the editing process

## Appendix: User experience scenario

Here is an example of how the user would work with the application. Suppose the goal of the user is to create a workflow that describes the manufacturing process of a piston. Here are the steps the user would take to accomplish this:

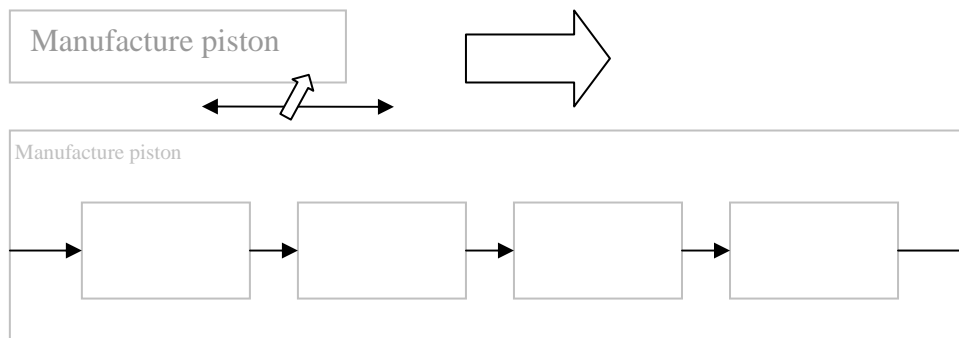
- 1) Creating a new workflow – the user chooses the File->New Workflow option from the main menu and enter a name for the new workflow (in this case “Manufacture piston”).



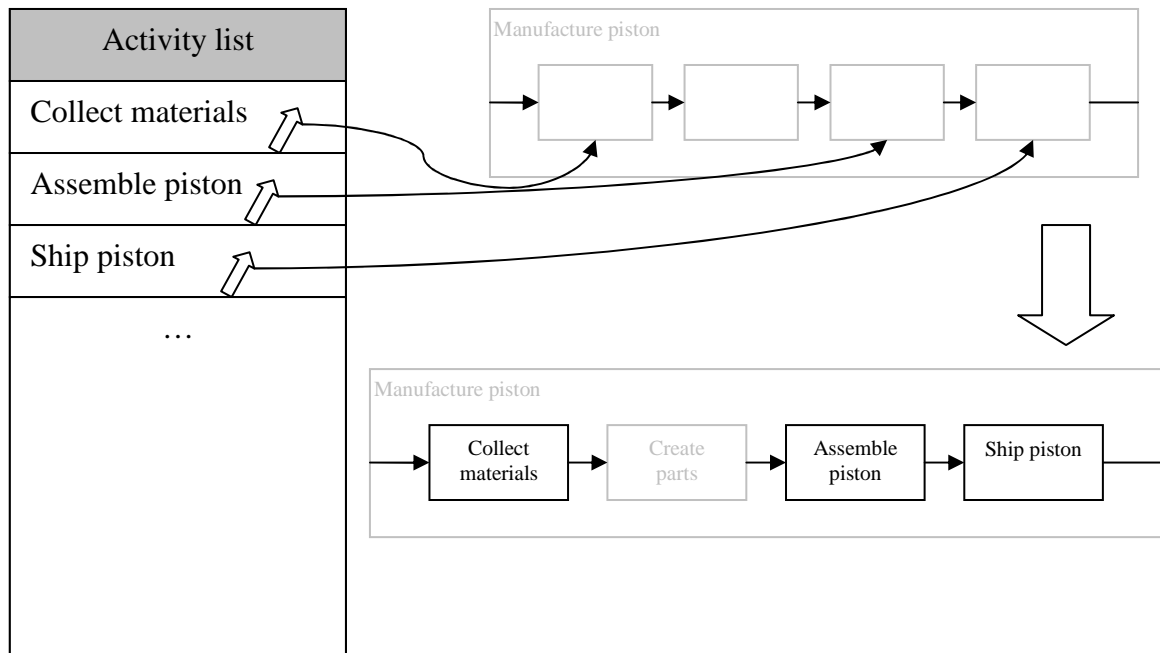
- 2) This will create a new workflow consisting of a single empty task (the initial task). This task will represent the entire workflow that is to be created. Therefore, its name will be the same as name of the workflow. Tasks will be rendered gray.



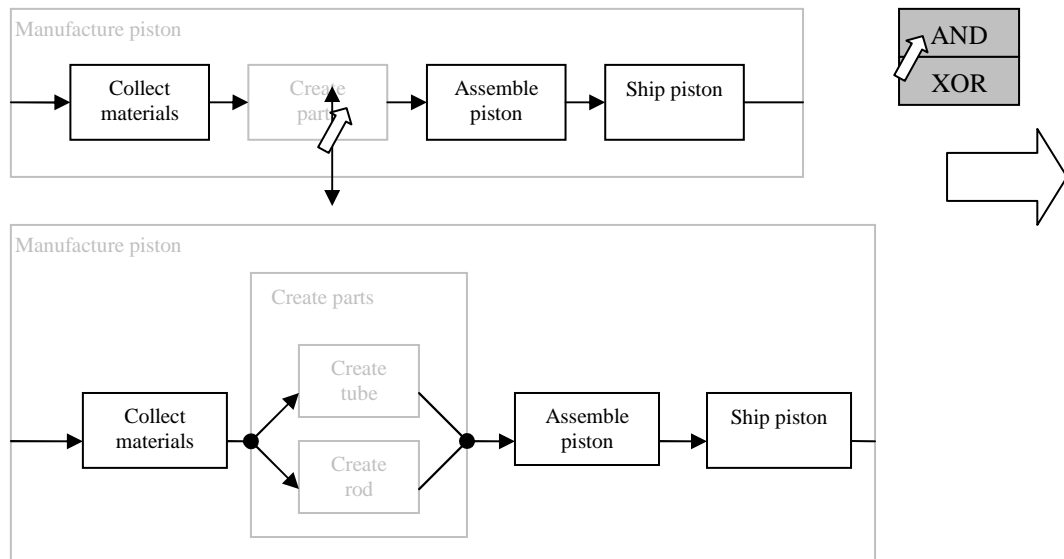
- 3) To build the workflow (i.e. create new tasks), the user has to use decomposition. First off, to manufacture a piston is to first collect the necessary materials, then create parts of the piston (one tube and one rod), then assemble the parts and in the end, ship the piston. This means that the task Manufacture piston will be decomposed using serial decomposition into four child tasks – Collect material, Create parts, Assemble parts and Ship piston. This is done by left clicking on the parent task and dragging the mouse **horizontally** to adjust the number of the child activities (we need four). Please note that this is an abstract example. I am aware that some of these activities may be handled outside the workflow editor in the Make environment (like shipping) and therefore would not require an activity within the workflow, this workflow is just for demonstration purposes.



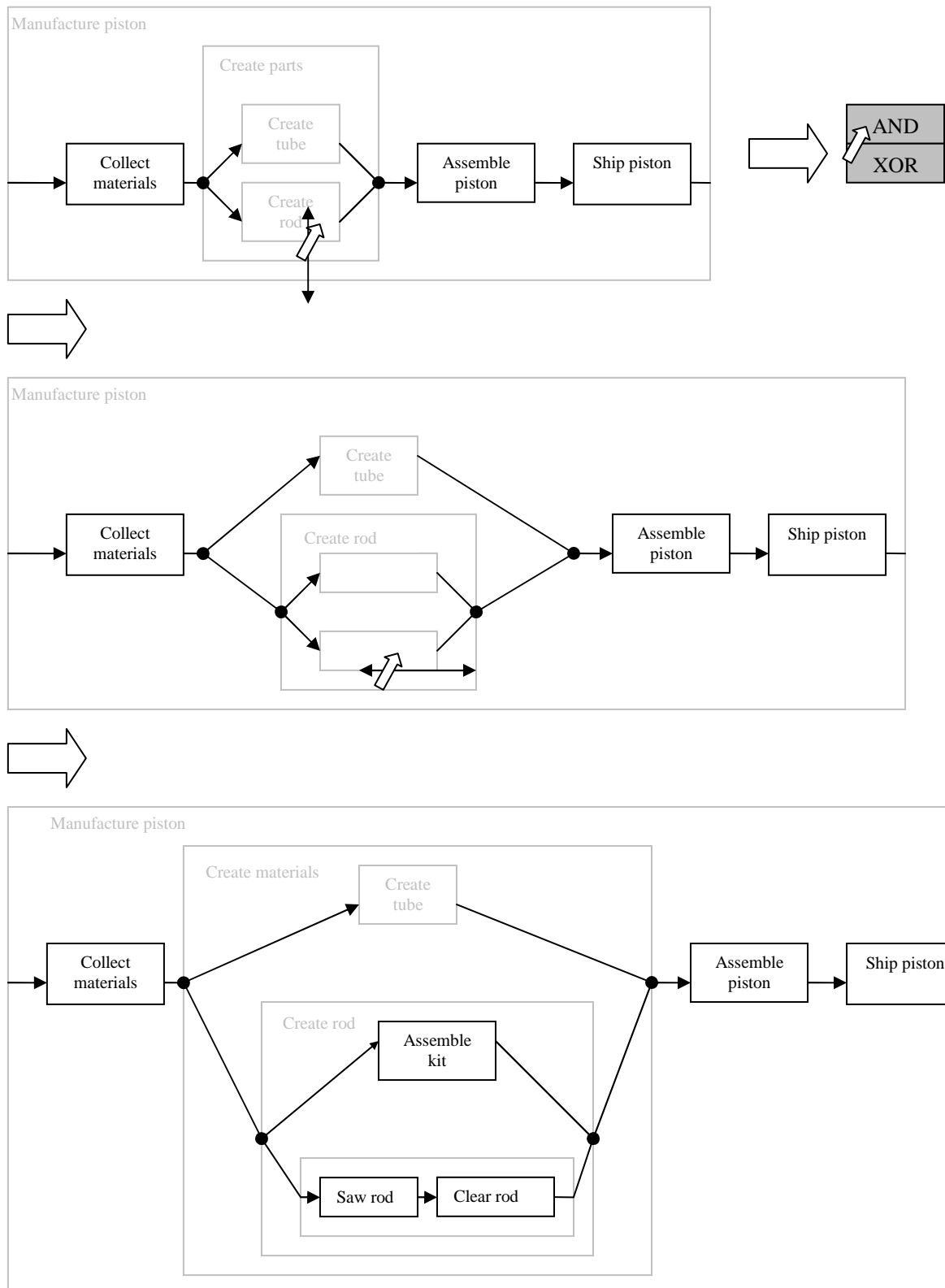
- 4) Now we have decomposed the task Manufacture piston into four child tasks – all of these are still empty. As said earlier, they represent the need to first collect materials, create parts of the piston, assemble them and in the end ship the piston. Let us say that collecting materials, assembly and shipping are all elementary activities. Therefore we select them from the activity list and drag them on their respective tasks. These are therefore no longer empty tasks - they have an activity in them. Activities will be rendered black to distinguish them from tasks. Activities will be rendered black to distinguish them from tasks. Creating parts of the piston however is not an elementary activity, so further decomposition will be needed. For now, let us just give the corresponding task a name to indicate its purpose.



- 5) Next step is specifying how to create parts of the piston. For a piston, we need a tube and a rod. These may be manufactured separately and in parallel, so we decompose the task Create parts into two tasks: Create rod and Create tube. Parallel decomposition is done by clicking on the parent task and dragging the mouse **vertically** to adjust the number of activities in the nest. In this case we need two. Upon releasing the mouse, a pop-up window appears where we specify whether we want parallel or alternative decomposition – here we want parallel.



- 6) Now we need to define how to create a tube and how to create a rod. These may in fact be viewed as separate workflows on their own. Therefore, it is possible to create a separate workflow for example for creating a tube and then insert this workflow into the task Create tube. Note that due to consistency of our workflow model, any (non-empty) task is a valid workflow, so this operation is well defined. One may want to create a separate workflow for multiple reasons, one of them being reuse of the created workflow. Another reason to creating a separate workflow may be simplification of design – the user can focus only on creating the tube and once this is defined, insert it in the larger workflow. This is classic modularity principle like you see in programming languages etc. To demonstrate, we will define the task Create rod in place of the original workflow and the task Create tube as a separate workflow, which we then insert into the original workflow.
- 7) To create a rod, we need to first saw it, and then clear it. Also, we need a special kit that must be assembled first before it can be welded onto the tube, however this can be done in parallel with the welding and sawing. Therefore, we decompose the Create rod task into two tasks in a parallel way. We fill one of them with the Assemble kit activity (let us again assume that this is an elementary activity). The other task we decompose into two using serial decomposition and fill the resulting two tasks with activities Saw rod and Clear rod respectively. All these actions can be done in the same way as before – decomposition through dragging the mouse on the parent task and assigning an activity by dragging it from the activity list to the particular task.

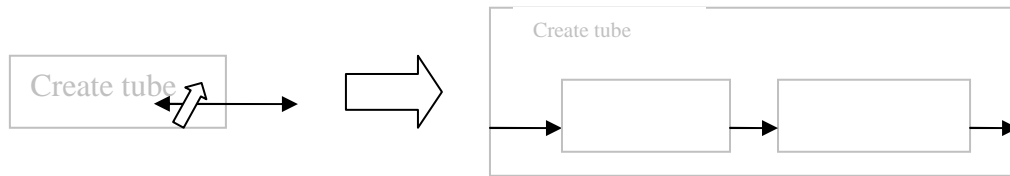


- 8) Last thing to be done is defining how to create a tube. As stated above, we do this not in the original workflow (as we did with the task Create rod) but rather in a separate new workflow to demonstrate this functionality.  
We first create a new workflow (see step 1) and call it Create tube. This creates an



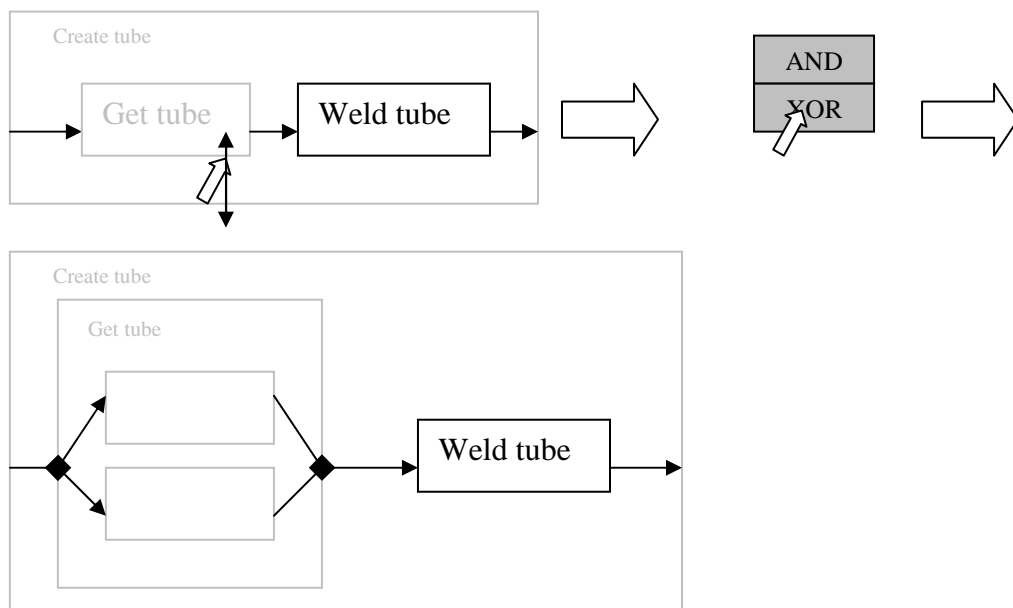
empty initial task called Create tube. We know that to create a tube, we can either buy a tube, or make it ourselves, which involves sawing the tube and clearing the tube. No matter whether we buy the tube or make it ourselves, we have to perform some welding on it to make it fit for use in the piston.

This means that we first decompose the Create tube task into two in a serial way – first we have to get the tube somehow, then we must weld it.

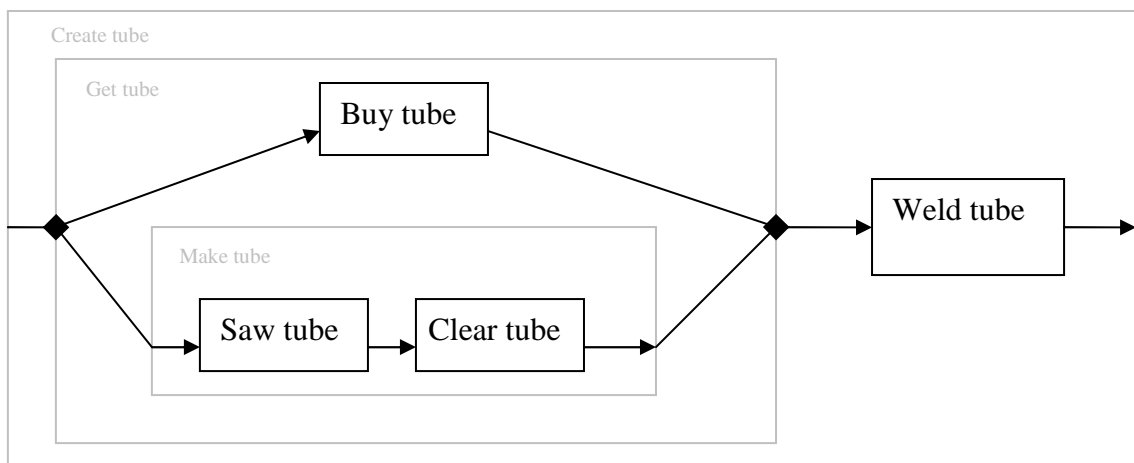
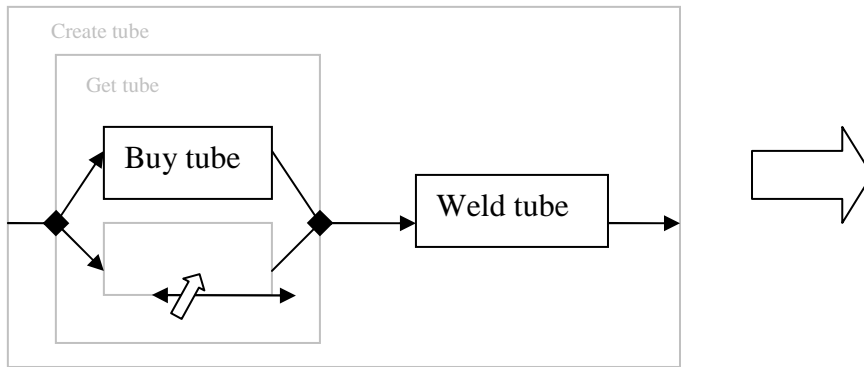


- 9) Now we know that the final welding of the tube is an elementary activity, so we place in the respective task.

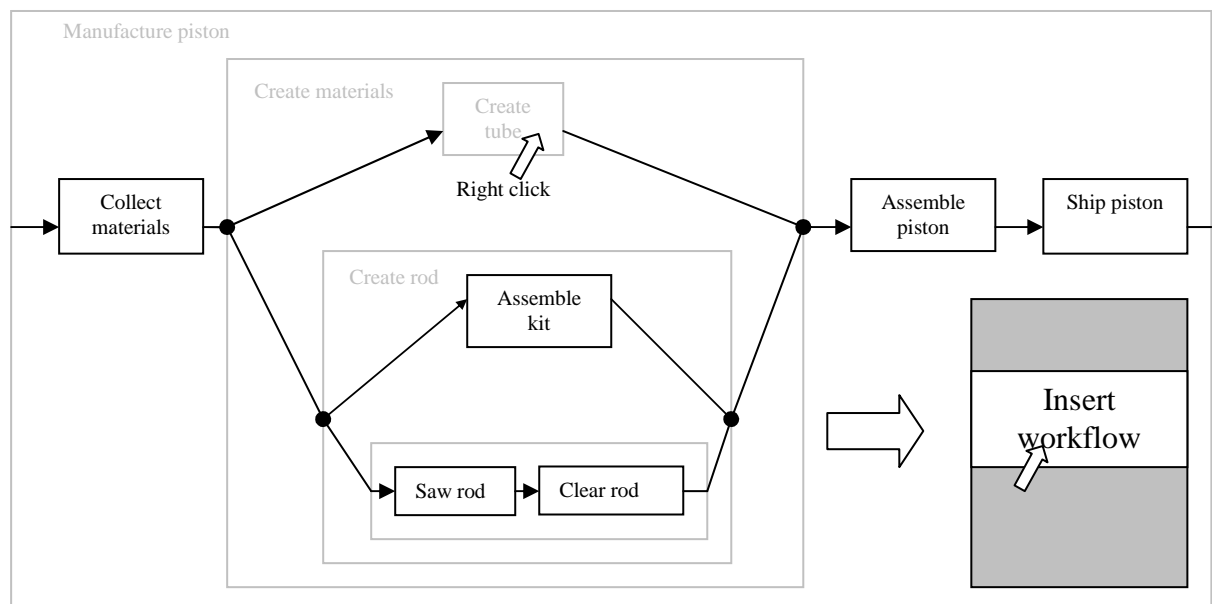
Also, to get the tube we can either buy it or make it ourselves – a classic example of alternative decomposition. This is done very similarly to parallel decomposition – we again click on the parent task (here Get tube) and drag the mouse vertically to adjust the number of tasks in the nest (here we need two). The only difference from creating parallel decomposition is that in the pop-up menu that appears upon releasing the mouse button, we choose XOR this time.

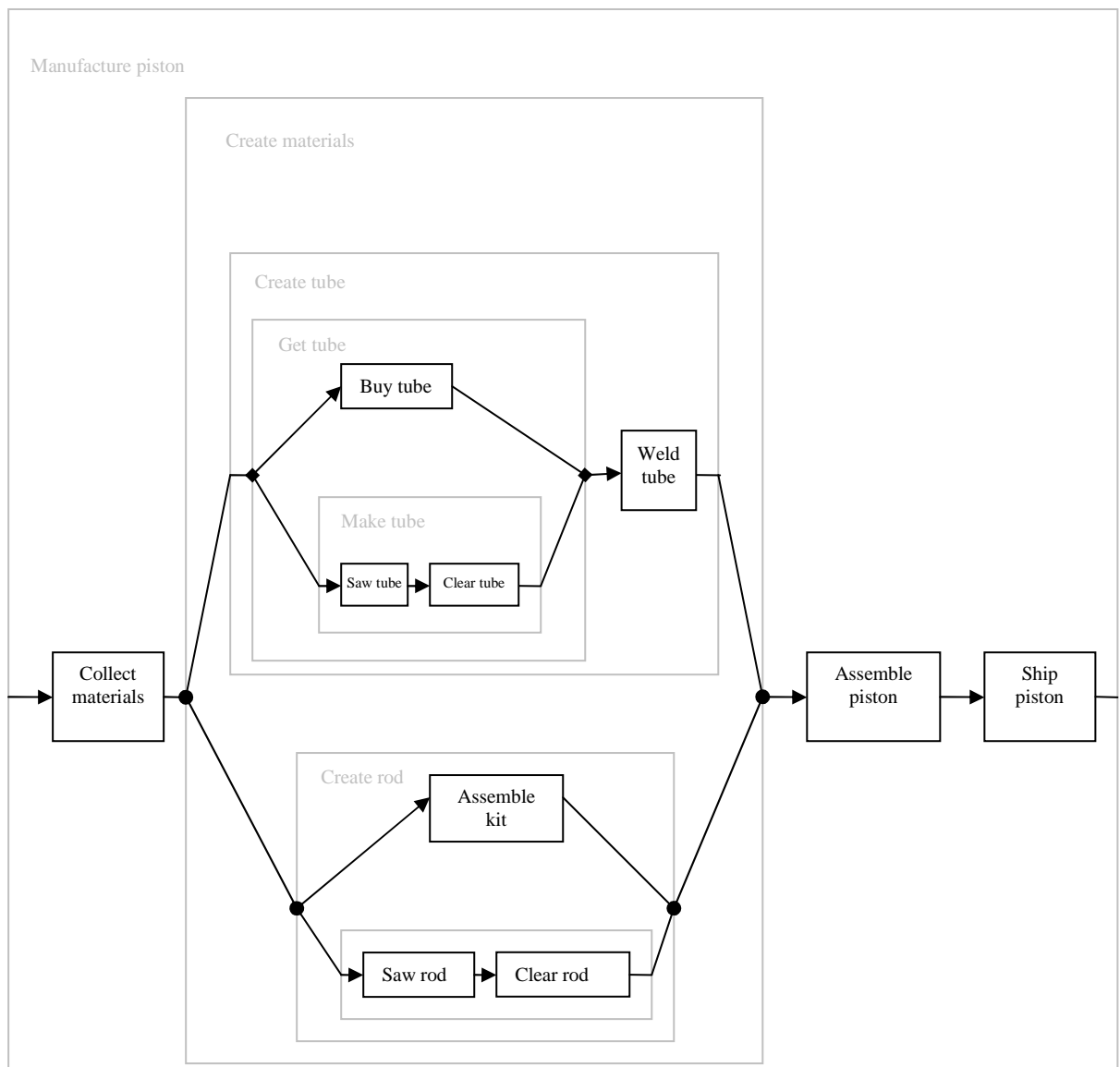
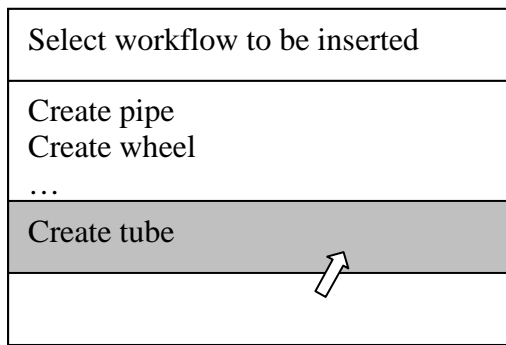


- 10) Now to get the tube, one way is to buy it. This is an elementary activity, so we once again just drag it from the activity list into the particular task. The other way is to make the tube ourselves, which means first sawing it, then clearing it. Those are both elementary activities, so we decompose the other task in a serial way into two tasks and we fill them with said activities.



11) Here we have created a separate workflow just for making tubes. We now want to insert this workflow into the larger one for making pistons. We therefore open the Create piston workflow again, right click on the Create tube task and select Insert workflow from the pop-up menu. A dialog will appear where the user will select the workflow to be inserted (here it is the Create tube workflow). This will complete our workflow for making pistons, which is now valid and may be scheduled.





## Table of Contents

Purpose .....	2
Used terms and remarks .....	2
Font conventions .....	2
Various remarks .....	2
Application overview .....	2
Hardware and software requirements .....	3
Operating Environment .....	3
Workflow model .....	3
Nested TNA model description .....	3
Domain model .....	3
Tasks and Activities .....	3
Precedence, Logical and Synchronization arcs .....	5
User interface .....	6
Zooming .....	6
Undo .....	6
Selection .....	6
Activity and arc properties .....	7
Resources .....	7
Choice of orientation .....	7
Editing .....	8
Description of editing process .....	11
Another way to do editing actions .....	16
Removal of activities and tasks .....	16
Hierarchy .....	16
Process description .....	16
Manipulating tasks .....	17
Padding .....	18
Transforming arcs .....	18
Import/Export of the workflow .....	19
Data model .....	19
Other non-functional requirements .....	19
Security .....	19
User Documentation and localization .....	20
Robustness .....	20
Testability .....	20
Extensibility .....	20
Maintainability .....	20
Performance and scalability .....	20
Usability .....	20