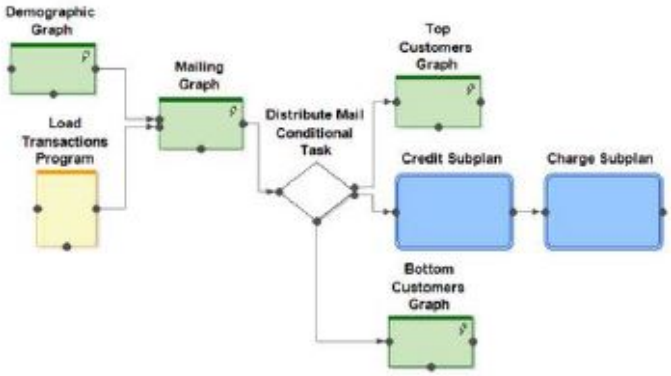
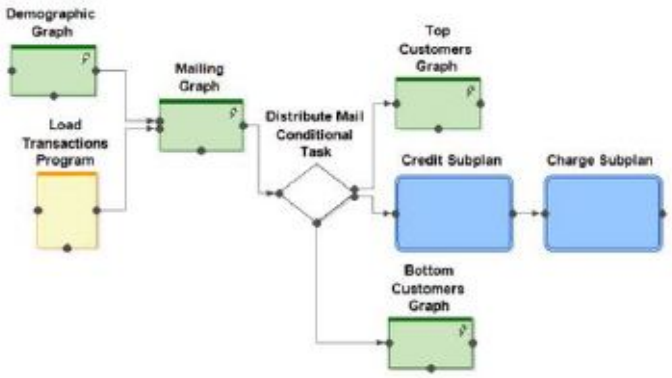
Conduct>It

For Ab Initio Plan Developers



Conduct>It

For Ab Initio Plan Developers



Preface: Confidential and Proprietary

NOTICE

This document contains confidential and proprietary information of Ab Initio. Use and disclosure are restricted by license and/or non-disclosure agreements. You may not access, read, and/or copy this document unless you (directly or through your employer) are obligated to Ab Initio to maintain its confidentiality and to use it only as authorized by Ab Initio. You may not copy the printed version of this document, or transmit this document to any recipient unless the recipient is obligated to Ab Initio to maintain its confidentiality and to use it only as authorized by Ab Initio.

Contents: Conduct>It Training Contents

- Introduction
- Demonstration
- Tasks
- Methods
- Failure and Recovery
- Parameters
- Subplans and Loops
- Resources
- Conclusion

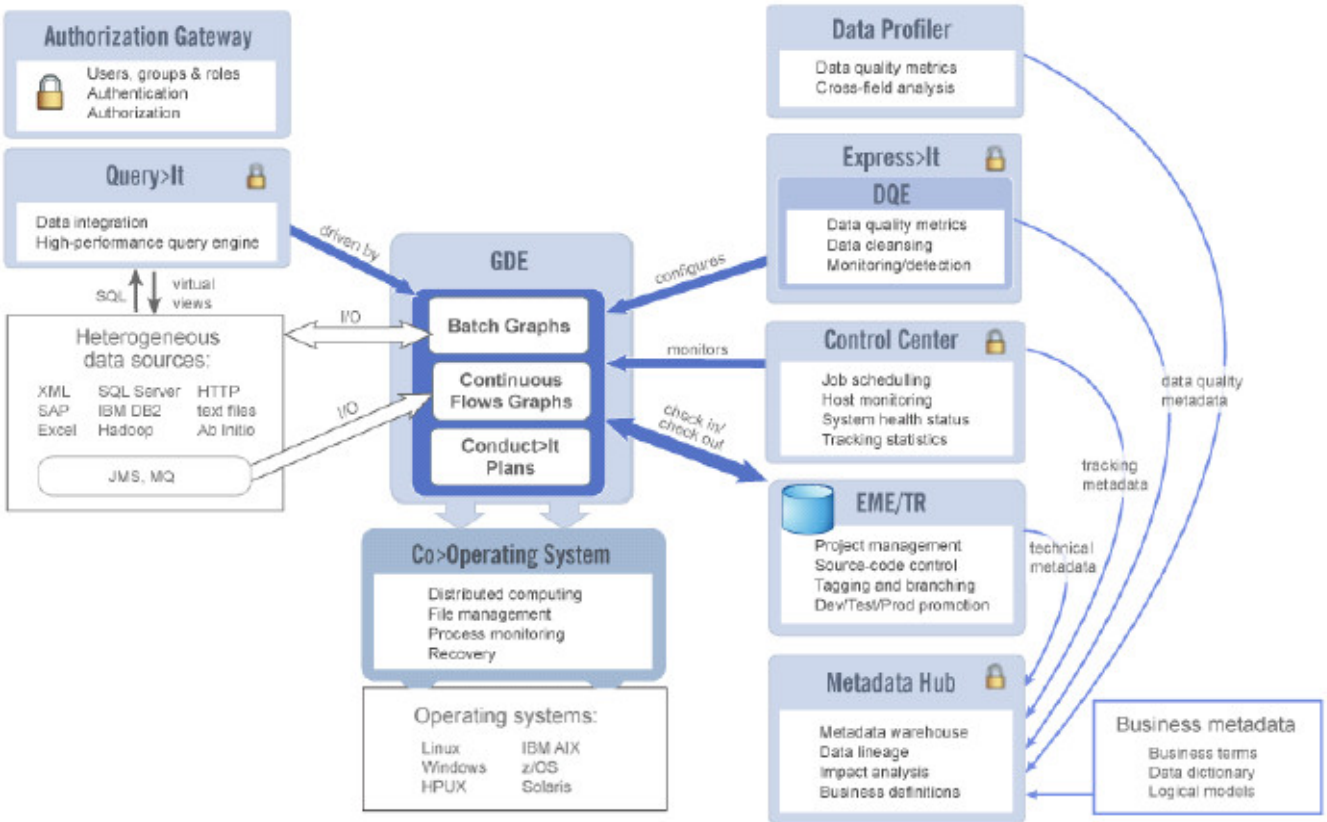
Introduction

An Overview of Conduct>It

Introduction: Section Topics

- What is Conduct>It?
- Why use Conduct>It?
- Introduction to Plans

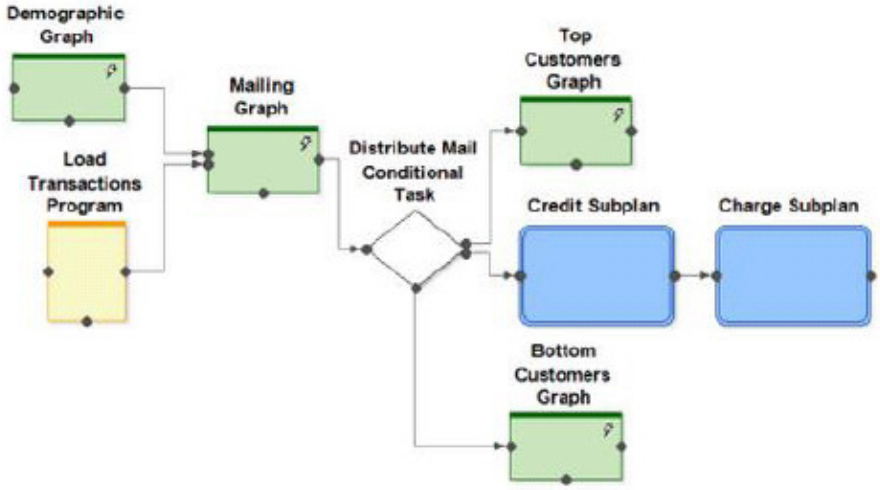
Ab Initio Product Spectrum



Introduction: What Is Conduct>It?

Conduct>It is an Ab Initio software product for building applications that coordinate the activity of multiple graphs, third-party programs, and user-defined scripts.

These higher-level applications are called ***plans***.

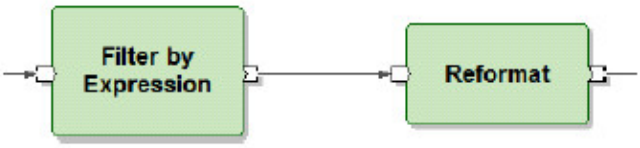


Introduction: Why Use Conduct>It?

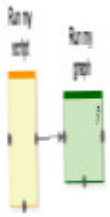
- Ease of development
- Maintainability
- Reusability
- Recoverability
- Resource Management
- Graphical Development Environment (GDE)

Introduction: Graphs –vs- Plans

In a **graph**, a connection between *ports* defines a *data flow*.



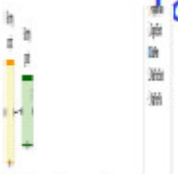
In a **plan**, a *branch* between two task *connection points* defines a *dependency between the two tasks*.



This task cannot run until the prior task is finished.

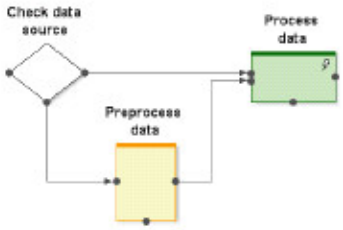
Introduction:

How do plans allow you to coordinate tasks?

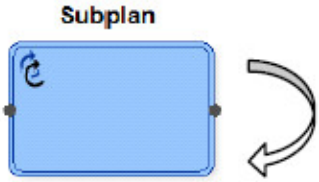


Task dependencies

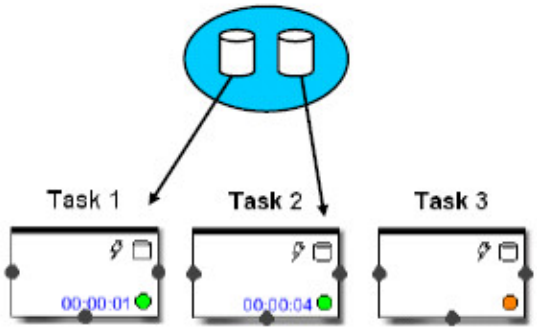
Event Signaling



Control flow



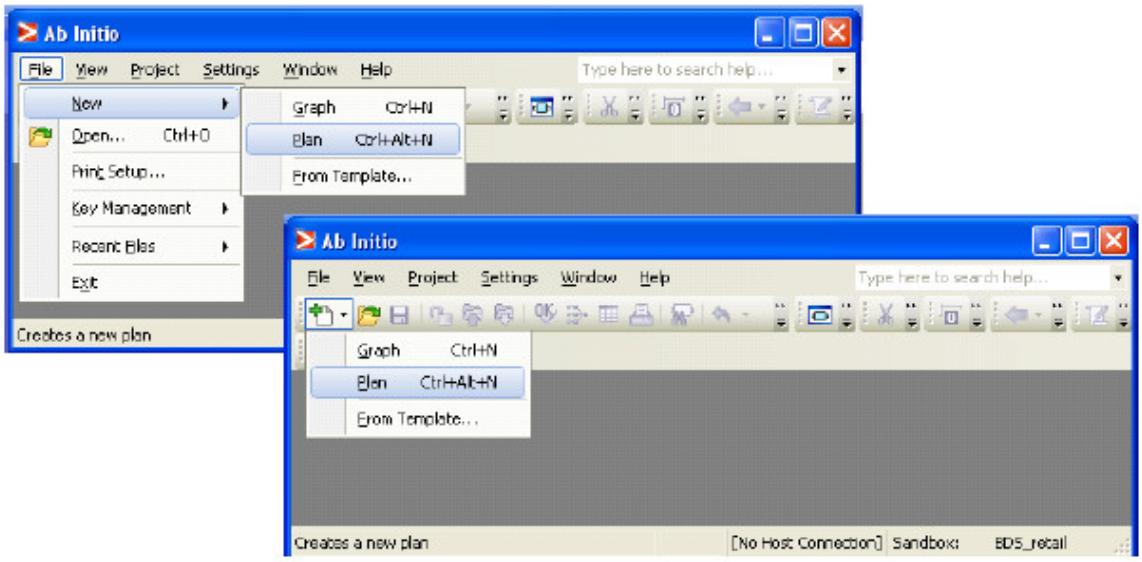
Looping Subplans



Resource Pools

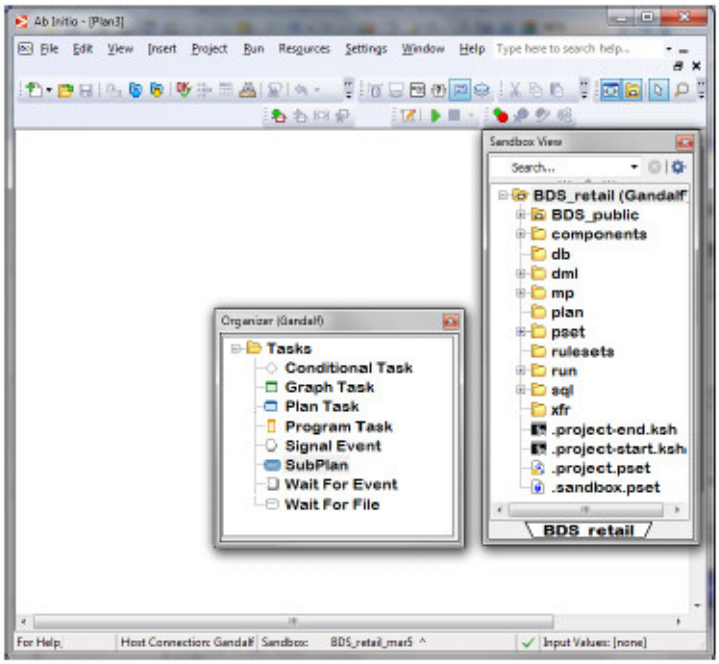
Introduction: Plans are implemented in the GDE

Making a new plan is as simple as choosing File > New > Plan from the GDE menu.



Introduction: Plans in the GDE

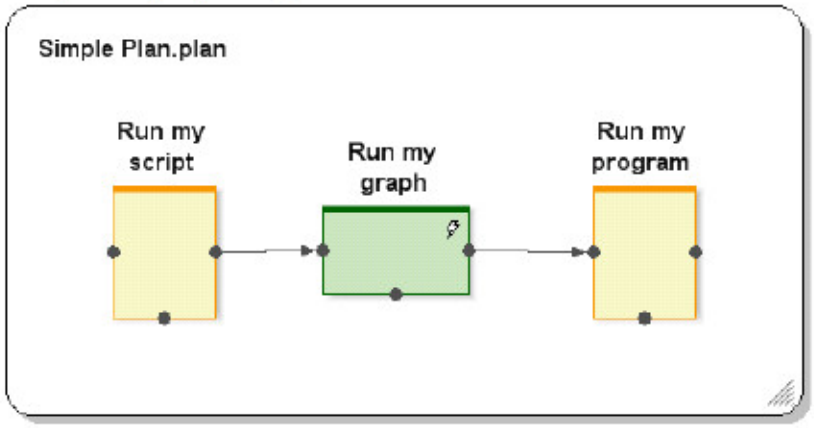
Drawing a plan is like drawing a graph



- Blank canvas for drawing plans
- Component Organizer with built-in Conduct>It components
- Sandbox, including “mp” and “plan” directories

Introduction: Introduction to Plans

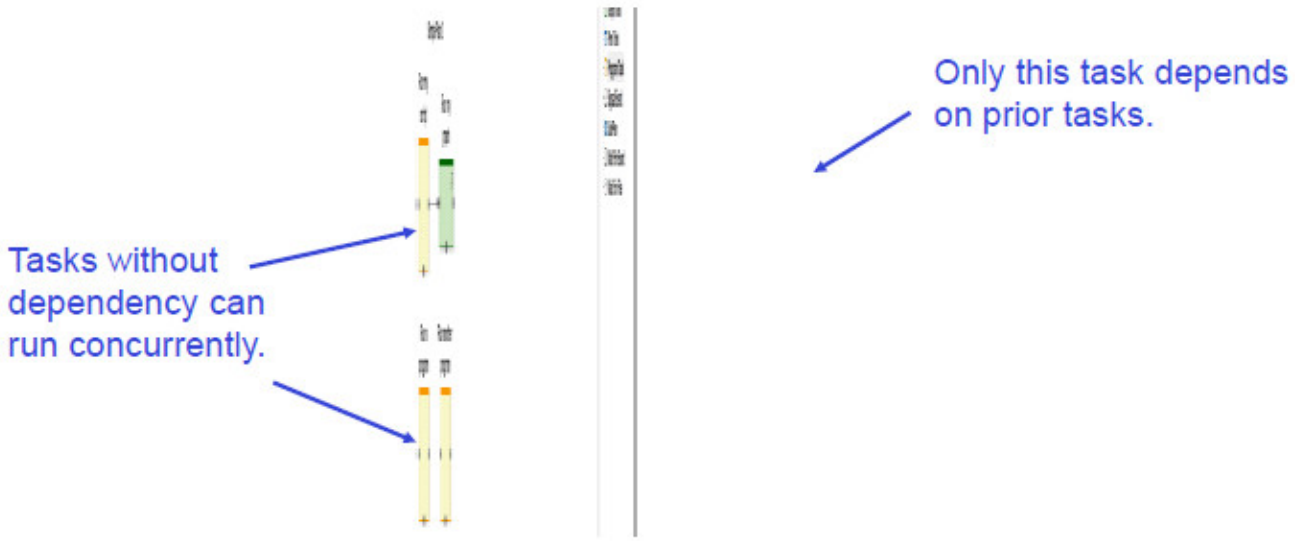
Plans can be collections of simple sequential tasks, made up of Ab Initio graphs, custom scripts, and third-party programs.



Plans execute Left to Right

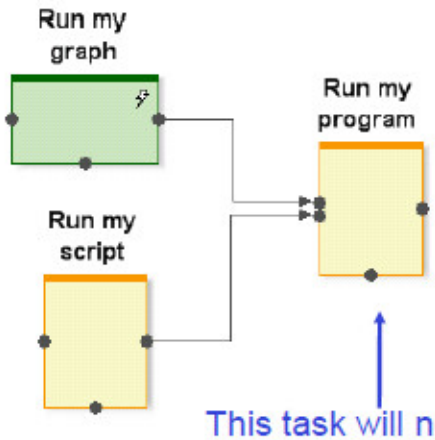
Introduction: Introduction to Plans (cont'd)

Plans can also contain *non-sequential* tasks…



Introduction: Introduction to Plans (cont'd)

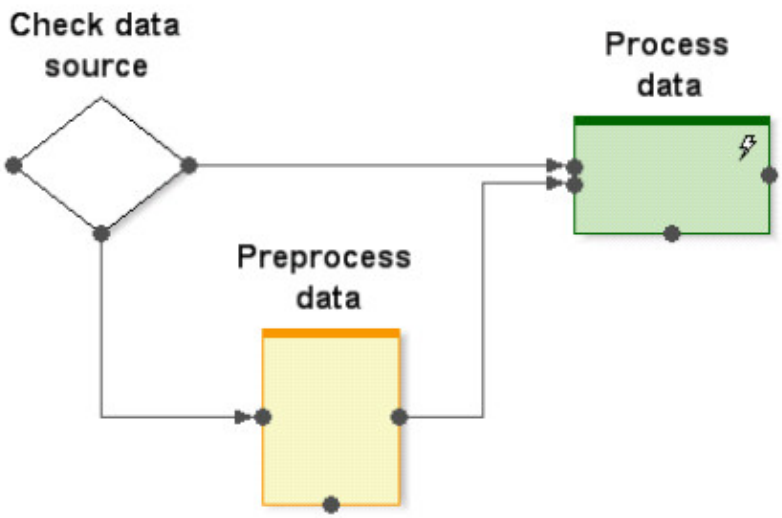
When a task is dependent on multiple upstream tasks, it will not run until all of those upstream tasks either complete or are disabled.



This task will not run until "Run my graph" and "Run my script" complete or are disabled.

Introduction: Introduction to Plans (cont'd)

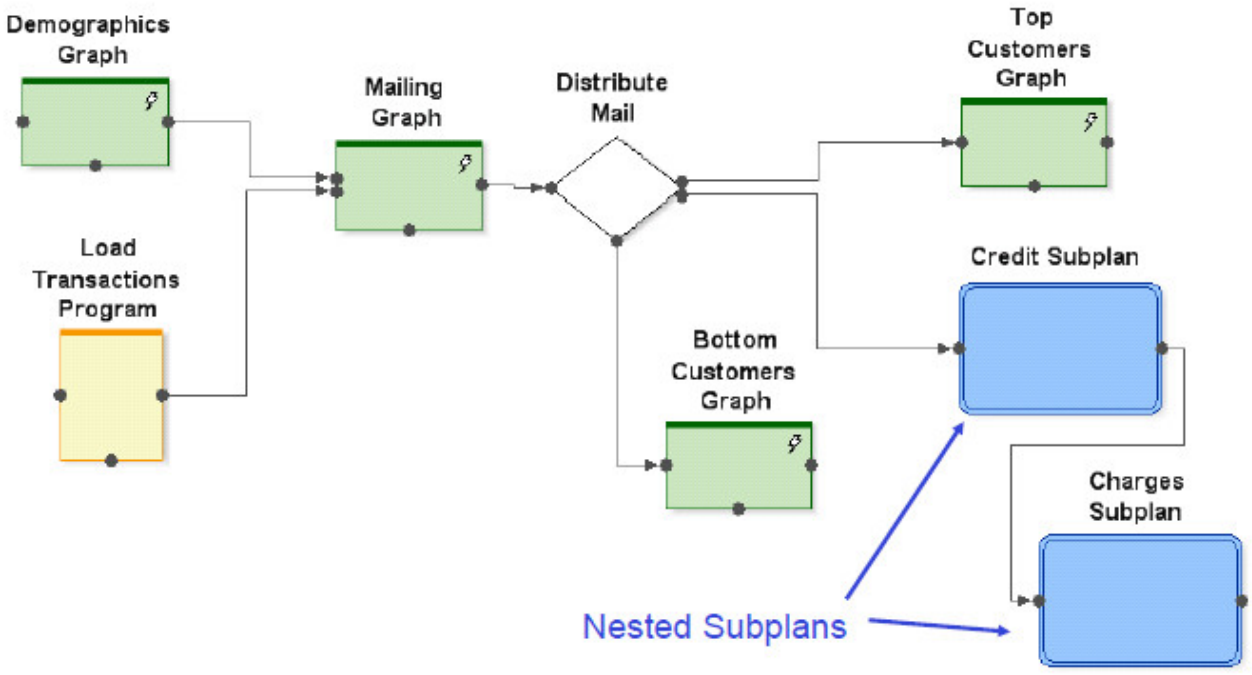
Plans can include *conditional logic*.



Different actions are taken depending on data, parameter values, or the processing environment.

Introduction: Introduction to Plans (cont'd)

Complex plans can also include *nested subplans*.



Introduction: Typical Conduct>It Use-Cases

- Waiting for a file to show up before triggering the execution of a Graph
- There are multiple interdependent graphs, scripts, programs, and plans. Some tasks can only run if other tasks succeed first.
- You want a collection of graphs to either succeed as a group or fail and recover as a group.
- A graph should only be run if certain conditions are met: time of day, availability of some resource, existence of a file, etc.
- Multiple graphs need to share and possibly manipulate certain parameters.

Introduction: Section Review

In this section, we discussed:

- The basic features of Conduct>It
- Why we would use Conduct>It
- The basic features of a plan

A Brief Demonstration

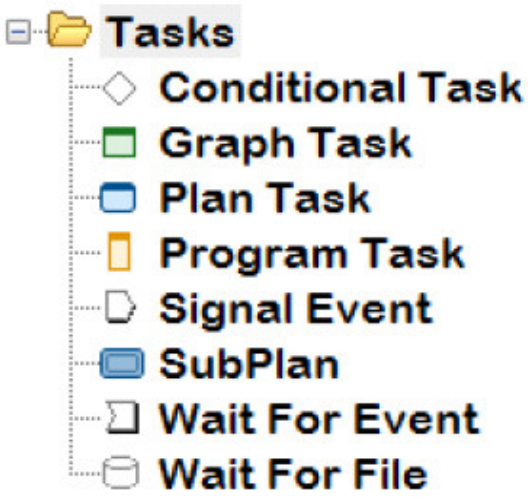
Tasks

Tasks: Section Topics

- Properties of Tasks
- Available Tasks
- Input Parameters
- PDL

Tasks: Introduction to Tasks

Conduct>It components are called **tasks**.



Tasks: Introduction to Tasks (cont'd)

Tasks that run graphs, plans, and programs:

- **Graph Task**: Runs a graph
- **Plan Task**: Runs a plan
- **Program Task**: Runs a program or script

Tasks that are event-driven

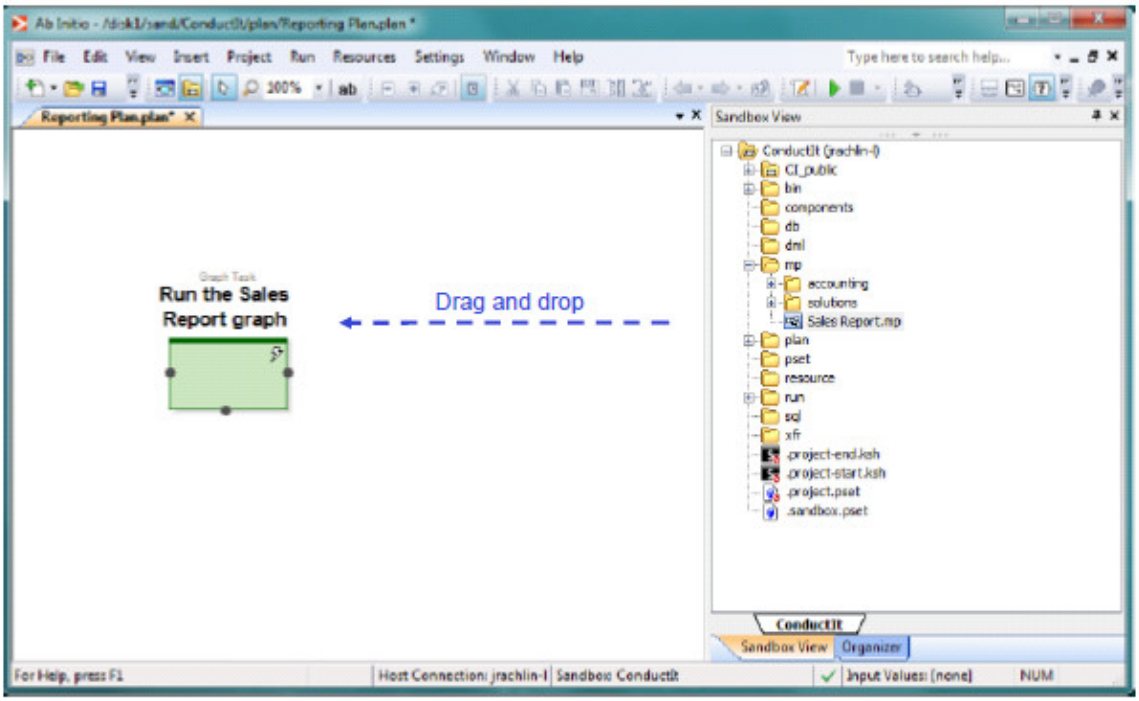
- **Signal Event**: Trigger a signal
- **Wait For Event**: Activate upon receiving an event
- **Wait For File**: Wait for a file to become available

Miscellaneous Tasks

- **Conditional Task**: Evaluates a condition
- **SubPlan**: A container for other tasks

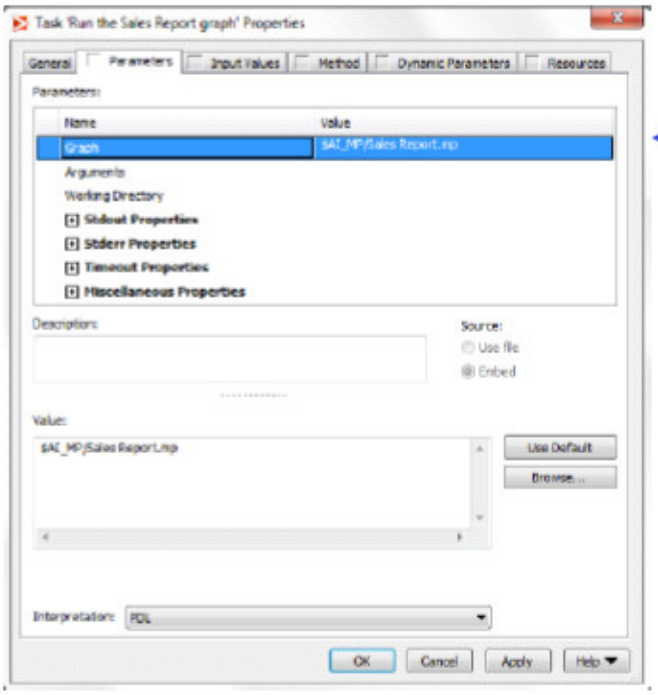
Tasks: Graph Tasks

Graph tasks run graphs. Create a graph task by dragging and dropping the graph .mp file onto the canvas



Tasks: Graph Tasks (cont'd)

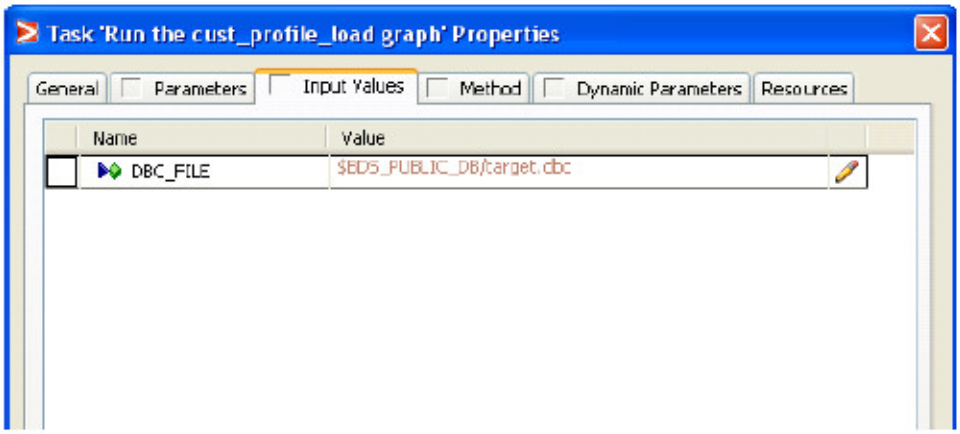
The Graph parameter points to the particular graph that this graph task runs.



Location of graph or pset

Tasks: Task Input Parameters

Look at the Graph task properties Input Values tab...

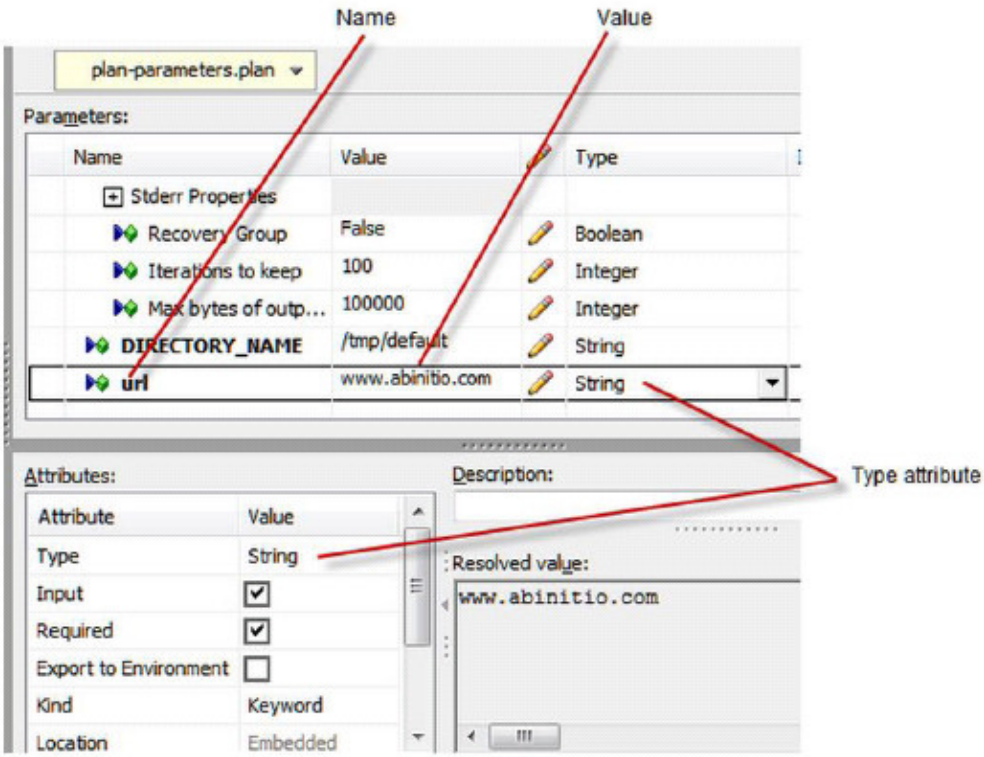


Here you specify the input parameters associated with the graph. The value can be provided using Plan parameters.

Tasks: Plan Parameters

You can define plan input parameters using the parameters editor.

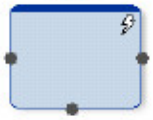
Edit > Parameters

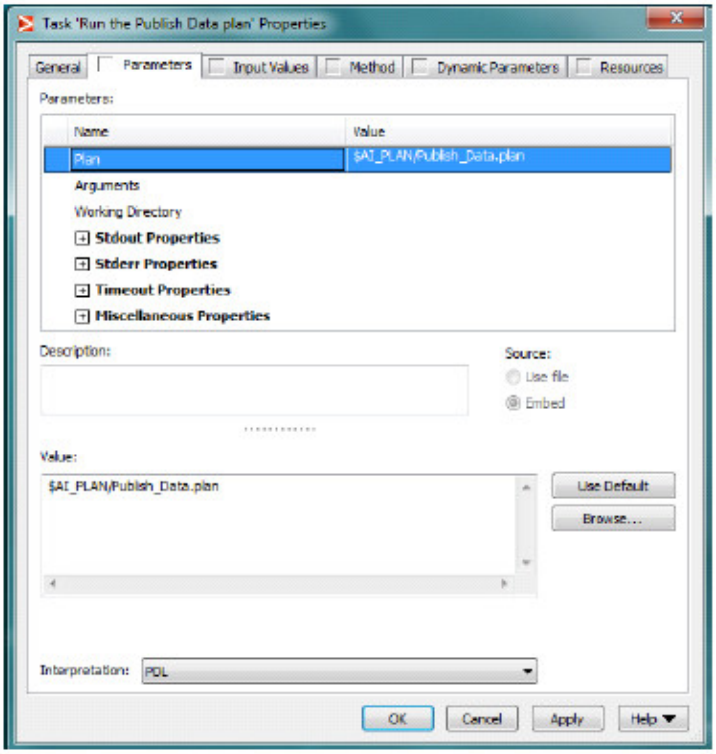


Tasks: Plan Tasks

Similar to Graph Tasks, a *Plan Task* runs a plan.

Plan Task
Run the Publish Data plan

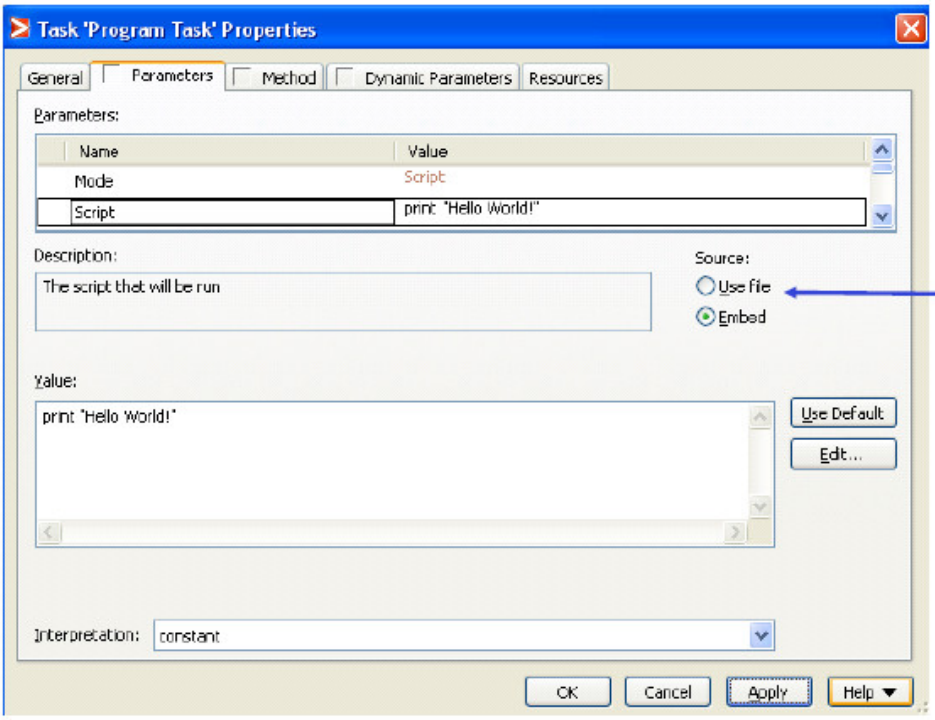




Location of the plan

Tasks: Program Tasks

Program tasks run programs and scripts.



An embedded script or the location of an executable

Setup: Setup Exercises

Setup your training environment as follows:

1. Check out the ConductIt sandbox from the EME/TR
2. From the GDE, Project > Create Data Directories (set AI_TEST_FLAG to your username)

Exercise: Exercise 1 – Simple Dependency

Build a plan that does the following:

- Runs the Check Directories script ($AI_BIN/check_dirs.ksh) with the argument "both" specified.
- After the script is complete, runs the Move and Reformat graph ($AI_MP/MoveAndReformat.mp), where the graph's input parameters are specified by input parameters defined at the plan level.

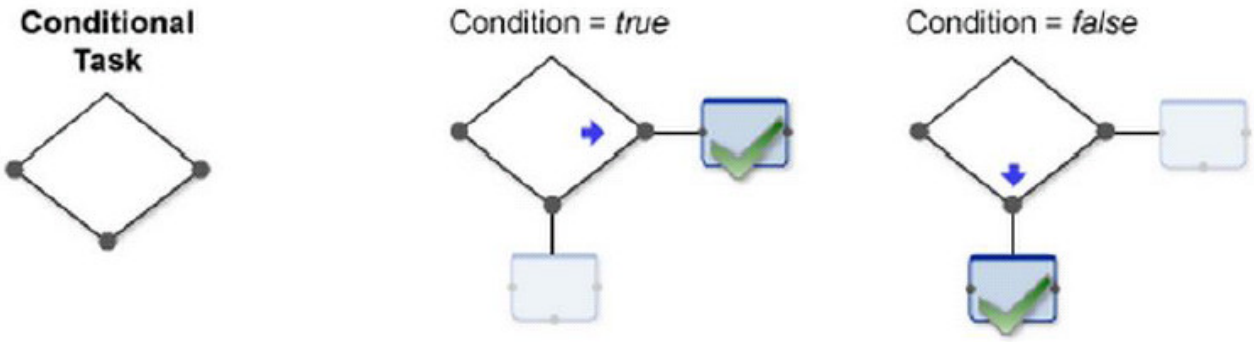
Create plan-level parameters for the following:

- The path for the input file is: $AB_HOME/examples/data/transactions.dat
- The path for the output file is: $AI_SERIAL/reformatted_trans.dat
- Make sure that the graph-level parameters use the plan parameters that you have created

Save your solution as **ex1_simple_dependency.plan**

Tasks: Conditional Tasks

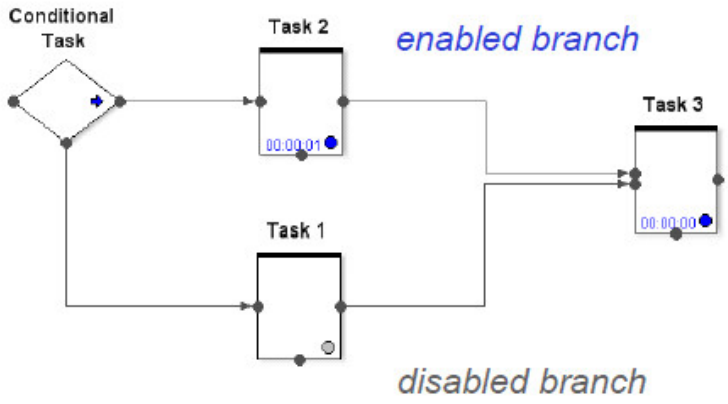
Conditional tasks evaluate a condition.



It enables one branch, and disables the other.

Tasks: Conditional Tasks

Tasks connected only to a disabled branch are *disabled*.



By default, tasks connected to **at least one** enabled branch are enabled.
This can be changed to require that **all** incoming branches be enabled.

Parameter: *Enabled branches required to run task*
Choices: Any / All

Tasks: Conditional Tasks

Conditional tasks can evaluate:

- An expression
- A program

Each has its own definition of True/False

	TRUE	FALSE
EXPRESSION	• A nonzero integer • The string **T** or **t**, or any string starting with **T** or **t**.	• The integer **0** • The string **F** or **f**, or any nonblank string that does not start with **T** or **t**.
PROGRAM	• Exit status of **0**	• Nonzero exit status

Tasks: Introduction to PDL

You can use DML functions in the expression of a Conditional task using PDL.

Parameter Definition Language

PDL (the Parameter Definition Language) is a simple but comprehensive set of notations for expressing the values of parameters in components and graphs.

Previously we have expressed parameters using a combination of:

$Substitution
${}Substitution
Constant Substitution
Shell Substitution

Tasks: Introduction to PDL (cont'd)

PDL offers you the maximum of flexibility in parameter interpretation, with a minimum of bother in specifying which kind of interpretation you want.

Both $ and ${ } interpretation are part of PDL — you simply use either one when you need to.

In addition, PDL allows you to use inline DML within parameter definitions to perform almost any operation or calculation you want.

Tasks: Introduction to PDL (cont'd)

Inline DML Expression:

- Enclosed in $[ ]
- Can make use of DML functions
- Can reference previously defined parameters.

The parameter names are not prefixed with the **$** symbol within the bracketed expression.

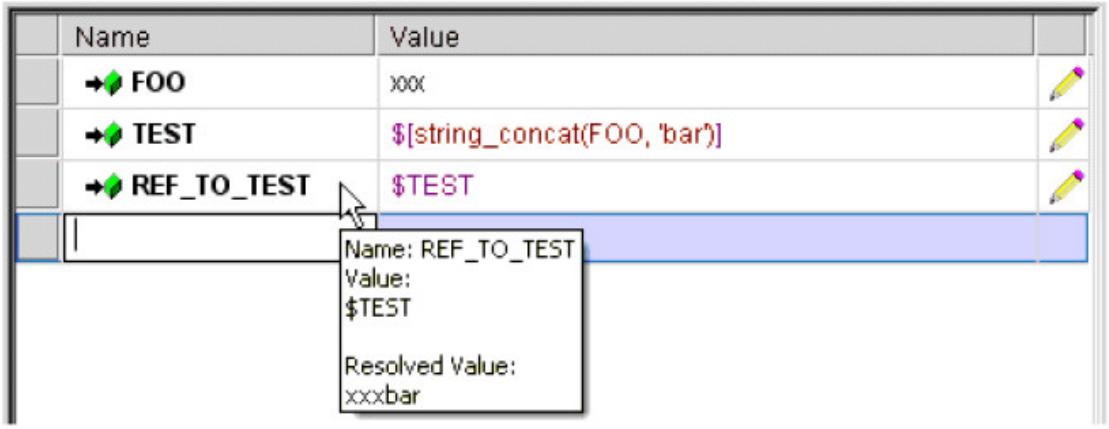
All free variables in a PDL expression must be parameters.

String constants must be in quotes.

Tasks: Introduction to PDL (cont'd)

Examples of PDL expressions

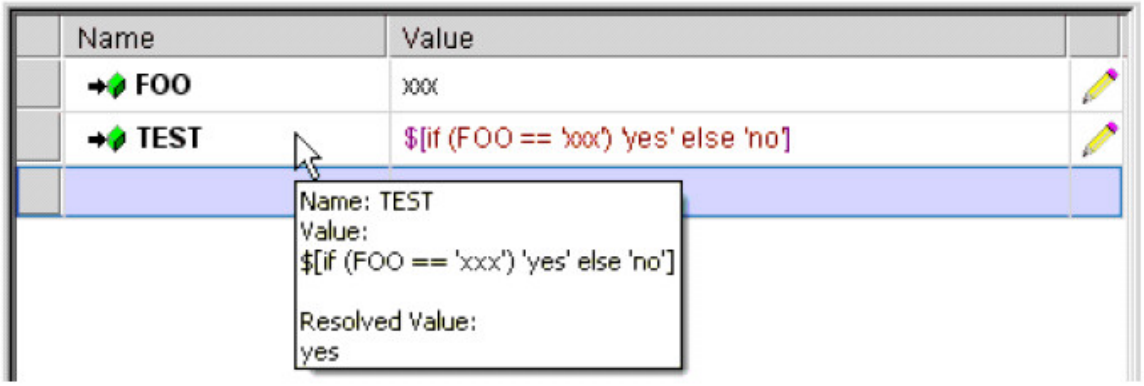
In this example, FOO is interpreted as a reference to the previously-defined parameter FOO:



Tasks: Introduction to PDL (cont'd)

Examples of PDL expressions

In this example, the value of the same parameter FOO is tested in a condition:



Tasks: Introduction to PDL (cont'd)

You can use most DML functions within inline DML expression.

For example:

- string_filter_out()
- datetime_month()
- ceiling()
- vector_sum()

Some useful functions for plans:

- directory_listing()
- file_information()
- string_split()

Exercise: Exercise 2 – PDL

In a new plan, use PDL to define a plan parameter that resolves to the current day of the month.

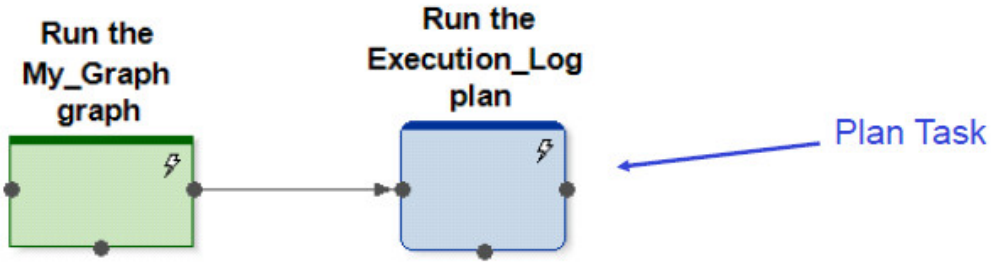
Save your solution as ex2_pdl.plan

Hint:

Co>Operating System Help -> Reference -> DML Core Function Reference -> Using the DML core functions -> Date and datetime operations using DML functions

Tasks: Plan Tasks

Plan tasks run entire plans.

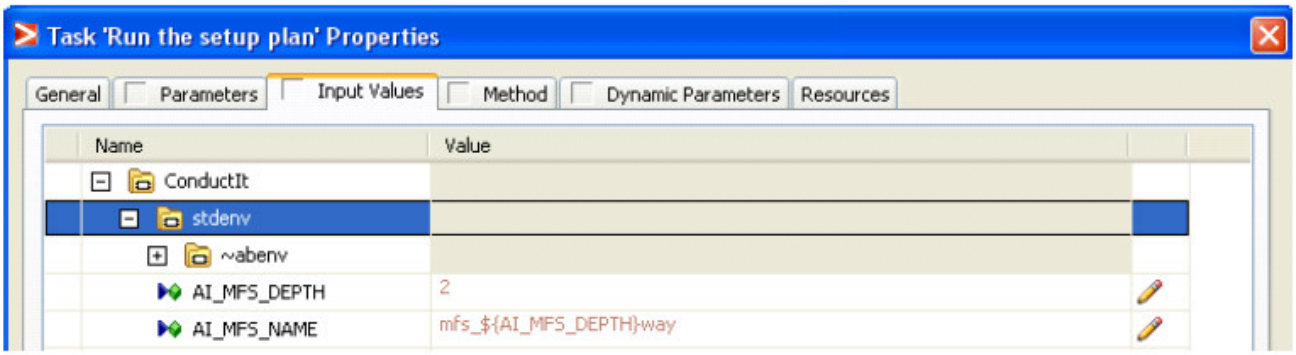


This makes it convenient to reuse plans.

Tasks: Plan Task Input Values

Like Graph tasks, input parameters for a plan are displayed on the Input Values tab.

This includes Input Parameters from included projects.



Exercise: Exercise 3 – Conditional Dependency

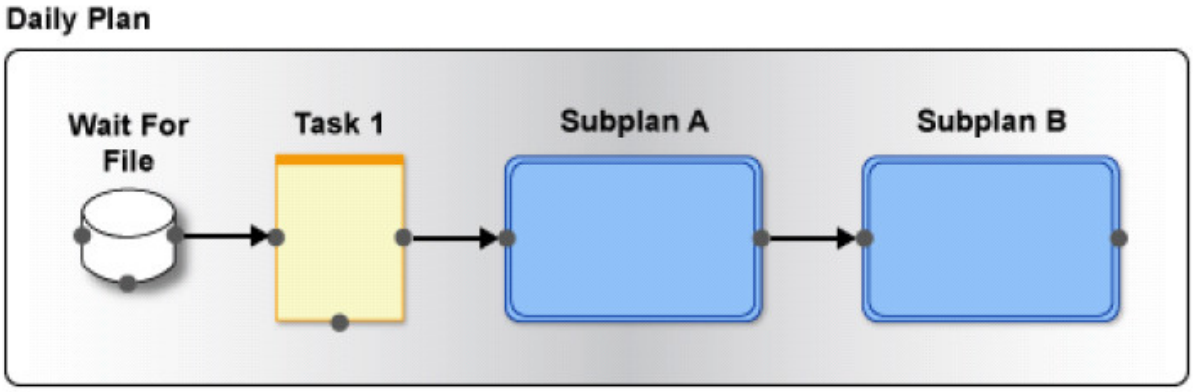
Build a plan that does the following:

- Checks whether the output file exists.
- If the file does not exist, runs the Simple Dependency plan from Exercise 1.
- If the file does exist, runs just the Move and Reformat graph.
- After all tasks have completed, runs a script in a separate task that prints the message "File has been moved successfully!"
- Specifies the paths to the input file and output file as plan input parameters.

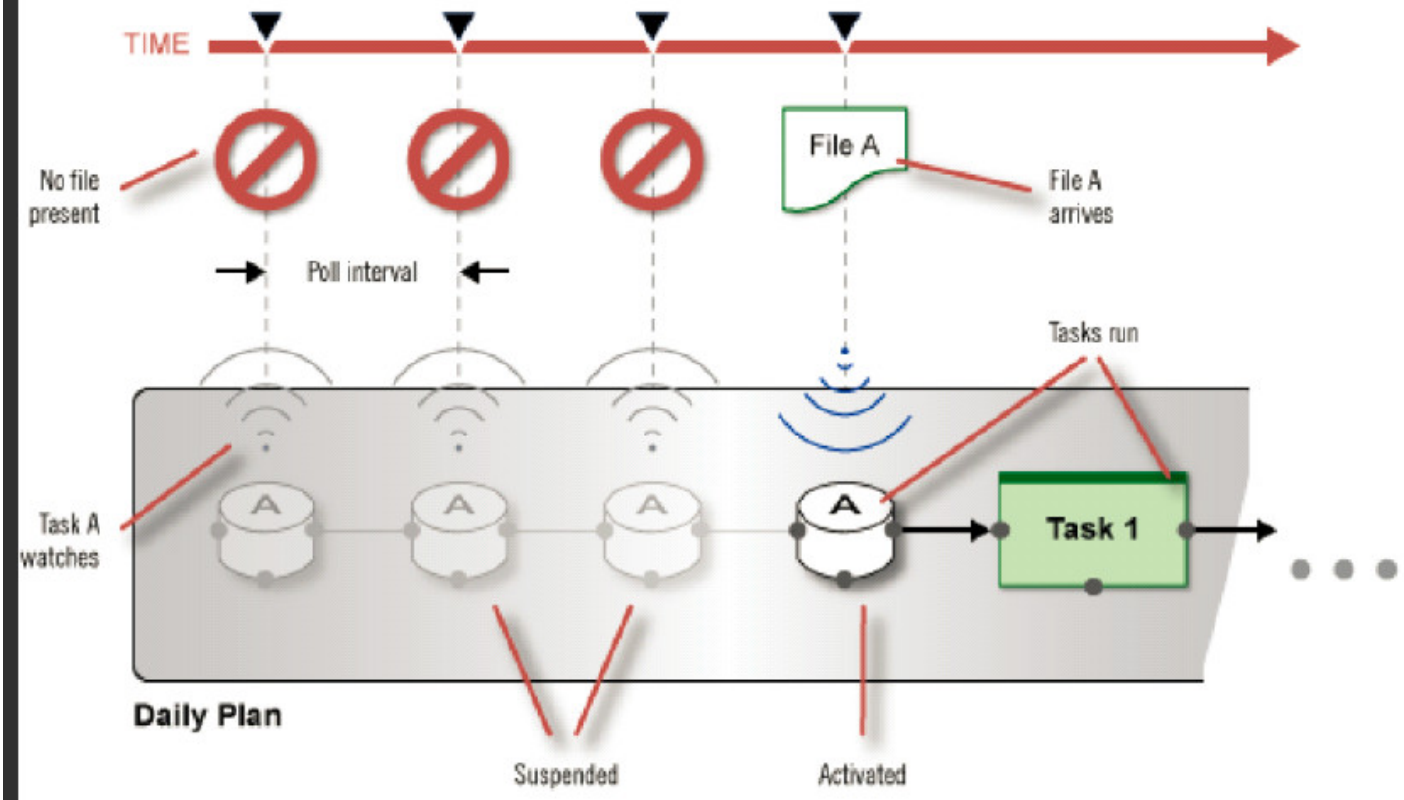
Save your solution as **ex3_conditional_dependency.plan**

Tasks: Wait For File

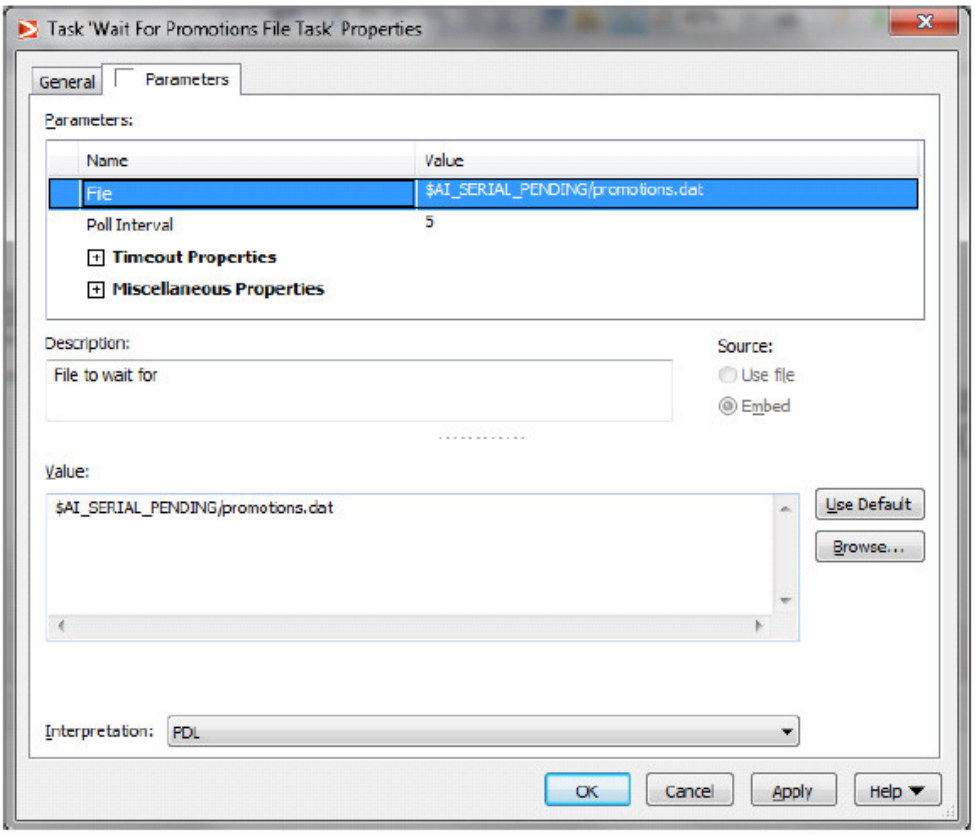
- Only supports local files
- Filename must be explicit (no wildcards)
- The Poll Interval parameter controls how often the task checks to see if the file has arrived.
- The Timeout properties allow you to limit how long the task waits.



Wait for File: Wait for File Task Operation



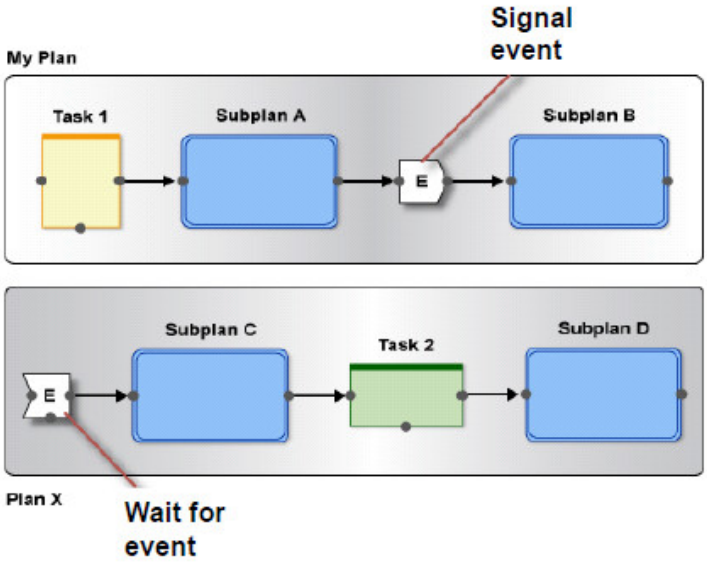
Resources: Wait for File Task Properties



Tasks: Signal Event & Wait For Event

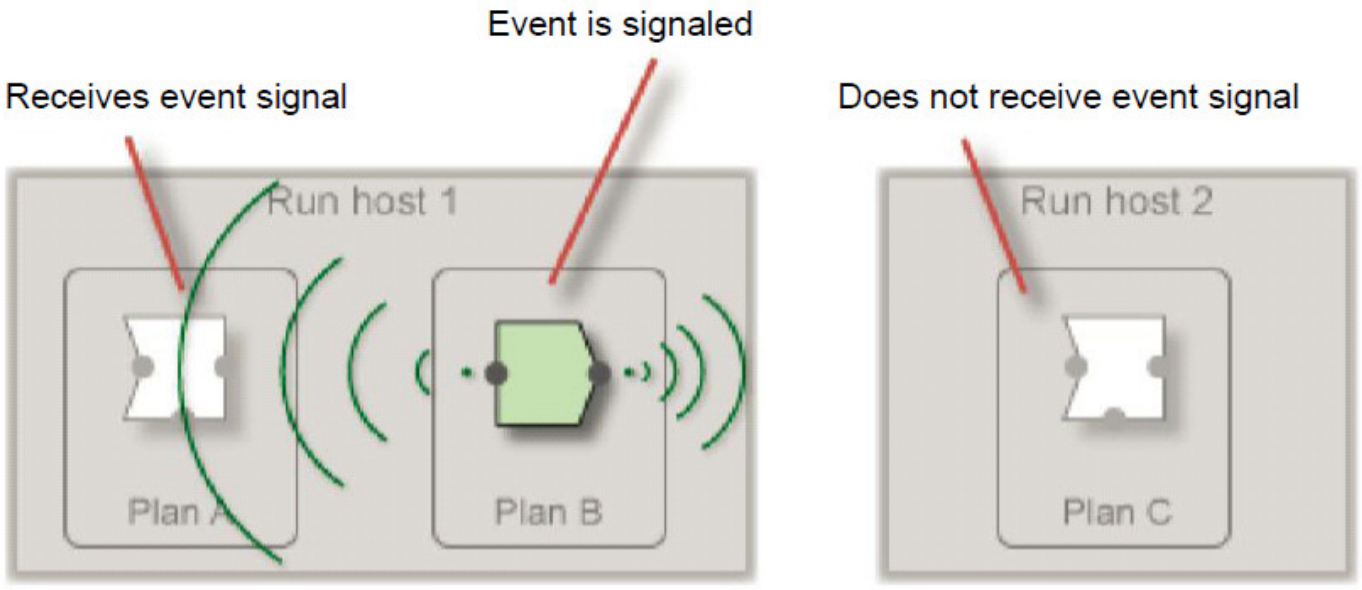
- Used for synchronization between two plans
- Once signaled, events remain in that state until you reset them
- To reset a signaled event, use:

  resource-admin unsignal event-name

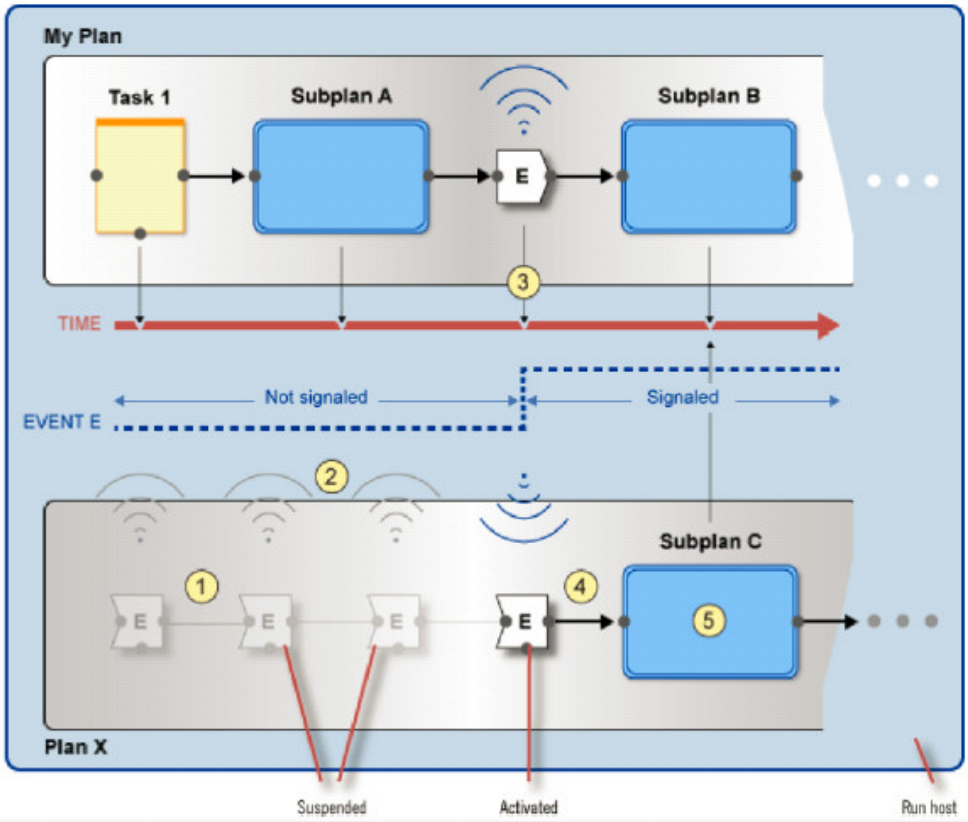


Event Tasks: Range of an Event Task Signal

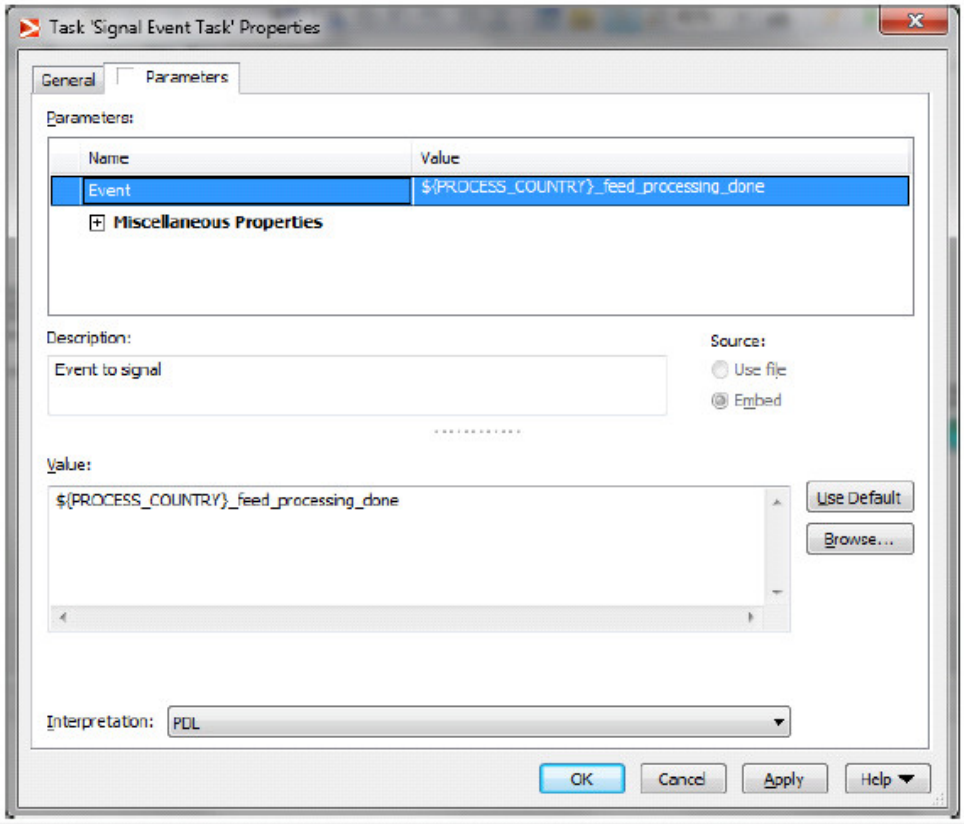
Events are received on the same Run host that issued the signals.



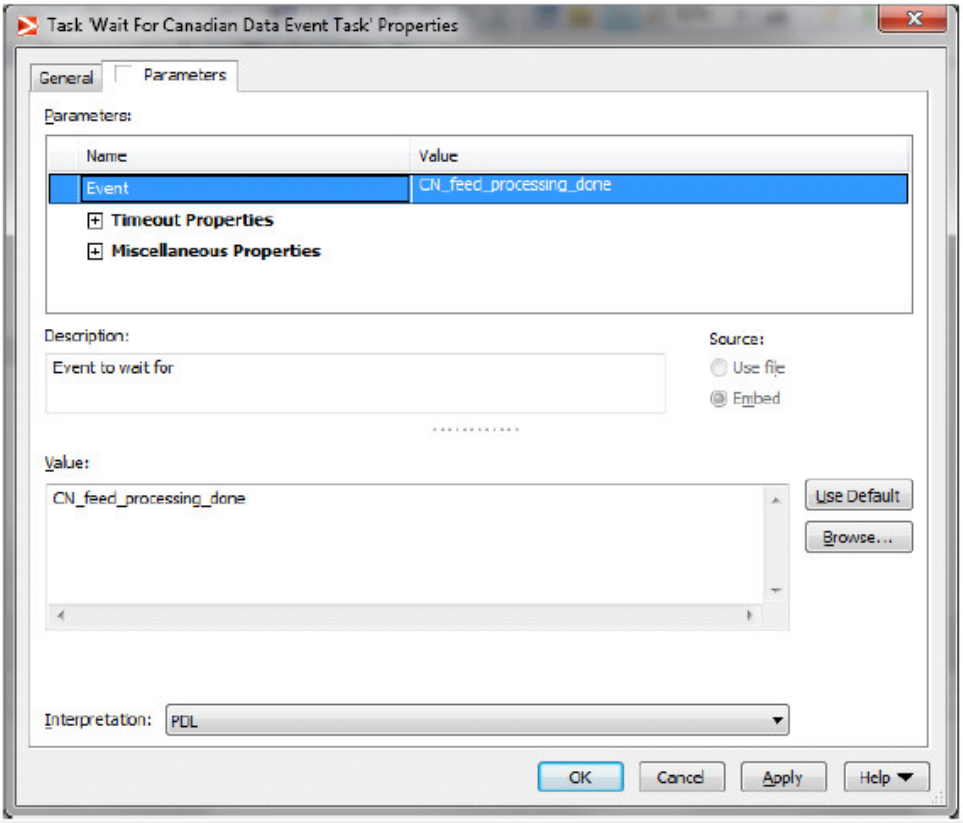
Event Tasks: Event Task Signal – Operation



Resources: Signal Event Task



Resources: Wait for Event Task



Tasks: Section Review

In this section, we discussed:

- Different types of tasks
  (Graph, Program, Conditional, Plan, Signal Event, Wait for Event, Wait for File)
- Input Parameters
- PDL

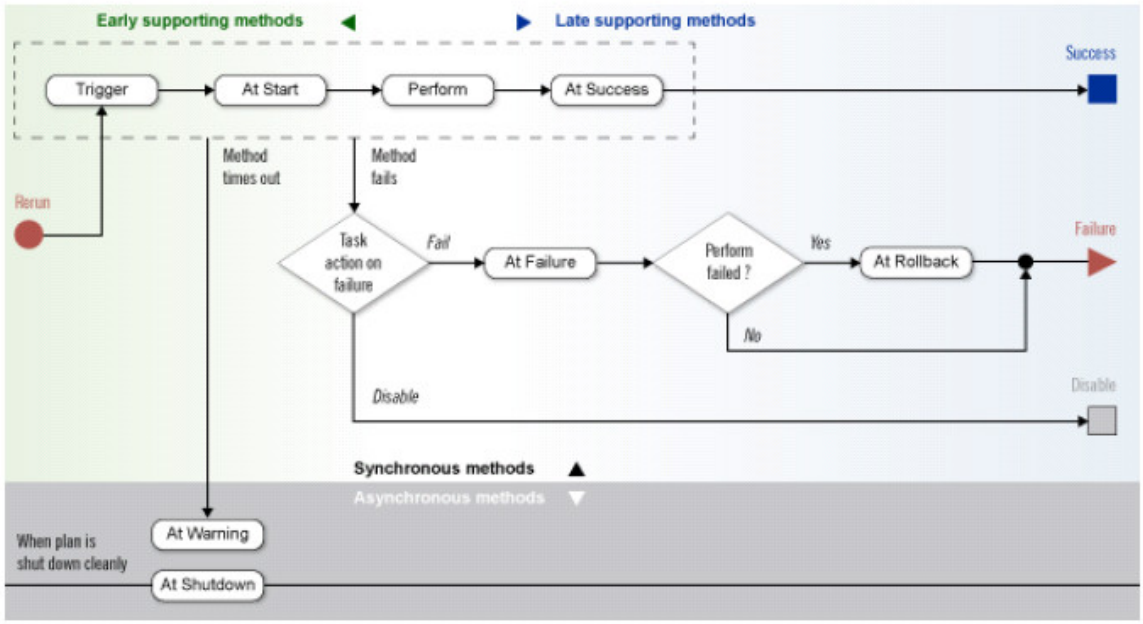
Methods

Methods: Section Topics

- Different methods associated with a task
- Inherited methods
- Conduct>It specific parameters

Methods: What are Methods?

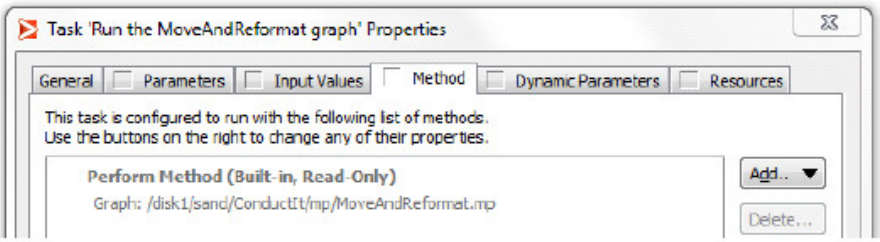
A task is actually a *collection of methods* that execute in a prescribed way when the task runs.



Methods: The Perform Method

Tasks have a built-in read-only method called *perform*.

When you configure a task, you are really configuring its perform method.



Methods: Trigger and At Start Methods

These methods run *before* the Perform method:

Trigger Method:

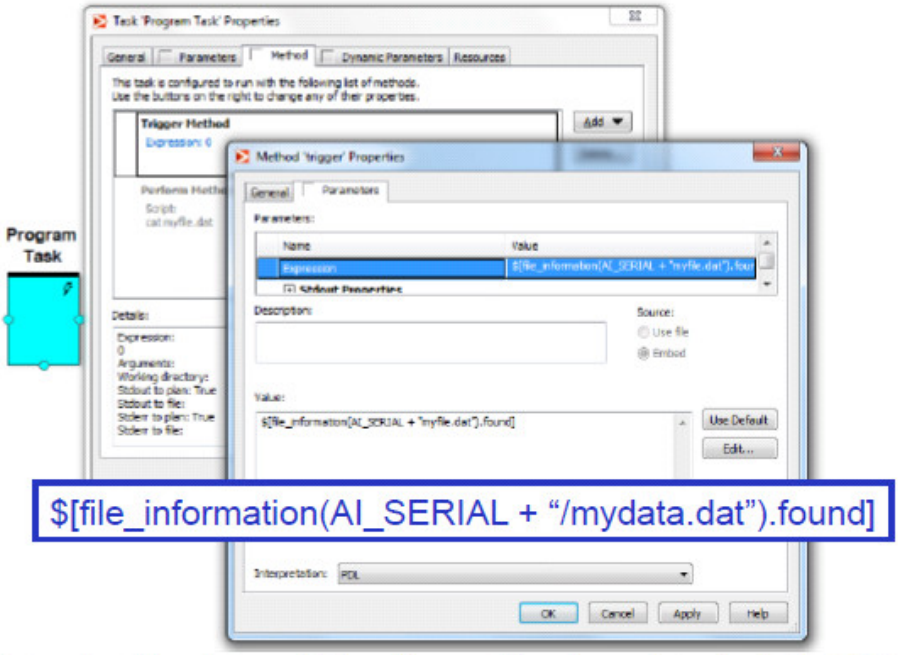
Specifies a condition that must be met before the task executes

At Start Method:

Runs after a task is triggered but before the perform method.

Methods: Trigger Methods

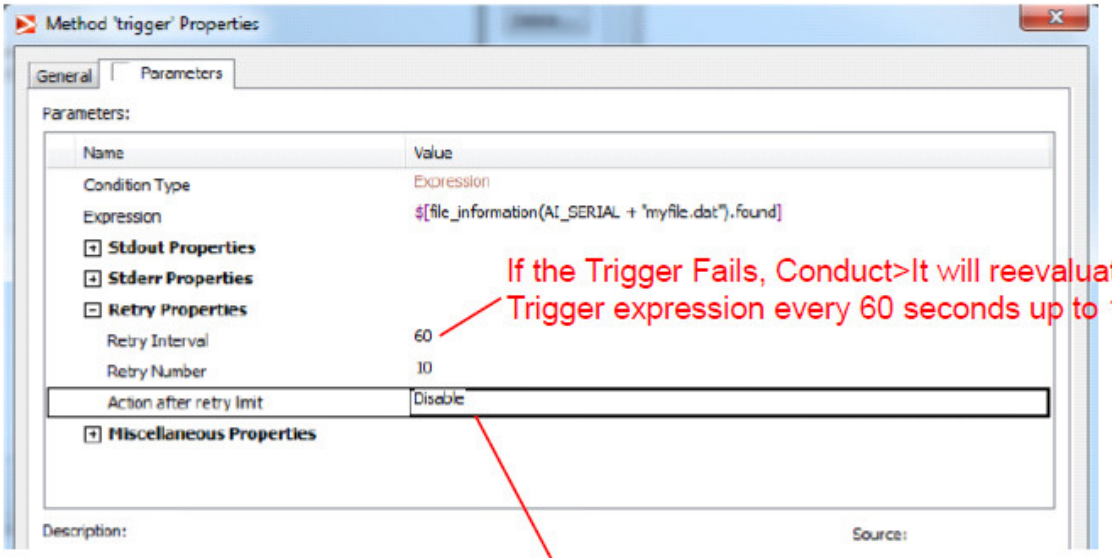
By setting a Trigger method, you can require a condition to be satisfied before Conduct>It begins executing a task.



This task will only run if the file **myfile.dat** exists in **$AI_SERIAL**.

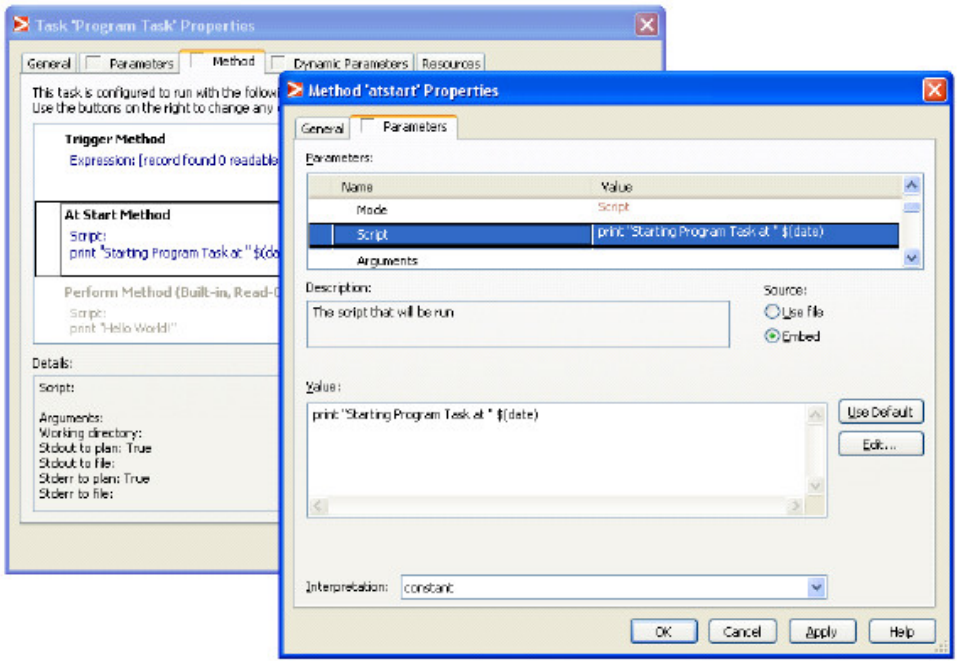
Methods: Trigger Methods (cont'd)

A Trigger doesn't have to be a one-shot test — either pass or fail. You can *retry* a Trigger.



Methods: At Start Methods

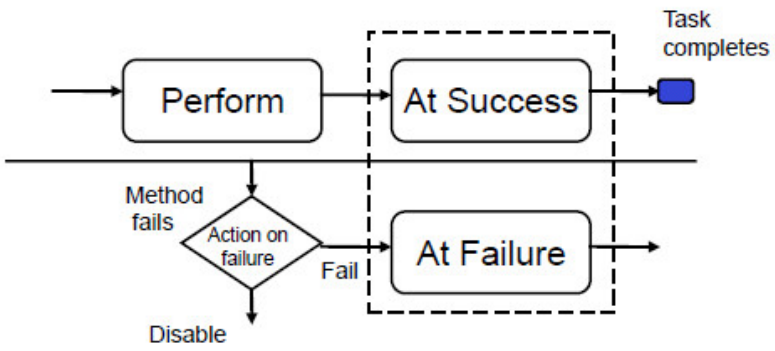
You can use an At Start method to run an arbitrary script or program after the Trigger, but before the Perform method.



At Start methods are often used to log start-up information.

Methods: At Success and At Failure Methods

These methods depend upon the success of the task:



At Success Method

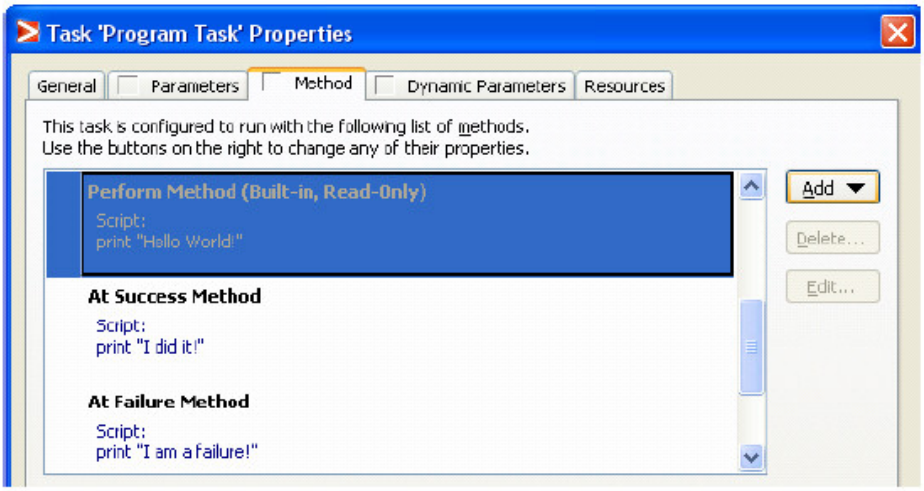
Runs when the Perform method finishes successfully.

At Failure Method

Runs when any previous method in the task fails.

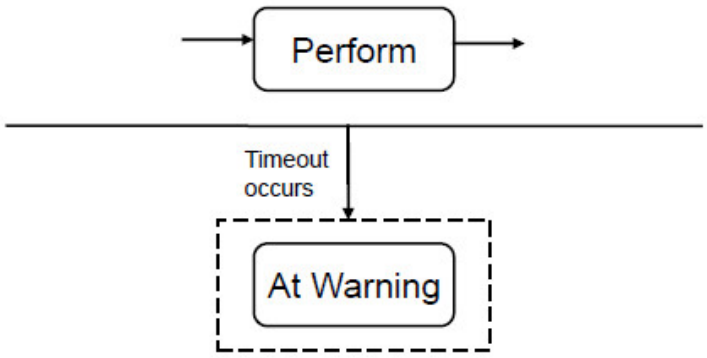
Methods: At Success and At Failure Methods (cont'd)

As with the At Start method, the At Success and At Failure methods can run arbitrary scripts or programs.



Conduct>It will run only At Success OR At Failure, depending on the return status of the Perform Method.

Methods: At Warning Method

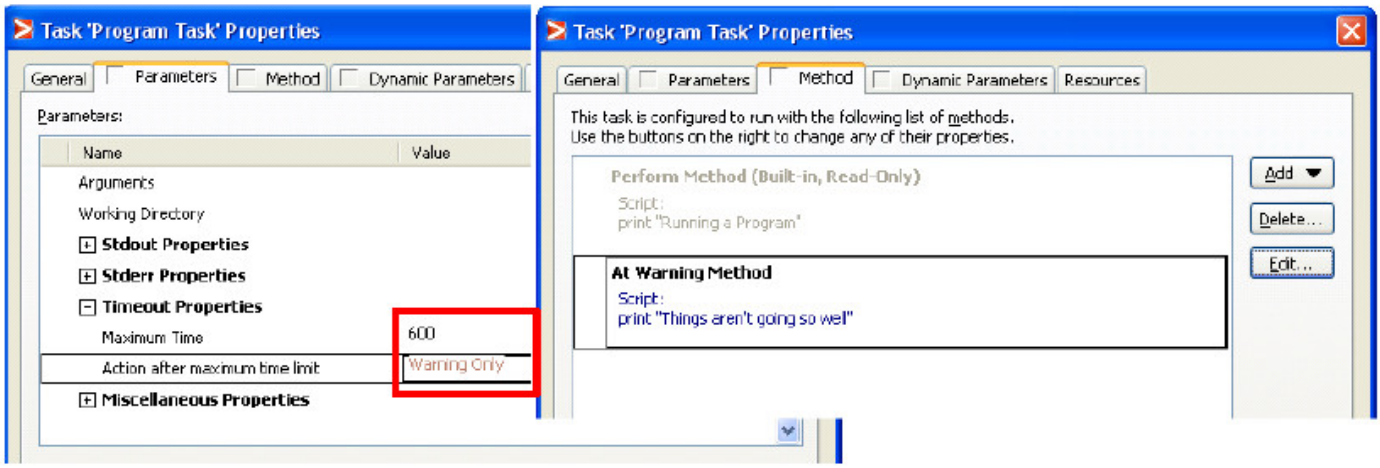


At Warning Method

Runs when a task's Perform, At Start, or At Success method (or a Plan/SubPlan's At Start or At Success method) exceeds a specified execution-time threshold.

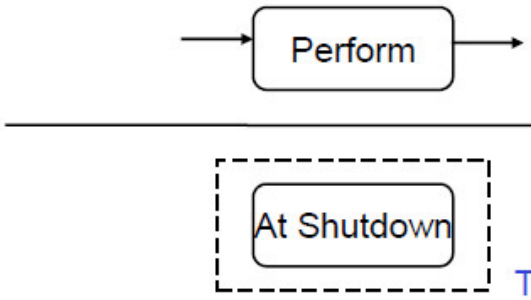
Methods: At Warning Method (cont'd)

You can set the Perform, At Start, and At Success methods' Timeout properties so that, if a time limit is exceeded, the *At Warning* method will be run.



After 10 minutes, Conduct>It will run this task's At Warning method to print a message to stdout.

Methods: At Shutdown Method



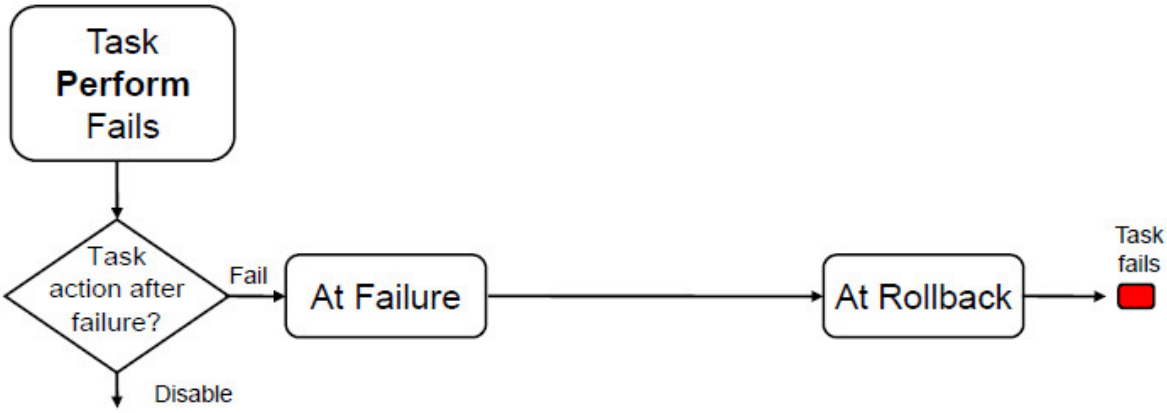
To invoke At Shutdown run:
plan-admin *recfile* shutdown

At Shutdown Method

Contains an executable required to shut down the Perform method when this becomes necessary.
You run this method manually.

This method is usually used to shutdown continuous graphs.

Methods: At Rollback Method



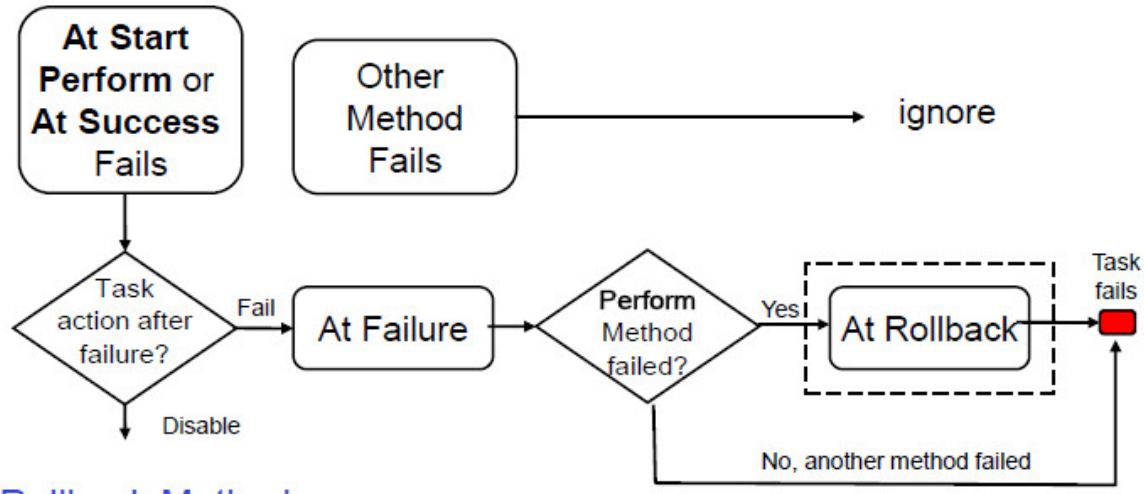
At Rollback Method

Runs only when the ***Perform*** method fails and causes the task to fail.

Restores the state of the environment so that the task can be recovered

But *what if some other method fails??*

Methods: At Rollback Method



At Rollback Method

Runs only when the ***Perform*** method fails and causes the task to fail.

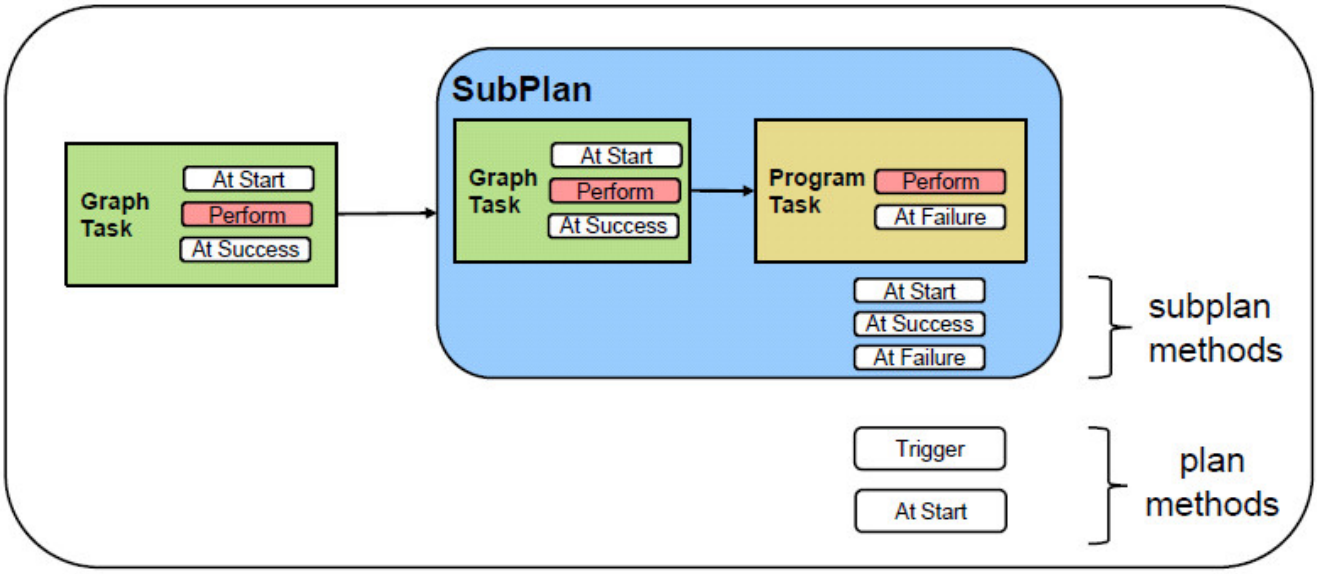
Restores the state of the environment so that the task can be recovered

Any other failing method will not trigger the At Rollback Method

Methods: Attaching Methods to Plans

Most methods you implement for tasks can also be implemented for plans and subplans.

The exception is the Perform Method which is only valid for plan *tasks*, graph tasks, and program tasks.



Exercise: Exercise 4 – Methods

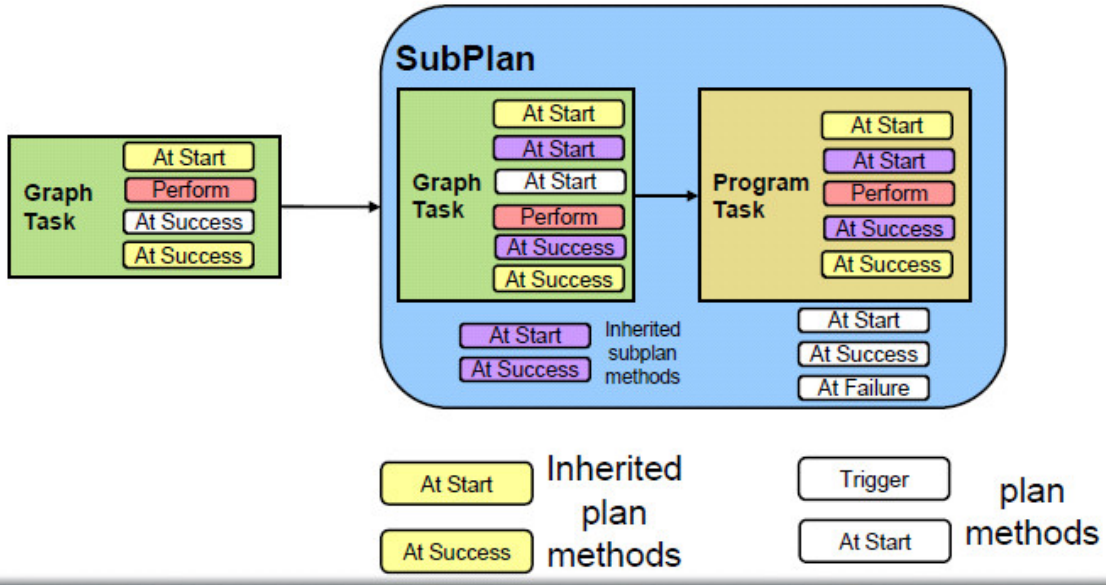
Modify the plan from Exercise 3 (Conditional Dependency) such that:

- It has a trigger that ensures tasks will only run after 12PM
- It uses methods to print a message that indicates whether the conditional task evaluated to true or false.
- Save your solution as ex4_methods.plan
- Change the trigger condition to see how it affects the plan.

Methods: Inherited Methods

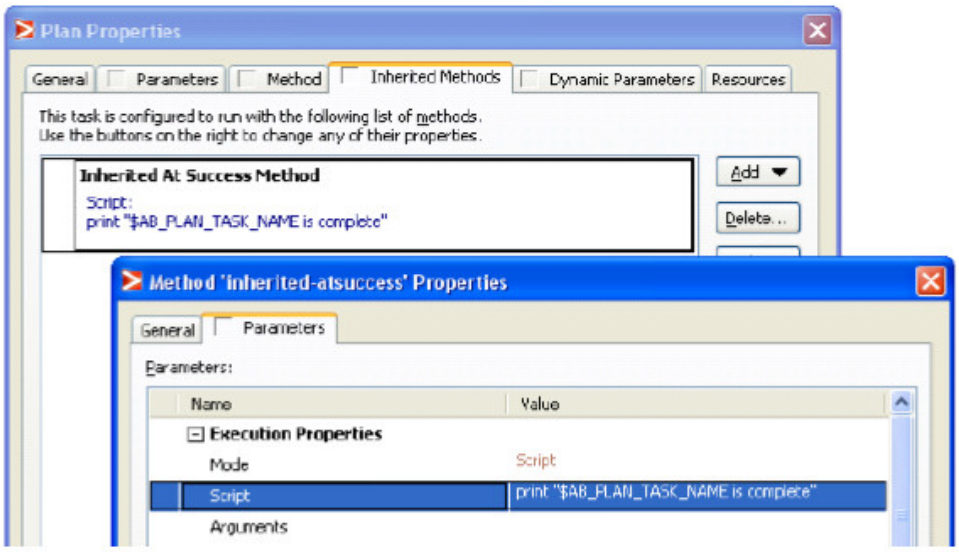
You can use inherited methods to quickly apply the same method to *all* the tasks in a plan or subplan.

Note: Suplans don't inherit methods, but their containing tasks do.



Methods: Inherited Methods (cont'd)

To add an inherited method, go to the Inherited Methods tab of the Plan Properties window.



Methods: Conduct > It Parameters

Some parameters (variables) are set in the environment of a plan and task, and can be used in method messages.|

AB_PLAN_FILE_PLAN_NAME
Returns the full path of the plan (.plan) or input values set (.pset) file that is being run.

AB_PLAN_TASK_NAME
Returns the name of the task the method is attached to.

AB_PLAN_METHOD_NAME
Returns the name of the referring method, e.g.
/My*Plan/SubPlan/Task-A/AtStart*

Exercise: Exercise 5 – Inherited Methods

Modify the plan from Exercise 4 (Methods) such that:

- It has an inherited method that prints a message indicating each task has succeeded.
- Includes the task name in the message.

Save your solution as ex5_inherited_method.plan

Make sure that trigger method is set so that the tasks will run.

Methods: Section Review

In this section, we discussed:

- Different types of methods
- Inherited methods
- Conduct>It specific parameters

Failure and Recovery

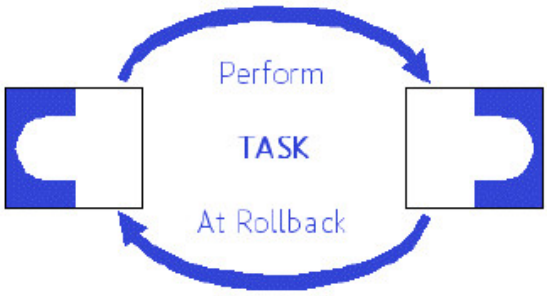
Recovery: Section Topics

- Rollback methods
- Recovery options
- Connection points

Recovery: At Rollback Method

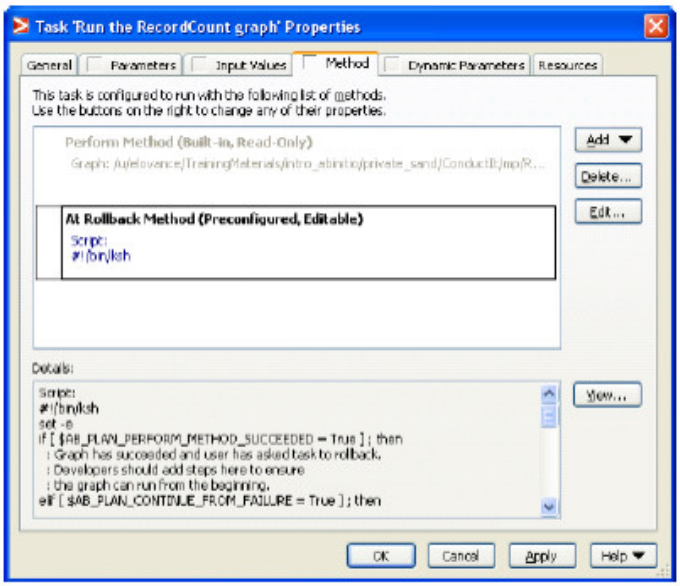
One of a task's supporting actions restores the system to a consistent state after failure so that it can rerun.

This is called the *At Rollback method*.



Recovery: At Rollback Method (cont'd)

Ab Initio graphs and plans have a built-in recovery mechanism. Conduct>It takes advantage of this by building in *default At Rollback* methods for Graph and Plan tasks.




```
if [ $AB_PLAN_PERFORM_METHOD_SUCCEEDED = True ] ; then
  : Graph has succeeded and user has asked task to rollback.
  : Developers should add steps here to ensure
  : the graph can run from the beginning.
elif [ $AB_PLAN_CONTINUE_FROM_FAILURE = True ] ; then
  : Graph has failed and user has asked task to continue.
  if [ -a "$AB_GRAPH_RECOVERY_FILE" ] ; then
    m_rollback "$AB_GRAPH_RECOVERY_FILE"
  fi
elif [ $AB_PLAN_RUNNING_AUTOROLLBACK = True -a
$AB_PLAN_TASK_HAS_FAILURE_CONNECTION != True ]; then
  : This task has nothing connected for failure, so leave .rec file.
  if [ -a "$AB_GRAPH_RECOVERY_FILE" ] ; then
    m_rollback "$AB_GRAPH_RECOVERY_FILE"
  fi
else
  : Graph has failed and user has asked task to rollback,
  : or task is running autorollback and user has something connected for failure.
  if [ -a "$AB_GRAPH_RECOVERY_FILE" ] ; then
    m_rollback -d "$AB_GRAPH_RECOVERY_FILE"
  fi
fi
exit 0
```

Recovery: At Rollback Method (cont'd)

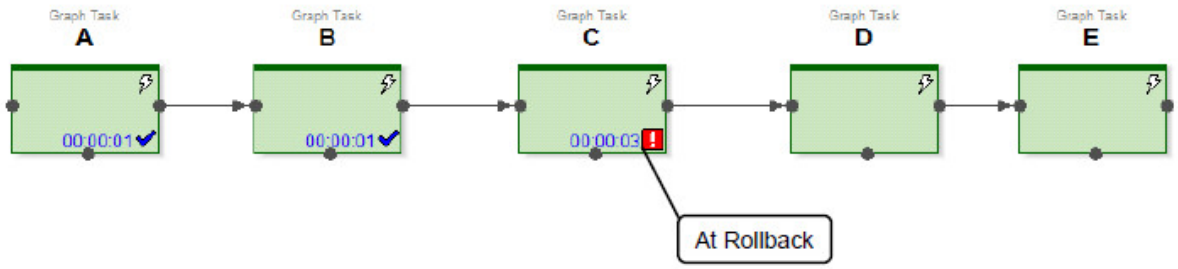
For program tasks, you may need to write the appropriate At Rollback method.

In an At Rollback method, you may do such things as:

- Clean up temporary files
- Drop tables
- Undo changes

Note that you can modify the default At Rollback method for Graph Tasks and Plan Tasks to perform additional actions.

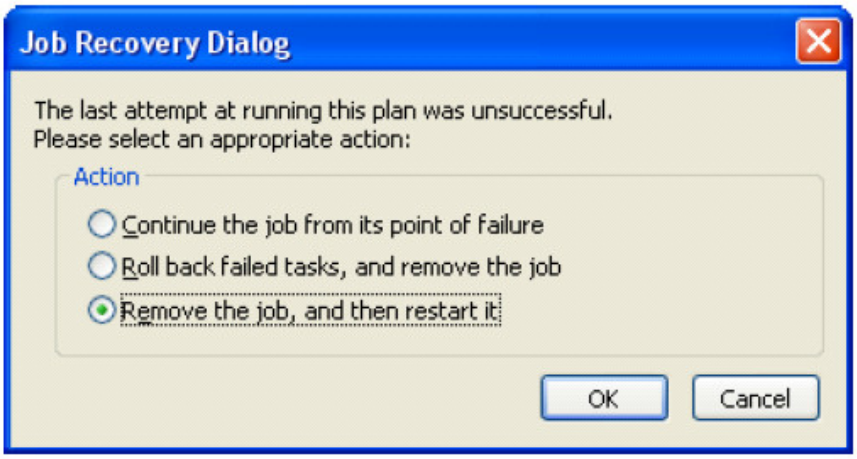
Recovery: Automatic Roll Back



By default, Conduct>It runs a failed task's At Rollback method automatically when possible, immediately after failure.

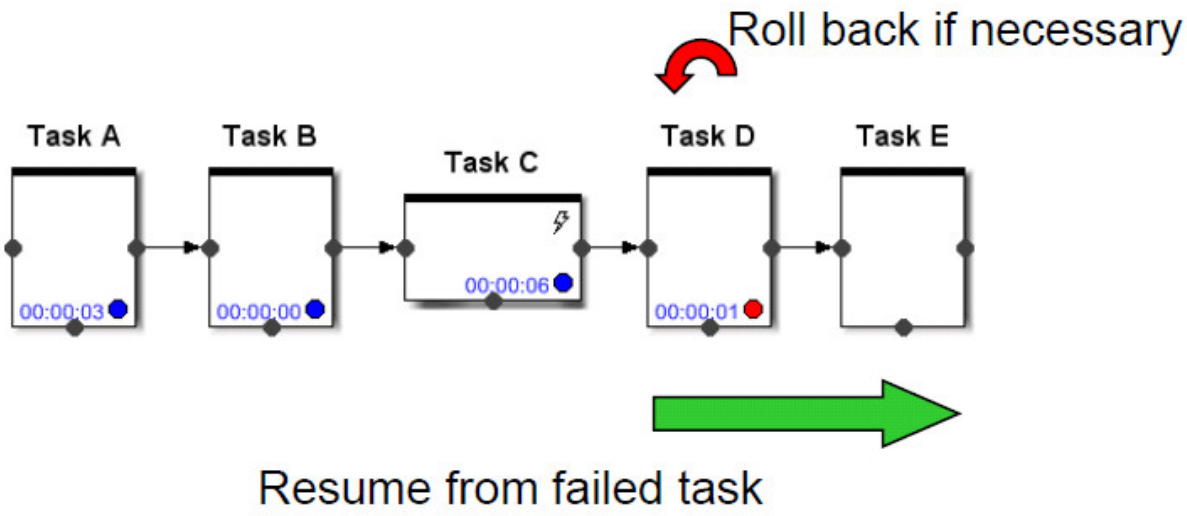
Recovery: Recovery Options

When you rerun a failed plan within the GDE, you are presented with the following recovery options:



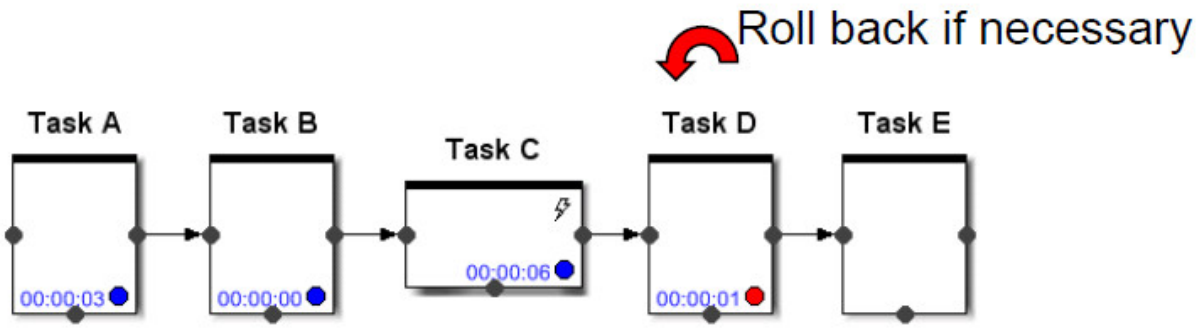
Recovery: Recovery Options (cont'd)

During a "continue the job from its point of failure" recovery, Conduct>It runs any At Rollback methods for failed tasks that did not automatically roll back and resumes from the point of failure.



Recovery: Recovery Options (cont'd)

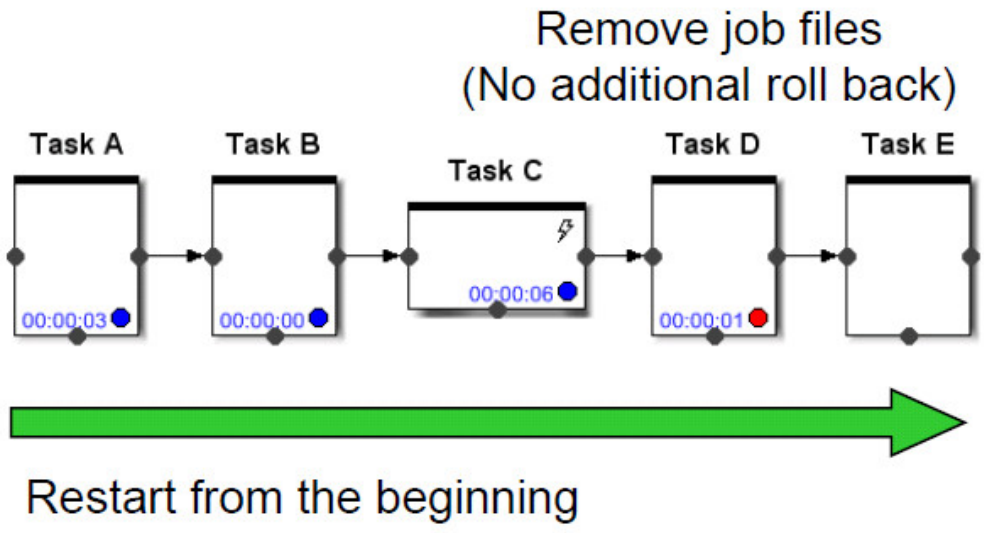
For "roll back failed tasks, and remove the job", Conduct>It runs the At Rollback methods for failed tasks that did not automatically roll back and removes the recovery file. *Plan itself is not run.*



Note: When the plan is rerun, it will start from the beginning.

Recovery: Recovery Options (cont'd)

To "remove the job, and then restart it", Conduct>It removes all job files, including the recovery file, and restarts the plan from the beginning.



Recovery: Recovery Options (cont'd)

To perform the same actions from the command line:

- Continue a plan from its point of failure

  air sandbox run *planfile*

- Roll back failed tasks, and remove the job

  air sandbox run *planfile* –rollback-failures-and-remove

- Remove the job, and then restart it

  air sandbox run *planfile* –remove-all

  air sandbox run *planfile*

The command-line offers more fine-grained options for rollback

Exercise: Exercise 6 – Recovery

Write a plan that does the following:

- Runs a script that creates the directory $AI_SERIAL/rec **mkdir –p …**
- Once the script completes, runs a script that touches the file $AI_SERIAL/rec/rec.dat and then exits with status 1 (failure). **exit 1**
- After both scripts complete, runs the Move and Reformat graph.
- Has an appropriate At Rollback method for both scripts.
- Has an inherited At Failure method that prints a message indicating which task failed.

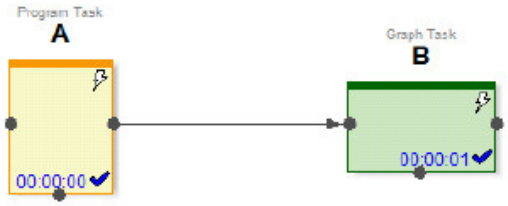
Save your solution as ex6_recovery.plan

Note that the plan will initially fail.

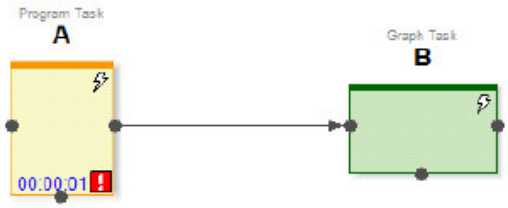
- After it fails, verify the relevant task rolled back as expected.
- Modify the touch script so that it exits with a status of 0 (success), and then rerun the plan from the point of failure.

Recovery: Unhandled failures cause the plan to fail

A succeeds
Plan succeeds



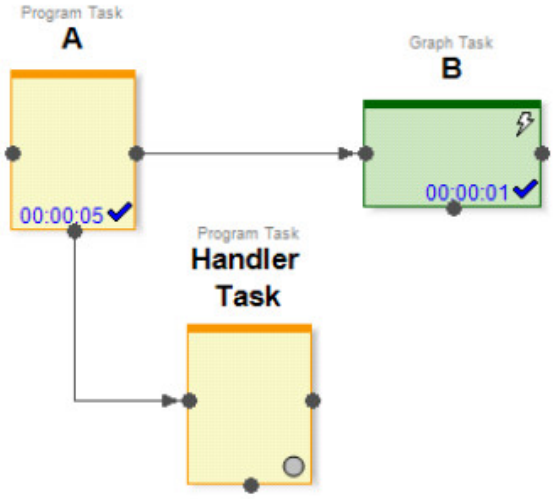
A fails
Plan fails



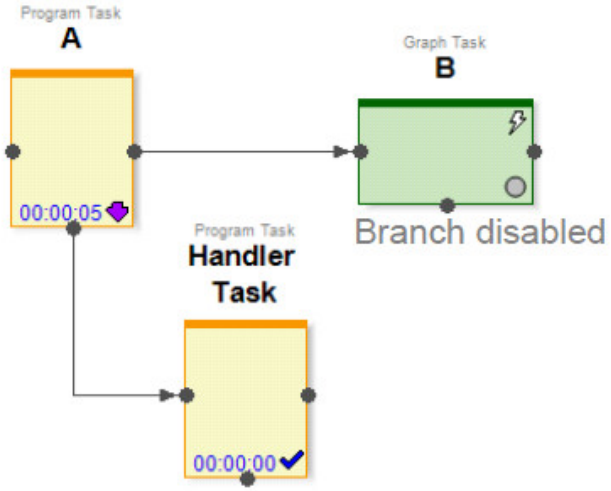
Recovery: Failure Connection Point

The failure connection point allows you to run error-handling tasks on failure.

A succeeds
Plan succeeds



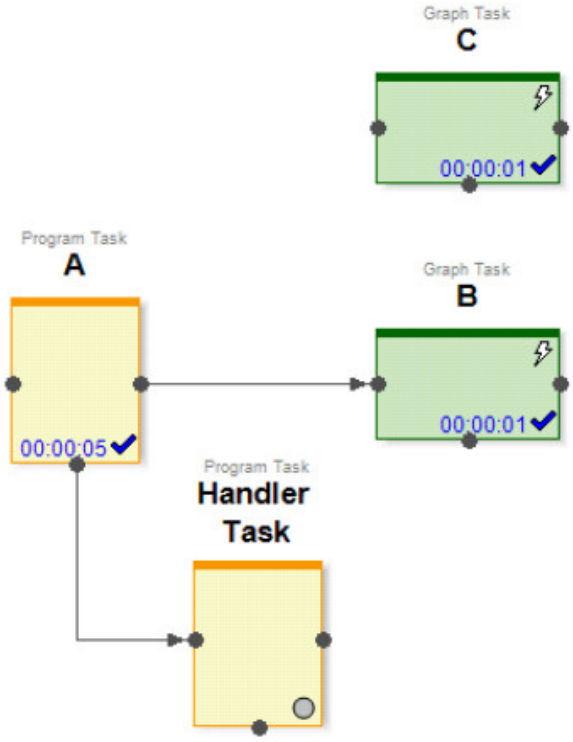
A is "handled"
Plan succeeds!



Recovery: Failure Connection Point (cont'd)

When a successor task is connected to a task's failure connection point:

- The plan will continue even if the task fails.
- On failure, the At Failure and At Rollback methods will run before control is passed to the successor task.



Exercise: # Exercise 7 – Connection Point

Modify the plan from Exercise 6 (Recovery) such that:

- If the touch script fails, another script runs before the graph task that touches the file $AI_SERIAL/rec/failure.dat
- Save your solution as ex7_connection_point.plan

Change the exit status of the touch script to see the different tasks run.

Recovery: Section Review

In this section, we discussed:

- Rollback methods
- Recovery options
- Connection points

Parameters

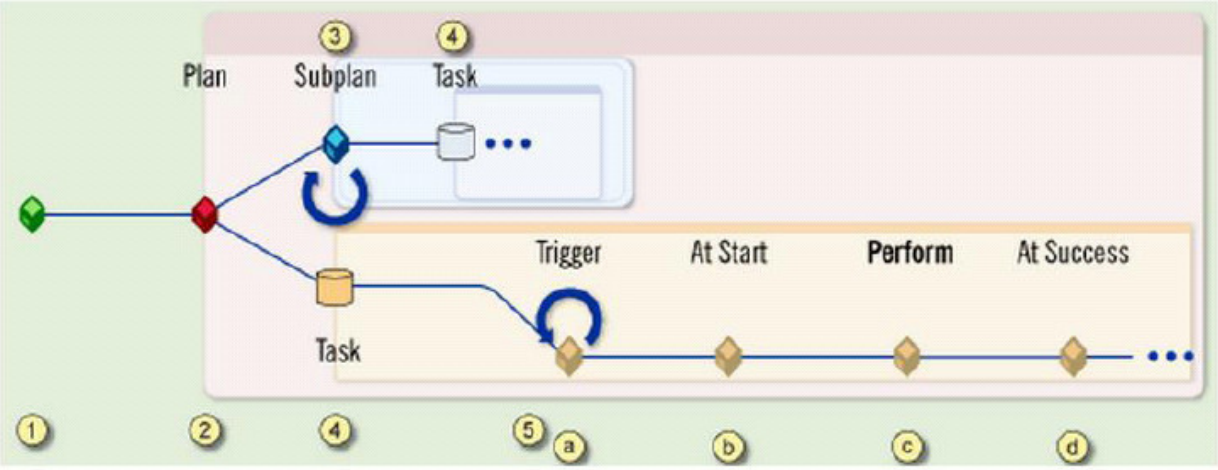
Parameters : Section Topics

- Order of Evaluation
- Dynamic Parameters
- Parameter Scope

Parameters: Parameter Order of Evaluation

Conduct>It evaluates parameters at different times:

- Plan/Sandbox parameters when the top-level plan starts
- Task/Subplan parameters when a task or subplan starts
- Looping Subplan parameters at each iteration

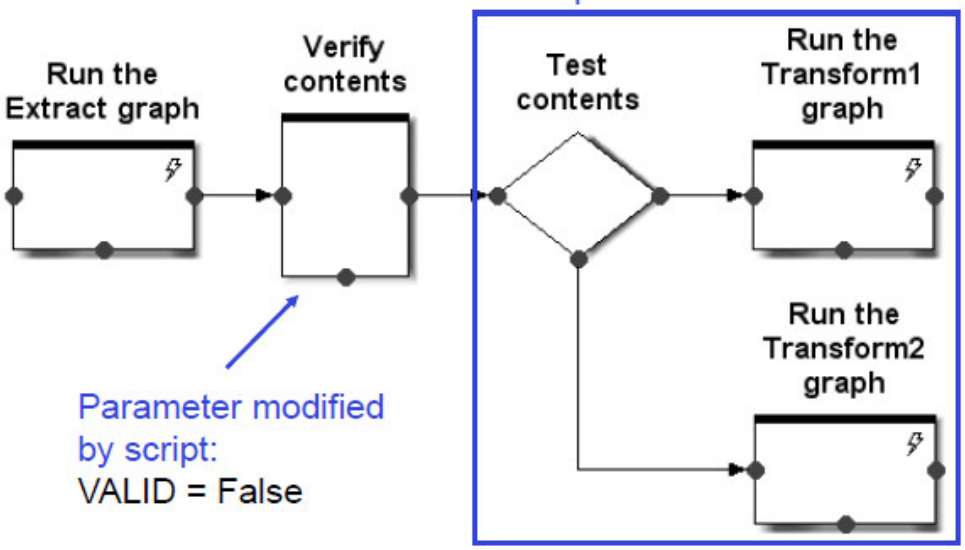


Parameters: Dynamic Parameters

Multiple plans, graphs, and tasks can read and modify the same parameters while a plan runs.

New value used for all dependent methods and tasks within the scope of the modified parameter.

Initial value of Parameter:
VALID = True



Parameter modified by script:
VALID = False

Parameters: Using Dynamic Parameters

Changing parameters dynamically requires 3 steps:

1. At the **plan level**, declare an input parameter.

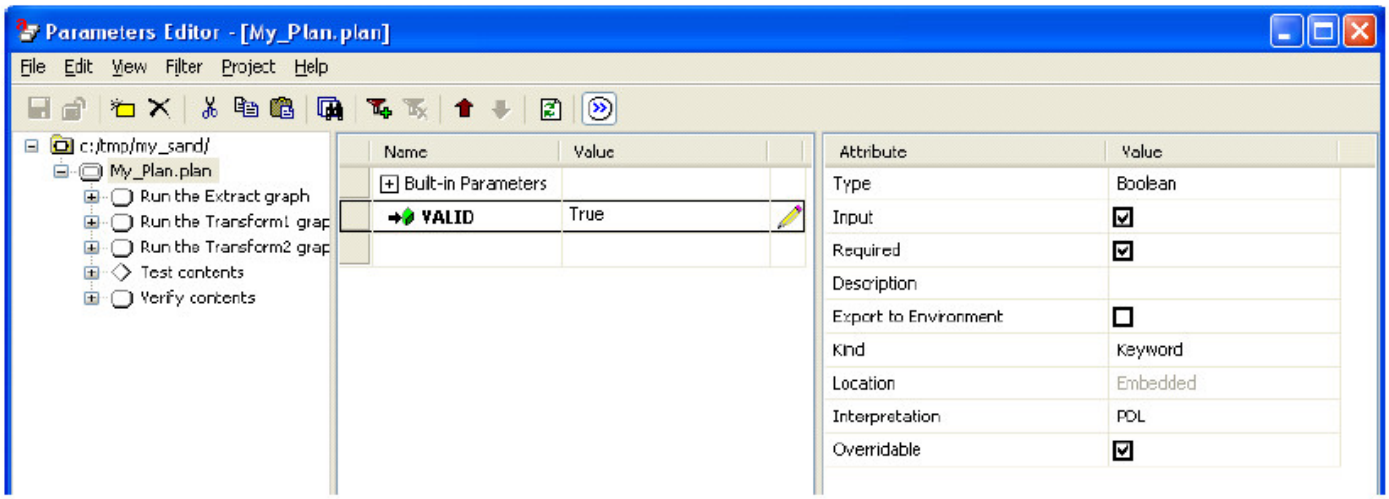
2. At the **task level**, go to the Dynamic Parameters tab and add the newly declared parameter by selecting it from the pulldown list.

3. At the **method level**, modify the parameter value by calling the command:

   plan-admin set <parameter-name> <value>

Parameters: Using Dynamic Parameters – Step 1

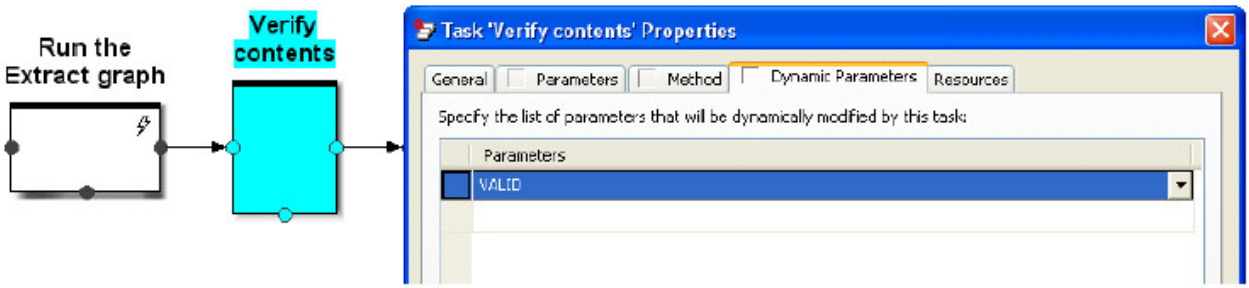
Plan level: Define the input parameter.



Parameters: Using Dynamic Parameters – Step 2

Task level: Declare the parameter *dynamic* by including it on the Dynamic parameters tab.

Listing VALID here indicates your intent to have this task modify the parameter's value.

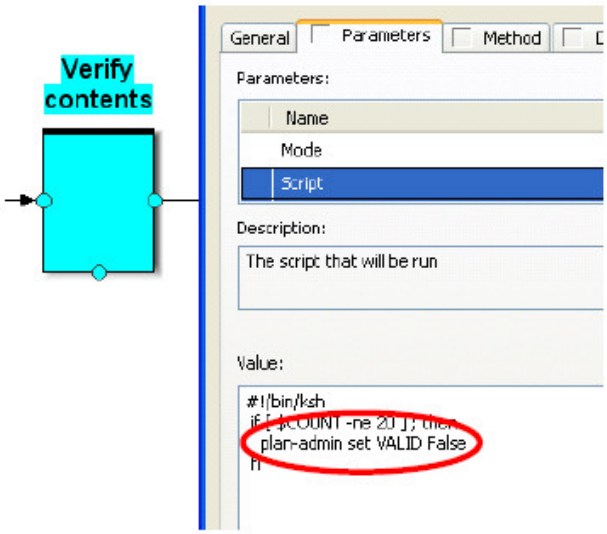


NOTE: The task *fails* if it attempts to modify a parameter that has not been declared dynamic.

Parameters: Using Dynamic Parameters – Step 3

Method level: Change the parameter's value with the command:

plan-admin set <parameter-name> <value>



You can use "plan-admin set" in any method to change the value of a dynamic parameter.

Exercise: Exercise 8 – Dynamic Parameters

Write a plan that does the following:

Has an input parameter the initial value “Yes”.

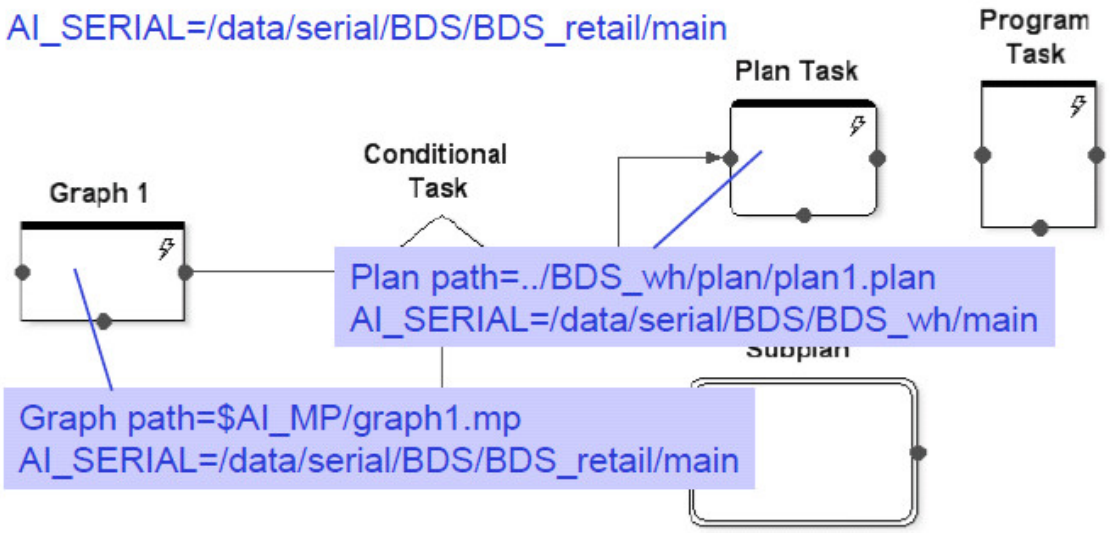
- Runs an initial script that prints the value of the parameter.
- Following the completion of that script, has another script that changes the value of the input parameter to “No”.
- Following the completion of the previous two scripts, runs a script in a separate task that prints the new value.
- Include a fourth task that is independent of all of the other tasks and that echo the value.
  (What will be the value of the value?)

Save your solution as ex8_dynamic_parameters.plan

Parameters:

What About the Graph itself?

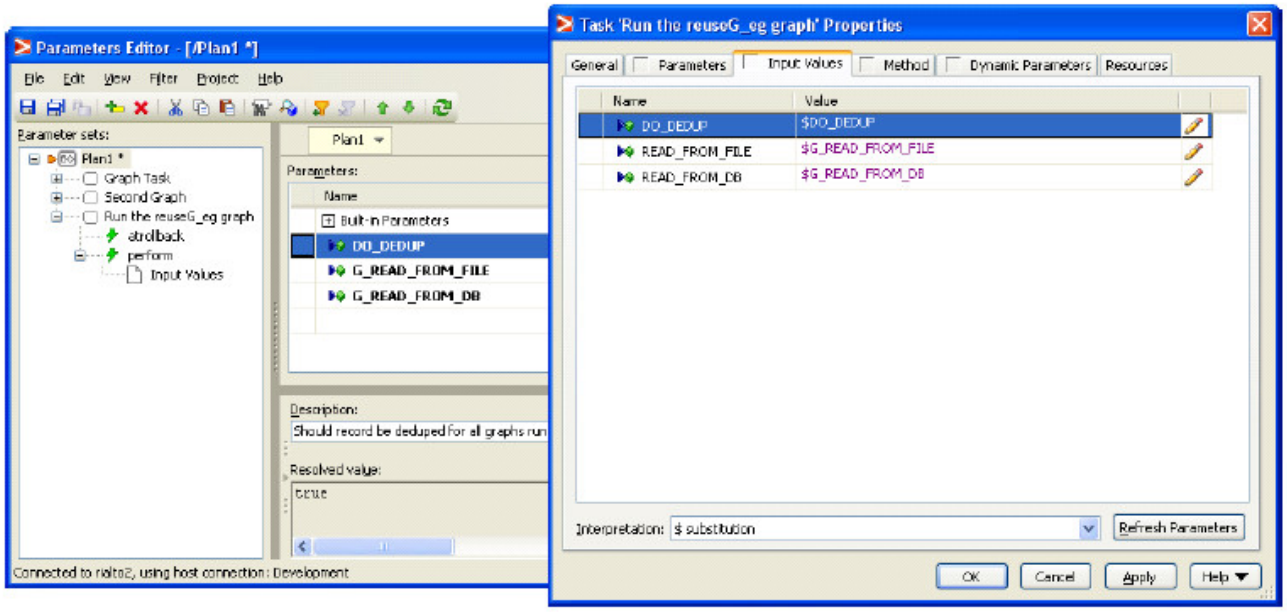
By default, an application run from a Plan or Graph Task will start in a "clean" shell and reevaluate the entire environment, as if it were run from the command-line.



Parameters:

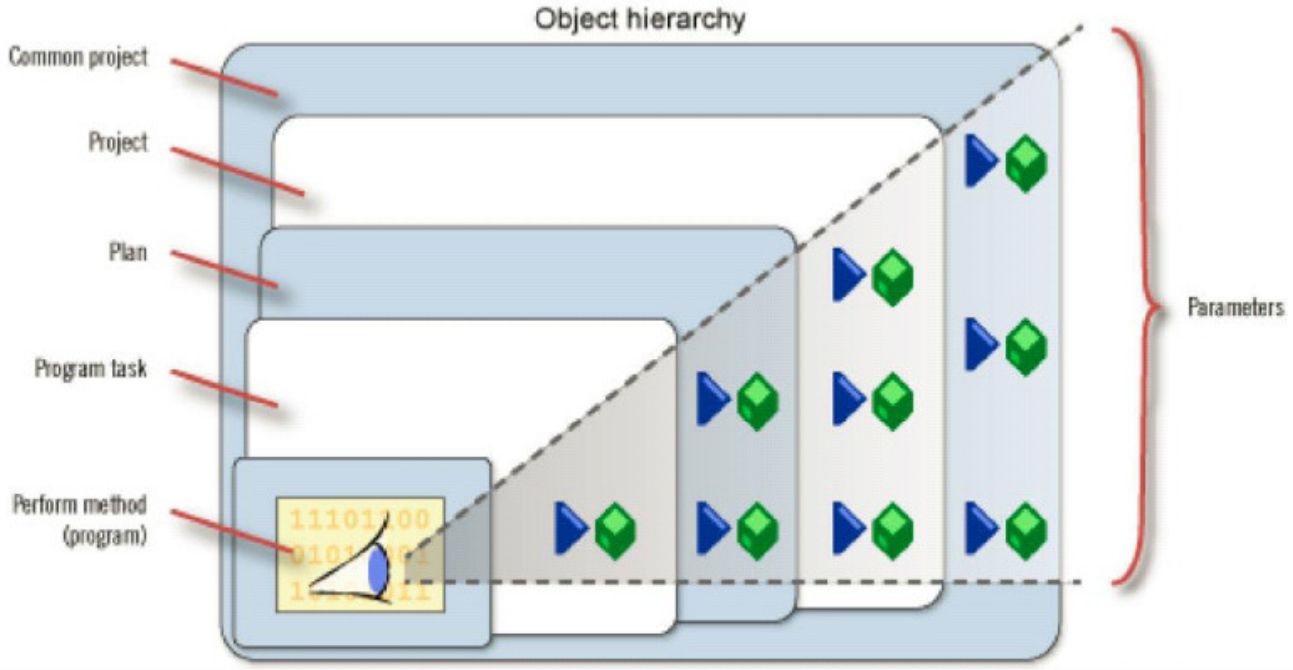
Using Plan Parameter Values in Graph/Plans

Plan parameter values can be provided to graphs and plans using the **Input Values** tab on the task.



Tasks: Program Task Parameter Scope

You can attach parameters to any plan, graph, project, or common project. Parameters can be shared, or "seen," throughout an object hierarchy.



Parameters: Section Review

In this section, we discussed:

- Order of Evaluation
- Dynamic Parameters
- Parameter Environment

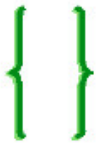
Subplans and Loops

Subplans: Section Topics

- Subplans
- Recovery Groups
- For Loops
- While Loops
- For Each Loops
- Subscribe Loops

Subplans: Subplans

Subplans are containers for:

- **Grouping** 

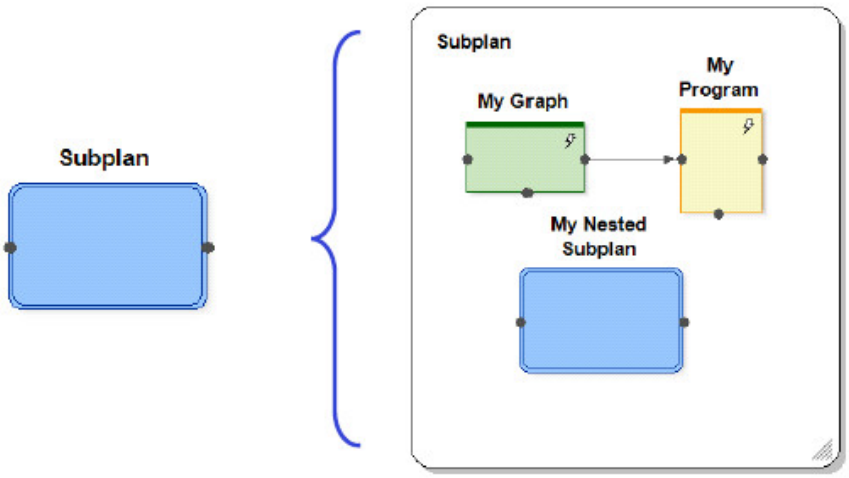
and

- **Looping** 

collections of tasks.

Subplans: Subplans (cont'd)

Grouping: A plan can contain any number of subplans, and subplans can be nested as needed.

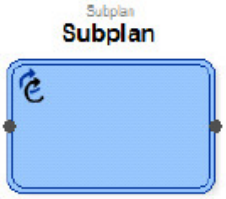


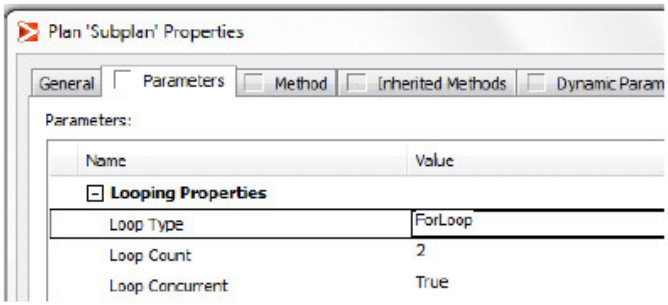
Subplans: Looping

Plans and their subplans can loop, where the constituent tasks and processing logic execute repeatedly.

Loop types:

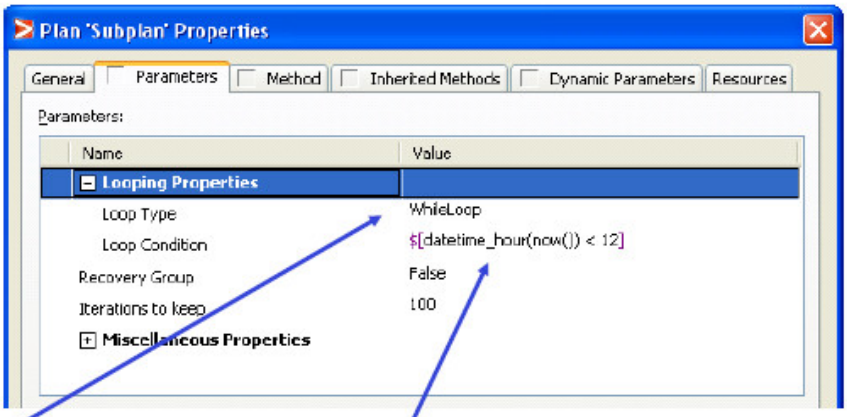
- While Loop
- For Loop
- For Each Loop
- Subscribe Loop





Subplans: While Loop

While loops run repeatedly while a specific set of conditions exist.



Select *WhileLoop* loop type.

Specify condition.

The loop will continue to execute until the condition is false.

Exercise: Exercise 9 – While Loop

Modify the plan from Exercise 5 (Inherited Methods) such that:

- The plan's trigger method is removed.
- The plan components are placed in a looping subplan configured with a while loop that runs as long as the file $AI_SERIAL/stop.control does not exist.
- The plan's input parameters are moved to the looping subplan

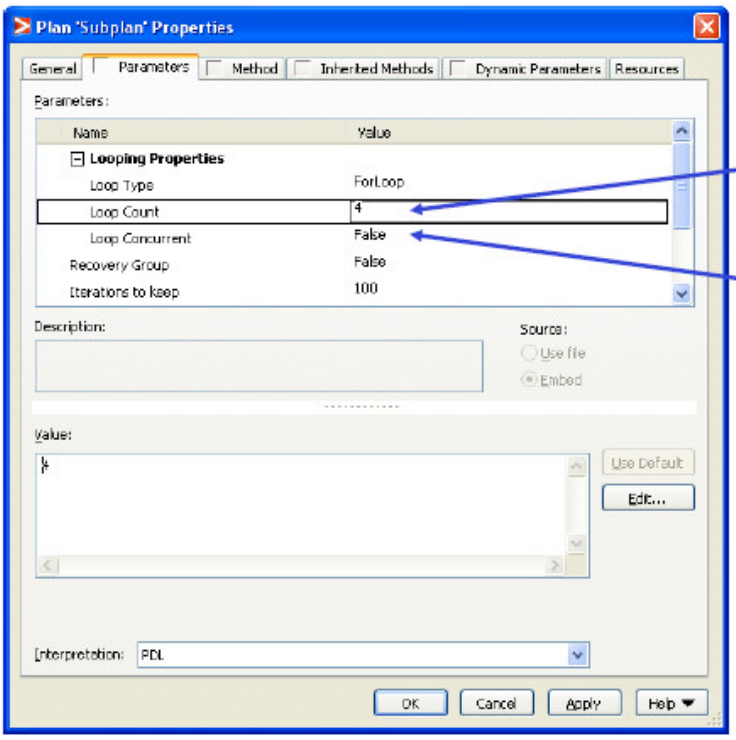
Save your solution as ex9_while_loop.plan

Track the execution of the plan in the GDE using View>Tracking Details.

Cleanly shutdown the plan by touching the file $AI_SERIAL/stop.control in the GDE using Run>Execute Command.

Subplans: For Loop

For loops run for a predetermined number of iterations.



Number of iterations

Each iteration can run sequentially or concurrently.

Subplans: Loop Parameters

Inside a loop, parameters give information about the current loop iteration.

AB_PLAN_LOOP_CURRENT_VALUE

Returns the name of the current iteration of a For Each loop, as taken from the **Loop value vector**.

For instance, if a plan loops over the vector [north, east, south, west], then during the first iteration **AB_PLAN_LOOP_CURRENT_VALUE** is set to **north**.

AB_PLAN_LOOP_INDEX

Returns a decimal value that specifies which loop iteration is running, **0** indicating the first iteration, **1** the second, and so on.

Exercise:

Exercise 10 – For Loop

Modify the plan from Exercise 9 (While Loop) such that:

- The While loop is replaced with a For loop that has two sequential iterations.
- The output filename has the loop index appended to it.

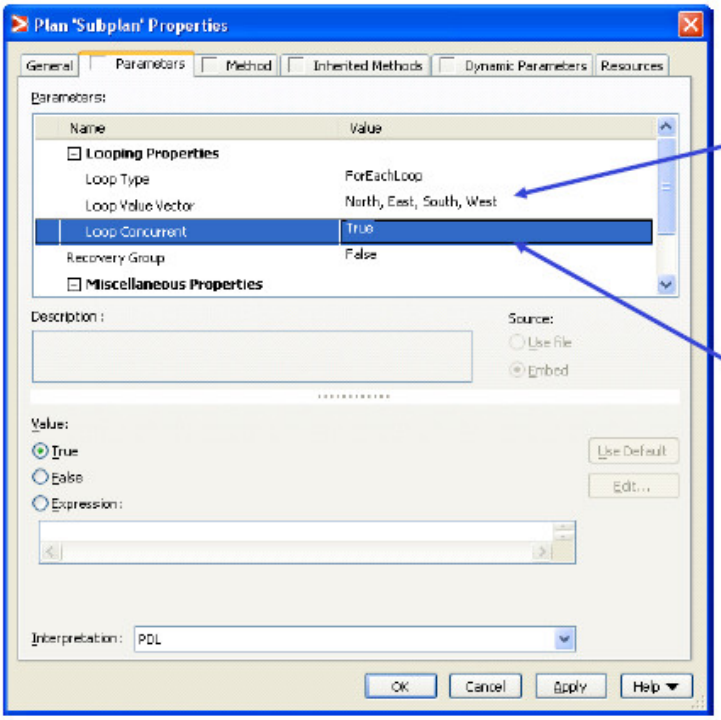
Save your solution as ex10_for_loop.plan

Run the plan.

Then change the iterations to be concurrent. What is different now?

Subplans: For Each Loop

For Each loops run against a vector list of items.



There is one iteration for each item in the vector. You can specify either an expression that returns a vector, or use the "List" interface and enter values, one per line.

As with a For loop, each iteration can run sequentially or concurrently.

Exercise: Exercise 11 – For Each Loop

Modify the plan from Exercise 10 (For Loop) such that:

- The For loop is replaced with a For Each loop
- The input file for each loop iteration is a file in $AI_SERIAL with the file pattern reformatted_trans*.dat
- The output file is written to $AI_SERIAL with a unique name such as out_transN.dat where N is the loop current value.

Save your solution as ex11_for_each_loop.plan

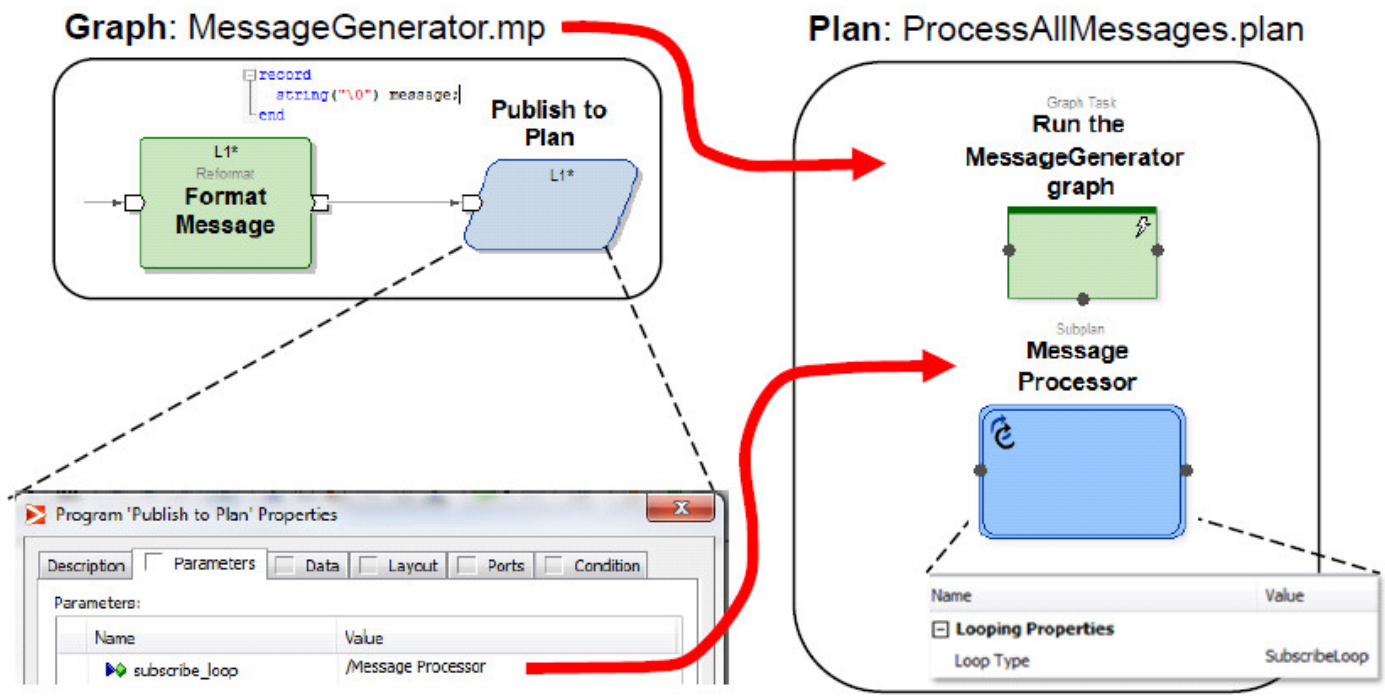
Hint: This DML function will be useful:

$[directory_listing(<path_to_dir>,"[!.]*")]

Subplans: Subscribe Loops

- When a plan runs a **subscribe loop**, it listens continuously for messages coming from an outside source.
- Each arriving message triggers a new loop iterations.
- Multiple messages generate multiple iterations that run concurrently.

Subplans: A design pattern for using subscribe loops



Subplans: Section Review

In this section, we discussed:

- Subplans
- Recovery Groups
- For Loops
- While Loops
- For Each Loops
- Subscribe Loops

Resources

Resources: Section Topics

- Resource contention
- Logical resources
- Defining and using resource pools

Resources: Limiting Concurrency

By default:

- Independent tasks run concurrently.

- All iterations of a concurrent loop run concurrently.

If you have numerous tasks or iterations, this can adversely affect performance.

Resources: Resources

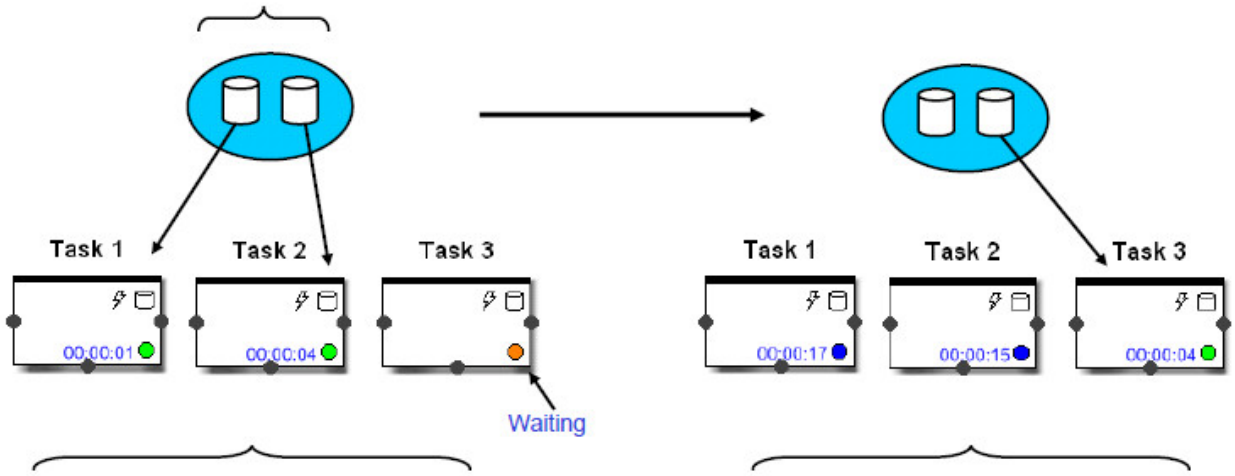
We can limit concurrency and other resource contention using Conduct>It resources.

- We define a logical resource.
- We specify the amount of the resource each task requires.
- A task or iteration will only run if it can acquire the necessary resources. Otherwise it waits for the resources to become available.

Resources:

Resources (cont'd)

Two units of resource available



Task 1 and Task 2 each acquire a unit, but Task 3 cannot and must wait.

Once one of the two other tasks completes, a unit of resource is available for Task 3.

All three tasks start simultaneously and require one unit of available resources.

Resources:

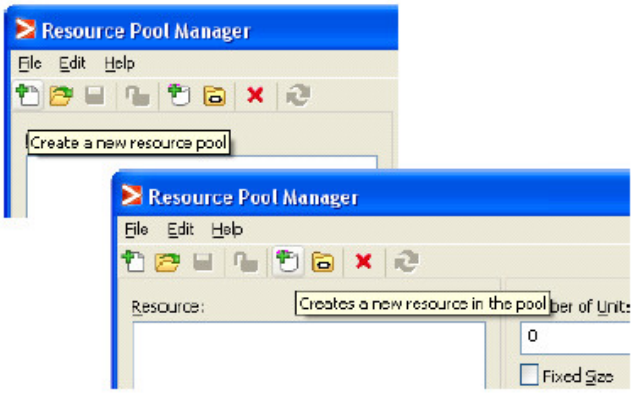
Resources (cont'd)

To use resources, you:

- **Create** logical resources in Conduct>It's Resource Pool Manager.
- **Save** resources in *pools* in your sandbox.
- **Assign** resources from pools to individual tasks and plans.

Resources: Resource Pools

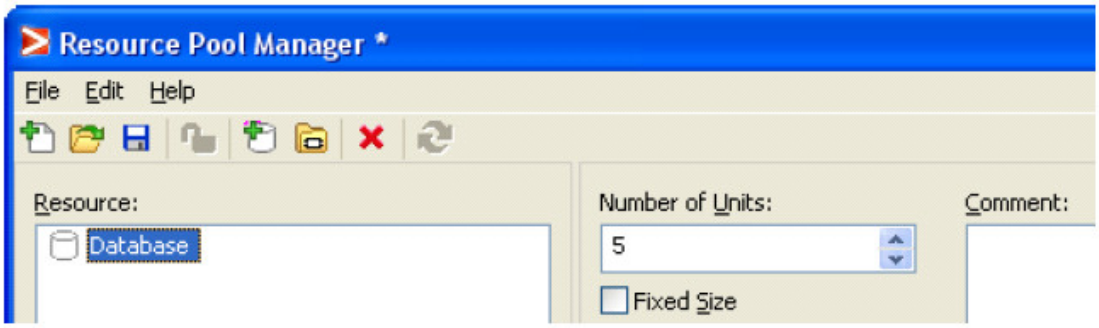
[1] Create resources.



Use the Resource Pool Manager to create sets of resources.

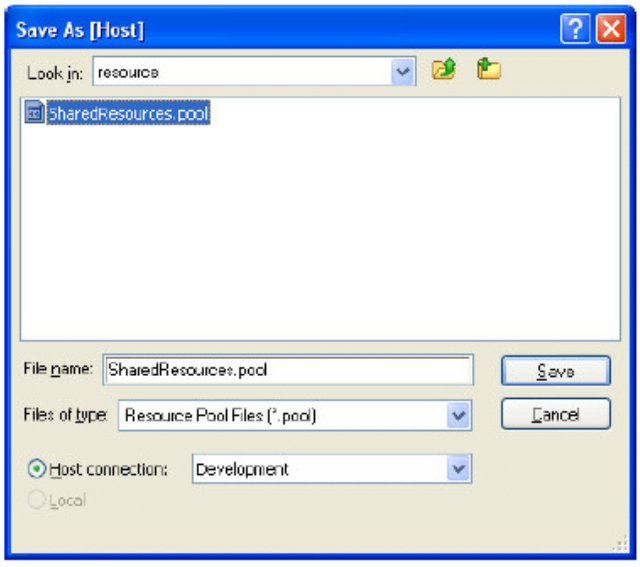
Resources: Resource Pools (cont'd)

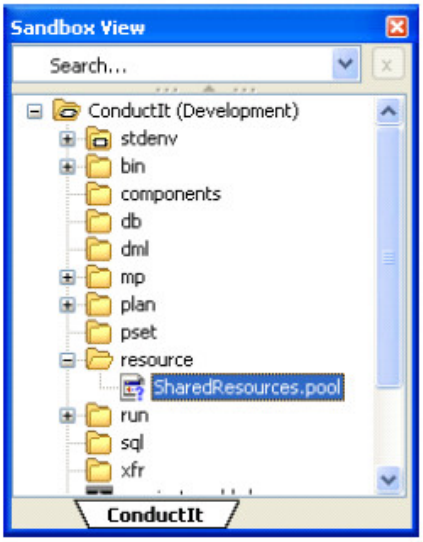
Resource Pool Manager lets you name a resource and assign it an arbitrary numerical value.



Resources: Resource Pools (cont'd)

[2] Save resources.



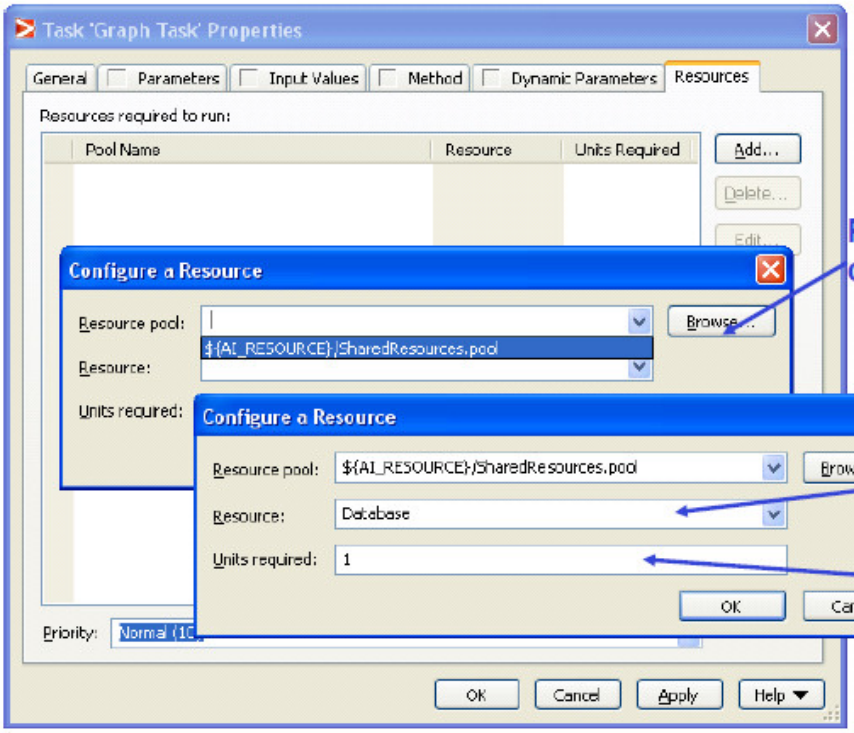


Save resources as **pools** into the resource folder of your sandbox.

Each pool can contain one or more resources.

Resources: Assigning Resources

[3] Assign resources to tasks.



Pulldown of resources from current sandbox

Resource Name

Units needed for one instance of this task

Resources: Logical Resources

Conduct>It resources are logical resources.
They can represent anything you choose.

Apples can represent:

- Number of available processes
- Amount of available memory
- Maximum number of simultaneous database connections
- Maximum number of concurrent file transfers

It is up to you to decide what a unit represents and how many units a task requires.

Resources: Managing Resources

The resource-admin command can be used to manage resources from the command line.

- Change the size of a pool to contain 99 apples:
  - **resource-admin adjust /ConductIt/resource/fruit.pool apple 99**
- Show list of tasks waiting for resources
  - **resource-admin queue**
- Increase priority of a resource request to “1”, the highest level
  - **resource-admin set-priority <handle> 1**

    (<handle> can be found from resource-admin queue output)

Exercise: Exercise 13 – Resources

Modify the plan from Exercise 10 (For Loop) such that:

- The number of concurrent loops is limited by a resource.

Save your solution as ex13_resources.plan

Resources: Section Review

In this section, we discussed:

- Concurrency
- Resource pools
- Using logical resources to control resource contention

Conclusion

Conclusion: Summary

In this course we have discussed:

- Plans
- Tasks
- Methods
- Failure and Recovery
- Dynamic Parameters
- Subplans and Loops
- Resources

Conclusion: Getting Help

Resources

- On-line help
- The Ab Initio Discussion Forum
- Ab Initio Support: **support@abinitio.com**

Include a package for support:

Help > Support > Create Package-for-Support

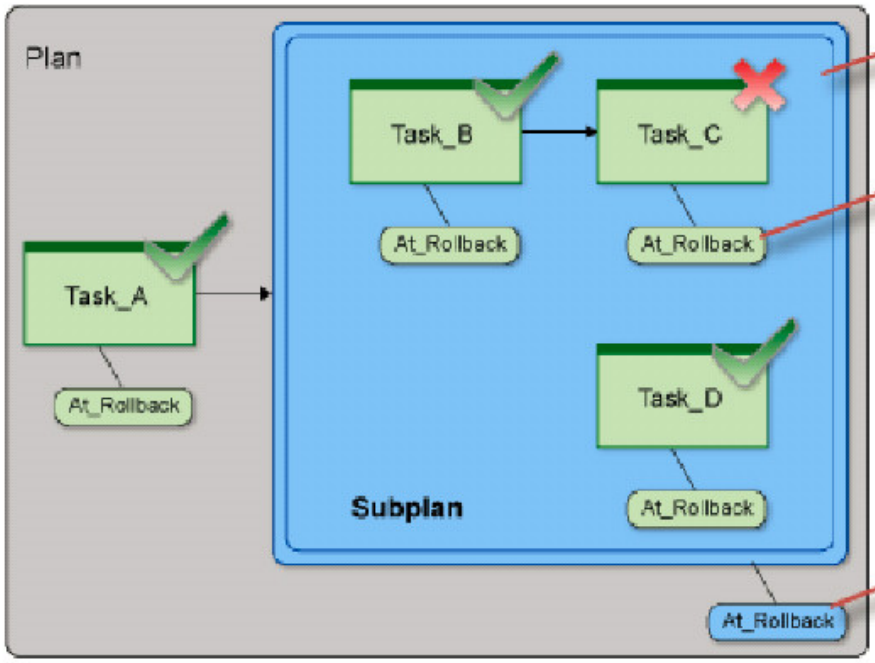
The package for support includes the plan, plans or graphs run in tasks, and other important information.

The End

Additional Topics

Subplans: Recovery Groups

You can use subplans to put tasks in a *recovery group*.



Recovery group

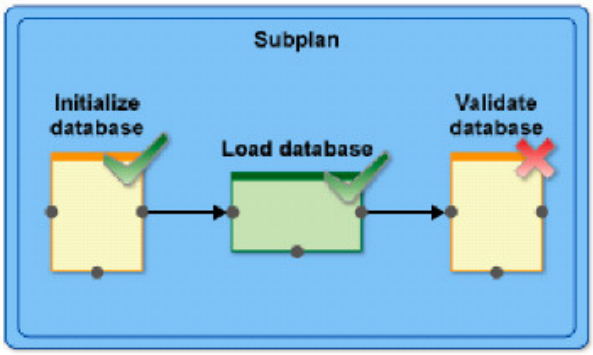
Task-level
At_Rollback method

If any task in the group fails, the subplan and all of the tasks within the group are rolled back.

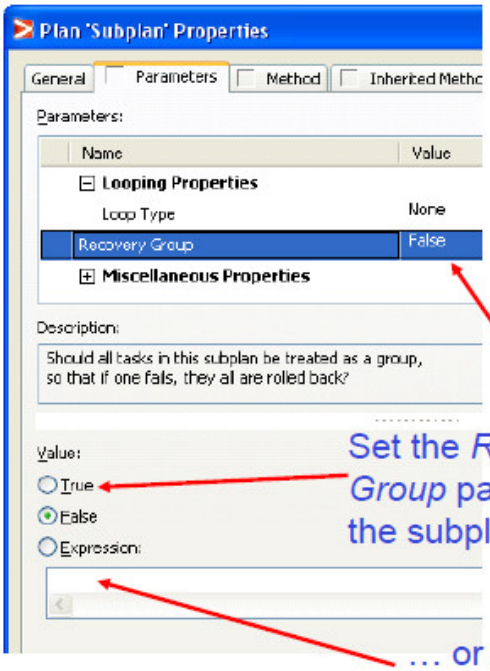
Subplan-level
At_Rollback method

Subplans: Recovery Groups (cont'd)

Use *recovery groups* to associate tasks that must succeed as a group.



Put the relevant tasks within a subplan.



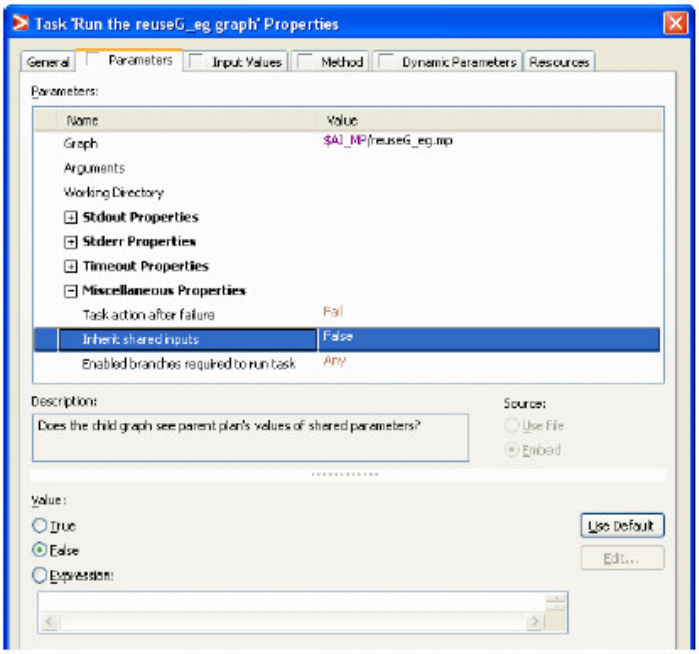
Set the *Recovery Group* parameter for the subplan to *True* …

… or specify an expression.

Parameters:

Using Plan Parameter Values in Graphs/Plans

You can configure the task to allow any shared sandbox input parameters to take on the value from the Plan environment.



Be Careful!

Changing Inherit shared inputs will cause graphs which are run through tasks to run differently from when they are run from the command-line.

Tasks: Task Output

Task output, errors, and tracking information can be stored with a variety of options.