Altiris™ Workflow Solution 6.5 from Symantec

Models Guide

Legal Notice

April 28, 2009

Copyright © 2008 Symantec Corporation. All rights reserved.

Symantec, the Symantec Logo are trademarks or registered trademarks of Symantec Corporation or its affiliates in the U.S. and other countries. Other names may be trademarks of their respective owners.

The product described in this document is distributed under licenses restricting its use, copying, distribution, and decompilation/reverse engineering. No part of this document may be reproduced in any form by any means without prior written authorization of Symantec Corporation and its licensors, if any.

THE DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID. SYMANTEC CORPORATION SHALL NOT BE LIABLE FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS DOCUMENTATION. THE INFORMATION CONTAINED IN THIS DOCUMENTATION IS SUBJECT TO CHANGE WITHOUT NOTICE.

The Licensed Software and Documentation are deemed to be commercial computer software as defined in FAR 12.212 and subject to restricted rights as defined in FAR Section 52.227-19 "Commercial Computer Software - Restricted Rights" and DFARS 227.7202, "Rights in Commercial Computer Software or Commercial Computer Software Documentation", as applicable, and any successor regulations. Any use, modification, reproduction release, performance, display or disclosure of the Licensed Software and Documentation by the U.S. Government shall be solely in accordance with the terms of this Agreement.

Symantec Corporation20330 Stevens Creek Blvd.Cupertino, CA 95014

http://www.symantec.com

Technical Support

Symantec Technical Support maintains support centers globally. Technical Support's primary role is to respond to specific queries about product features and functionality. The Technical Support group also creates content for our online Knowledge Base. The Technical Support group works collaboratively with the other functional areas within Symantec to answer your questions in a timely fashion. For example, the Technical Support group works with Product Engineering and Symantec Security Response to provide alerting services and virus definition updates.

Symantec's maintenance offerings include the following:

- A range of support options that give you the flexibility to select the right amount of service for any size organization

- Telephone and Web-based support that provides rapid response and up-to-the-minute information

- Upgrade assurance that delivers automatic software upgrade protection

- Global support that is available 24 hours a day, 7 days a week

- Advanced features, including Account Management Services

For information about Symantec's Maintenance Programs, you can visit our Web site at the following URL:

www.symantec.com/techsupp/

Contacting Technical Support

Customers with a current maintenance agreement may access Technical Support information at the following URL:

www.symantec.com/techsupp/

Before contacting Technical Support, make sure you have satisfied the system requirements that are listed in your product documentation. Also, you should be at the computer on which the problem occurred, in case it is necessary to replicate the problem.

When you contact Technical Support, please have the following information available:

- Product release level

- Hardware information

- Available memory, disk space, and NIC information

- Operating system

- Version and patch level

- Network topology

- Router, gateway, and IP address information

- Problem description:

- Error messages and log files

- Troubleshooting that was performed before contacting Symantec

- Recent software configuration changes and network changes

Licensing and registration

If your Symantec product requires registration or a license key, access our technical support Web page at the following URL:

www.symantec.com/techsupp/

Customer service

Customer service information is available at the following URL:

www.symantec.com/techsupp/

Customer Service is available to assist with the following types of issues:

- Questions regarding product licensing or serialization
- Product registration updates, such as address or name changes
- General product information (features, language availability, local dealers)
- Latest information about product updates and upgrades
- Information about upgrade assurance and maintenance contracts
- Information about the Symantec Buying Programs
- Advice about Symantec's technical support options
- Nontechnical presales questions
- Issues that are related to CD-ROMs or manuals

Maintenance agreement resources

If you want to contact Symantec regarding an existing maintenance agreement, please contact the maintenance agreement administration team for your region as follows:

Asia-Pacific and Japan contractsadmin@symantec.com

Europe, Middle-East, and Africa semea@symantec.com

North America and Latin America supportsolutions@symantec.com

Additional enterprise services

Symantec offers a comprehensive set of services that allow you to maximize your investment in Symantec products and to develop your knowledge, expertise, and global insight, which enable you to manage your business risks proactively.

Enterprise services that are available include the following:

Symantec Early Warning Solutions These solutions provide early warning of cyber attacks, comprehensive threat analysis, and countermeasures to prevent attacks before they occur.

Managed Security Services - These services remove the burden of managing and monitoring security devices and events, ensuring rapid response to real threats.

Consulting Services - Symantec Consulting Services provide on-site technical expertise from Symantec and its trusted partners. Symantec Consulting Services offer a variety of prepackaged and customizable options that include assessment, design, implementation, monitoring, and management capabilities. Each is focused on establishing and maintaining the integrity and availability of your IT resources.

Educational Services -Educational Services provide a full array of technical training, security education, security certification, and awareness communication programs.

To access more information about Enterprise services, please visit our Web site at the following URL:

www.symantec.com

Select your country or language from the site index.

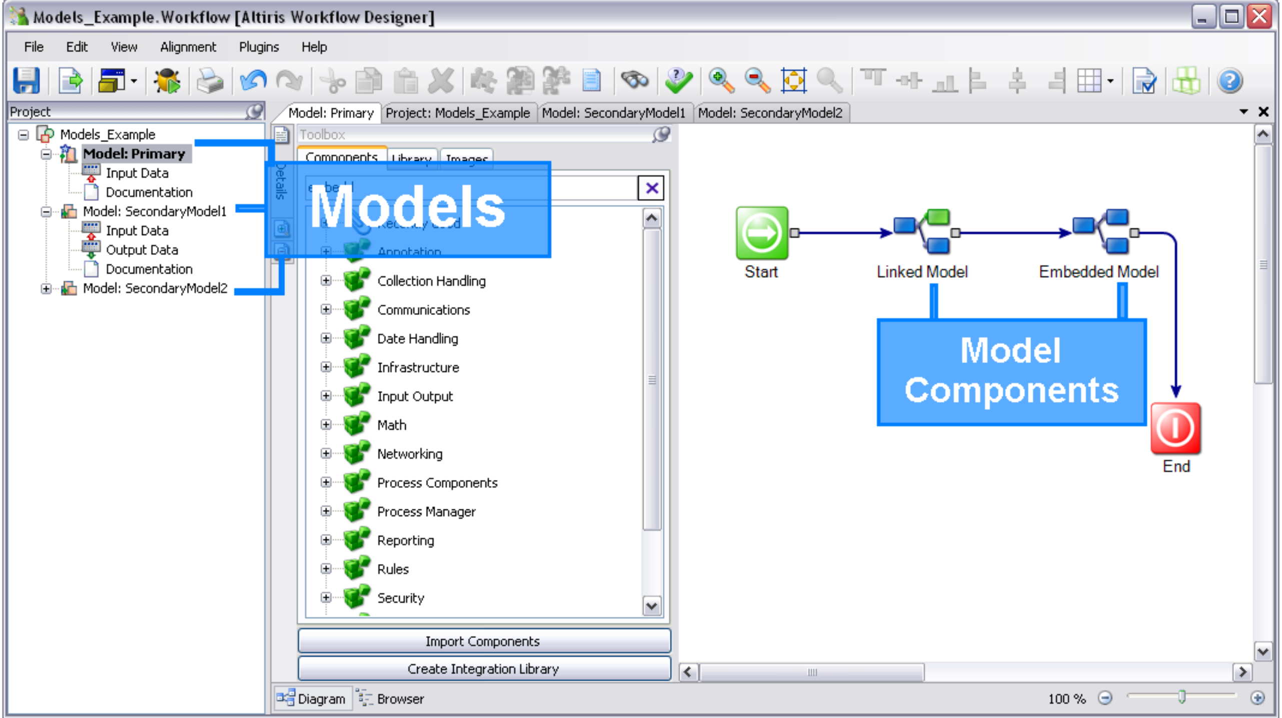
Chapter 1
Introduction

This document explores the Linked Model and Embedded Model components by describing them, showing examples, and providing instructions for using them.

Models vs. model components

Models should not be confused with model components. “Models” refers to either primary or secondary models that are listed in a project’s tree structure. Models are not components. Models contain logic on the project level. They can work with other models or operate independently of them. Every project has at least one model - a primary model - and can have an unlimited number of secondary models.

The screen shot below shows models and model components in Workflow Designer:



“Model components” refers to components that either contain logic on the component level, or point to a project model. Embedded models contain logic on the component level. Components with Embedded models include: Embedded Model component, Dialog Workflow (contains three embedded models), any component that can use a dynamic model (for example, Drop Down List and Add Items to Collection). Only two components point to a project model: the Linked Model component and the Dynamic Linked Model component.

See *About Linked and Embedded model components* (page 4)

About Linked and Embedded model components

The Linked Model and Embedded Model components are in some ways similar to other components: they have a representative icon, you can rename them, and they have input and an output paths. Yet they function very differently from a typical component. The Linked and Embedded Model components contain other components rather than performing any function themselves. Think of the Linked and Embedded Model components as independent processes that run within the context of another process.

Using Models provides three main benefits:

1. Breaking down larger processes into smaller, distinct sub-processes contained in linked models.

   This helps maintain organization and generally makes the main workflow more "readable."

2. Re-using a Linked or Embedded Model that is repeated throughout a process.

   Once you have configured a Linked Model, you can drag it into your process as many times as you like. This increases efficiency because repetitive portions of the process can be reused. Reusing Linked Models also aids in maintenance because if changes are required to this part of the process, they need to be made only once and will automatically be conveyed throughout the rest of the process.

   This function explains the name “Linked Model.” A given Linked Model is “linked” to all instances of itself in a project, so that changes made to one model occur in all instances of that model.

   You can also reuse an Embedded Model. Because Embedded Models contain their own model, you can reuse them by copying and pasting.

3. Caching

   Once a Linked or Embedded model has run, it can cache its data. If the model appears again in the process, the cached data is immediately available so the model doesn’t have to run again.

Differences between Linked and Embedded models

Although the Linked and Embedded Model components have similar functions, they have some notable differences.

Differences between Linked and Embedded Models

Linked Model	Embedded Model
Linked Model	Embedded Model
Points to a model in the project tree structure. Does not contain its own model. See *Creating a secondary model* (page 12)	Contains its own model.
Is linked to all recurrences of itself throughout the entire process, so changes are automatically distributed.	Is not linked to any recurrences of itself throughout the process. Changes cannot be distributed.
Cannot see any data outside of itself unless explicitly added to the Linked Model as input data. The containing process cannot see the Embedded Model's data unless it is explicitly declared as output data. Any data (except global data) that needs to come into the Linked Model must be added to the model's input data. Any data that needs to go out of the Linked Model must be added to the model's output data. See *Data contracts between models* (page 12)	Can see data preceding itself without adding it as input data. However, the containing process cannot see the Embedded Model's data unless it is explicitly declared as output data.
Can contain other Linked Models and Embedded Models. Cannot contain itself.	Can contain other Embedded Models but cannot contain Linked Models.
The project models to which Linked Models point can be invoked independently of the primary process.	The model contained in an Embedded model component can be invoked only dependently - only as it appears in the process.
You can reuse the function of a Linked Model component by adding them anywhere in the process and pointing them back to the same model.	You can reuse the function of an Embedded Model component by copying and pasting it anywhere in the process.
Can cache data based on duration and input parameter. Duration refers to time of caching retention, and input parameter refers to caching based on input value.	Can cache data based on duration only. Duration refers to caching retention time.

Parent and child models

Before you work with linked or embedded models you should be familiar with the concepts of a "parent model" and "child model." Very simply a parent model is a model which calls upon a second model to do some work and the child model is the model that is called. Embedded models are always child models because they are always invoked in the course of another model. Secondary models are not necessarily child models because they can be set as individual invocation targets.

See *Setting a secondary model as an invocation target* (page 8)

Where models live

Linked Models do not live in the same place that Embedded Models live. When you use a Linked Model in a process, the actual model lives in the project tree structure. When you use an Embedded Model in a process, the actual model lives in the workspace. This can be confusing, because both kinds of models have an icon in the workspace. Whereas Embedded Models actually live in their icons, icons for Linked Models merely point to the model that lives in the project tree structure.

This is an important distinction for understanding the difference between the kinds of models. When you edit a Linked Model (by adding components, deleting components, or any other configuration change), you are editing the model that lives in the project tree structure. Thus, when you edit a Linked Model, your changes are applied to every instance of the model throughout the process. When you edit an Embedded Model, you are editing the model that lives only in its icon in the process. Thus, when you edit an Embedded Model, your changes are not applied to any other Model in the process.

Chapter 2
Project models

About models

Models are containers of logic in a project. When you open a new project in Workflow Designer, a model is automatically generated: the primary model. The primary model appears in the project tree structure to the left of the workspace.

You can add other models to your project. These are called secondary models. Secondary models also appear as items in the project tree structure. They can work with other models or run independently (depending on their configuration). Every project has at least one model—a primary model—and can have an unlimited number of secondary models.

If secondary models are linked to other models with Linked Model or Dynamic Linked Model components, they are called "linked models."

Secondary models

"Secondary models" refers to any model you add to your project's tree structure. Using secondary models provides two main benefits:

1. Breaking down larger processes into smaller, distinct sub-processes

   This helps maintain organization and generally makes the main workflow more "readable."

2. Creating multiple independent models grouped together as one project

   Secondary models can be invoked independently of the project in which they were published. This is most useful when you have a number of smaller, related but independent processes. By creating these processes in secondary models in a single project, you can manage them more easily while designing and publishing. After publishing, you can invoke any of the secondary models through the same webservice layer (as long as you set the models to be invokable).

If you use secondary models in the first way, the models are dependent on each other, and so need to communicate. When the project runs, it does not run the models from top to bottom in the tree structure as you might think. Rather, it runs the models when they are called in the process. A Linked Model or Dynamic Linked Model component does the calling.

Also, when project models are dependent on each other, a data contract needs to be set up between them.

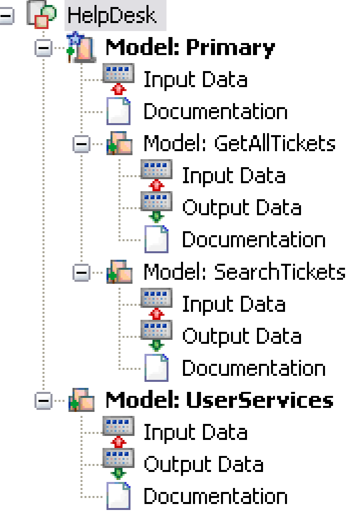
See *Data contracts between models* (page 9)

If you use secondary models in the second way, the models are independent of each other, and so do not need to communicate. The models are bundled together in one project, but do not rely on each other.

Used in this second way, no data contract needs to be set up between the models.

Creating a secondary model

Secondary models appear under the primary model in the project tree structure to the left of the workspace. Here's an example of a tree structure with a primary model and some secondary models:



Here we see a project called "HelpDesk." The primary model is "Model:Primary" and the secondary models are "Model: GetAllTickets," "Model:SearchTickets," and "Model:UserServices." Notice that the primary model and the last model on the list are both bold. This means they are invokable targets.

For information on setting a secondary model as an invocation target, see the Workflow Solution Best Practices guide.

www.altiris.com/support/documentation.aspx

To create a secondary model

1. In the project tree on the left, right-click on the project name.

   The project name is the top item in the project tree structure. For example, in the screen shot above, the project name is "HelpDesk."

2. Select **New Model.**

3. Name the model, then click **OK.**

Setting a secondary model as an invocation target

You can set any secondary model in your project as an invocation target. This means that you can invoke that model individually in the debug tool or in production. Setting secondary models as invocation targets is especially helpful for unit testing. If a model is an available invocation target, you can test it without having to run the entire process.

Models set as invocation targets must be able to run independently. In other words, they cannot run on their own if they rely on data or function existing in other models. However, if a model needs certain data to run, the debug tool lets you supply that data when you invoke the model. In production, a model must have access to all necessary data for it to run.

To set a secondary model as an invocation target.

1. In an open project, right click on a secondary model in the project tree structure in the left panel.

   If your project has no secondary model, add one.

   See *Creating a secondary model* (page 8)

2. Select **Set Model** As Invocation Target.

   The name of the model becomes bold in the tree structure.

Data contracts between models

Secondary models do not automatically share data with other models. Every project model is actually a separate process with variables that exist only within itself. Project models do not automatically understand how to relate to one another unless you set up a data contract between them. A data contract is a declaration of what variables a model needs as input data, and what variables it will return as output data. This contract is configured in the input and output data properties of a model. Locate these properties under the model name in the project tree on the left.

Adding input data to a secondary model

To determine what variables to use for input data, consider this question: "What variables exist outside of this model that its needs to have in order to accomplish its work?" For example, if the model needs to compare two variables that were gathered or created in another model, the first model must have access to those variables. Once you have answered this question, you are ready to add data to the secondary model.

If the secondary model is sharing data with other models in the project, the data still needs to be mapped through a Linked Model or Dynamic Linked Model component.

To add data to input data

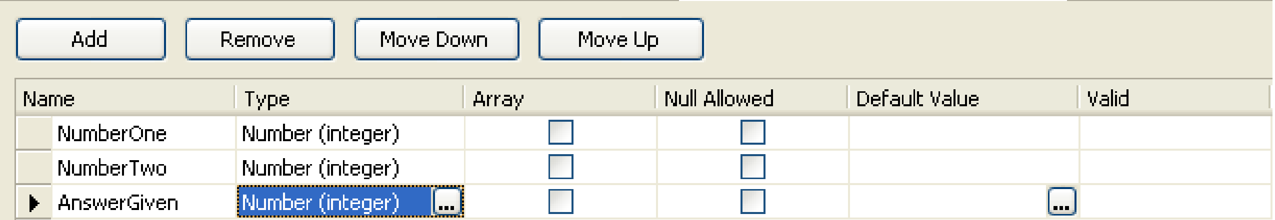
1. Under a Linked Model, click **Input Data** in the project tree.

   If you do not see **Input Data**, expand the items under the Linked Model.

2. In the right panel, click the **Add** button

3. Add and configure as many variables from the parent model as that Linked Model needs.

   You can call these variables anything you want, but the best practice is to use the variable names from the process. For example, if the Linked Model needs to use a process variable called **`Value1`**, create a variable in input data called **`Value1`**.



Here we see three added and configured variables. Each variable needs a name, a type, and three other parameters (array, null, and default value). Use the

checkboxes to indicate if the variable is an array or if it is nullable. The default value field is optional. Remember, each input variable needs to be of the same type as the variable to which it will be mapped.

Adding output data to a secondary model

If a secondary model needs to pass data out back to the parent model, it must be configured with output data.

There are some scenarios in which output data is not necessary. For example, if the secondary model writes data off to a database instead of handing it back to the process, there is no need to use output data.

Output data is configured similarly to input data. Configure one variable for each piece of data that the secondary model passes out.

To add output data

1. Select **Output Data** in the project tree

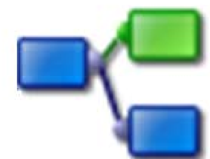
   If you do not see **Output Data**, expand the items under the model.

2. In the right panel, click the **Add** button
3. Add and configure as many variables as need to be passed out.

Here we see three added and configured variables. Each variable needs a name, a type, and three other parameters (array, null, and default value). Use the checkboxes to indicate if the variable is an array or if it is nullable. The default value field is optional.

Chapter 3
Linked model

About the Linked Model component



The Linked Model component points to a model in the project tree structure. The component could also be called the “Link Models” component, because it links two project models together. Think of the Linked Model component as a trigger that causes a secondary model to run within the context of another model. It serves as a command inserted into the middle of the process flow. When the process flow hits the Linked Model, it says, “Run model X.” When that model has run, the process flow resumes after the Linked Model component. You can use Linked Model in your primary model or even other linked models.

Using Linked Models provides three main benefits:

1. Breaking down larger processes into smaller, distinct sub-processes contained in secondary models that are linked with Linked Model components.

   This helps maintain organization and generally makes the main workflow more "readable."

2. Re-using a secondary model throughout a process.

   Once you have configured a secondary model, you can invoke it in your process as many times as you want. This increases efficiency by letting you re-use repetitive sections of logic. Reusing secondary models also makes process changes easier. If you need to change a secondary model, you have to change it only once, even if it is used multiple times throughout the process.

   This function explains the name “Linked Model.” A given Linked Model is “linked” to all instances of itself in a project, because all instances point to the same secondary model.

3. Caching

   Once a Linked Model component has run, it can cache its data. If the model appears again in the process, the cached data is immediately available so the model doesn’t have to run again.

You can add a Linked Model component either by dragging the component from the toolbox, or by dragging a project model from the project tree structure. If you drag it from the toolbox, the component needs to be configured to a project model. If you drag a project model onto the workspace, a Linked Model component appears automatically.

Because Linked Models point to secondary models, at least one secondary model must exist for a Linked Model component to work.

See *Creating a secondary model* (page 12)

The concepts of a "parent model" and "child model" are very important when working with Linked Models.

See *Parent and child models* (page 6)

Linked Model setup

There are three phases to setting up a Linked Model:

1. Creating a model in the project tree structure

   See *Creating a secondary model* (page 12)

2. Configuring the Linked Model's input and output data

   See *Data contracts between models* (page 12)

Creating a secondary model

Linked Model components point to secondary models that exist in the project tree structure. Thus, you must first create a secondary model before you can use a Linked Model component.

Although a model appears only once in the project tree, you can use it as many times as you want by adding Linked Model components that point to it. Also, you can link to any secondary model in any other model in your project.

To create a secondary model

1. In the project tree on the left, right-click on the project name.

   The project name is the top item in the project tree structure.

2. Select **New Model.**

3. Name the model, then click **OK.**

   The new model appears in the project tree structure.

4. Click on the name of the new model and drag it onto the workspace.

   This creates a Linked Model component that points to the new model.

Data contracts between models

Secondary models do not automatically share data with other models. Every project model is actually a separate process with variables that exist only within itself. Project models do not automatically understand how to relate to one another unless you set up a data contract between them. A data contract is a declaration of what variables a model needs as input data, and what variables it will return as output data. This contract is configured in the input and output data properties of a model. Locate these properties under the model name in the project tree on the left.

Adding data to input data

If a secondary model needs external data to do its work, the model must be configured with output data. All data that needs to come into a secondary model must be added to output data.

To determine what variables to use for input data, consider this question: "What variables exist outside this Linked Model that needs to know about in order to accomplish its work?" (For example, if the Linked Model needs to compare two variables that were gathered or created in the parent model, the Linked Model must have access to those variables.) Once you have answered this question, you are ready to add data to the Linked Model.

To add data to input data

1. Under a secondary model, click **Input Data** in the project tree.

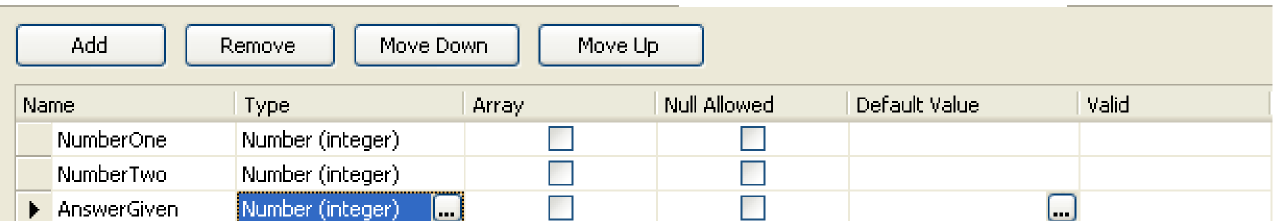
   If you do not see **Input Data**, expand the items under the model.

2. In the right panel, click the **Add** button.

3. Add and configure as many variables from the parent model as that Linked Model needs.

   You can call these variables anything you want, but the best practice is to use the variable names from the process. For example, if the Linked Model needs to use a process variable called **`Value1`**, create a variable in input data called **`Value1`**. These two values will be mapped in a later step.

   See *Mapping input data* (page 13)



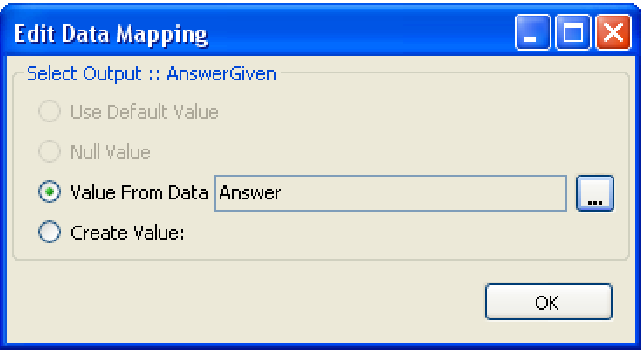
Here we see three added and configured variables. Each variable needs a name, a type, and three other parameters (array, null, and default value). Use the checkboxes to indicate if the variable is an array or if it is nullable. The default value field is optional. Remember, each input variable needs to be of the same type as the parent model variable to which it will be mapped.

Mapping input data

Before you can use a process variable in your Linked Model, you must map the process variable's value into its corresponding input data variable. You must create input variables before you can map any variables.

See *Adding data to input data* (page 13)

While mapping input data, you will see a dialog like this:



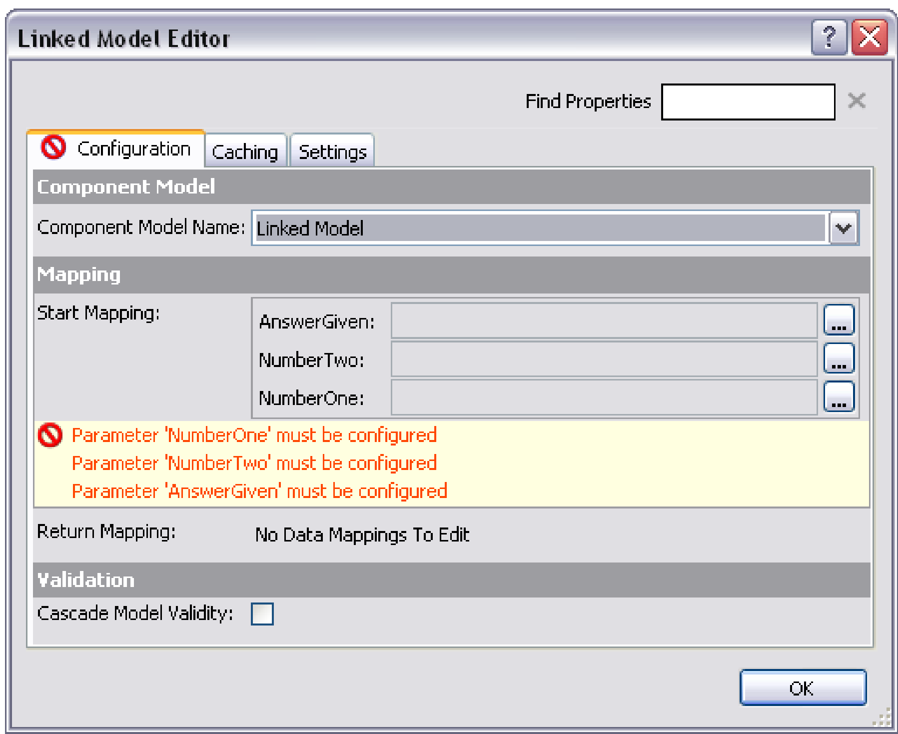
This dialog lets you decide where data comes from. The Value From Data option is used most frequently. Below is an explanation of each option.

Use Default Value	Uses the default value of the variable in question (in this case **CorrectAnswer**). This is an option only if you set a default value on an output variable.
Null Value	Designates a null value of the variable in question (in this case **CorrectAnswer**). This is an option only if you set an output variable to allow a null value.
Value From Data	Lets you pick a variable from the Embedded Model to map into the variable in question (in this case **CorrectAnswer**). This is the most commonly used option. Use this option to map the value of a variable in the Embedded Model into an output variable.
Create Value	Assigns a constant value to the variable in question (in this case **CorrectAnswer**).

To map input data

1. Open the Linked Model component editor

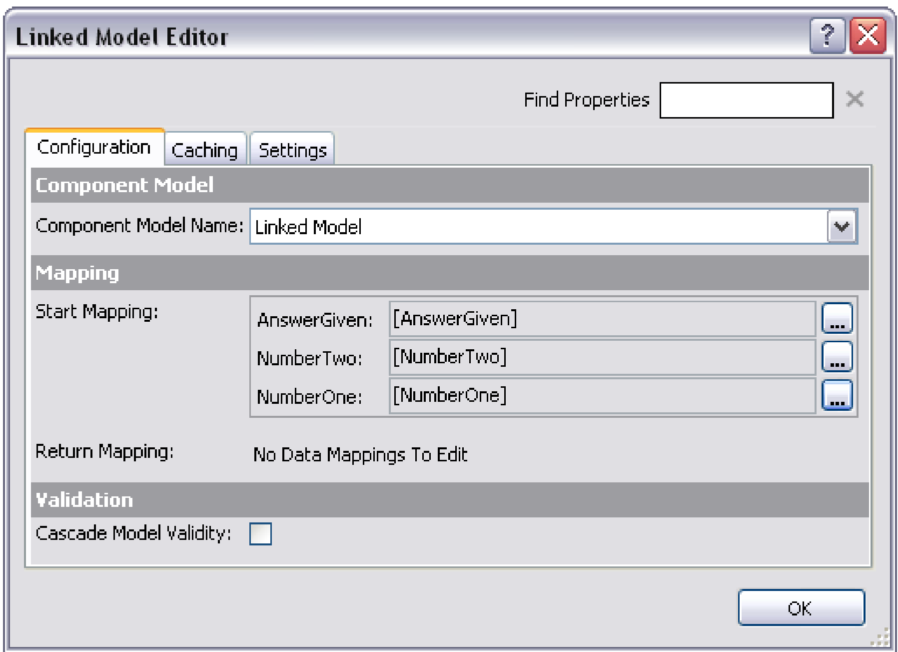
   To do this, right-click on the component in the workspace and select **Edit Component**. When the editor opens, in the Configuration tab under Mapping, you will see all of the variables you added to input data under "Mapping."



Here we see three variables under the mapping section (**AnswerGiven**, **NumberTwo**, and **NumberOne**). These variables appear here because they were added to the linked model's input data.

See *Adding data to input data* (page 13)

2. Click the **[...]** button for a variable you want to map.
3. Select **Value From Data** and click the **[...]** button.
4. Select the process variable you want to map into your input variable and click **OK.**



Here we see a Linked Model editor showing three mapped variables: **`AnswerGiven`**, **`NumberTwo`**, and **`NumberOne`**. The variable names on the left side represent three variables created in the model input data. The matching variable names on the right side represent the process variables whose values are being mapped.

5. Click **OK.**

Adding data to output data

If a linked model needs to pass data back to the parent model, it must be configured with output data. All data that needs to come out of the model must be added to output data.

There are some scenarios in which output data is not necessary. For example, if the linked model writes data off to a database instead of handing it back to the process, there is no need to use output data.

To determine what variables to add to output data, consider this question: "What variables exist inside this linked model that other models need to know about?" (For example, if the linked model compares two variables and renders an outcome to be used outside the model, that data should be passed out as output data.) Once you have answered this question, you are ready to add data to the linked model.

Output data is configured similarly to input data. Configure one variable for each piece of data that the child model is passing back.

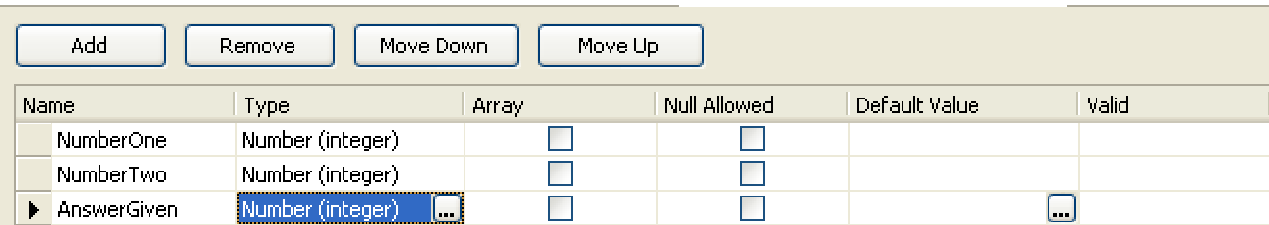
To add output data

1. Select **Output Data** in the project tree

   If you do not see **Output Data**, expand the items under the linked model.

2. In the right panel, click the **Add** button

3. Add and configure as many variables from the linked model as that it needs to output.



Name	Type	Array	Null Allowed	Default Value	Valid
NumberOne	Number (integer)	☐	☐		
NumberTwo	Number (integer)	☐	☐		
AnswerGiven	Number (integer)	☐	☐		

Here we see three added and configured variables. Each variable needs a name, a type, and three other parameters (array, null, and default value). Use the checkboxes to indicate if the variable is an array or if it is nullable. The default value field is optional. Remember, each variable needs to be of the same type as the variable to which it will be mapped.

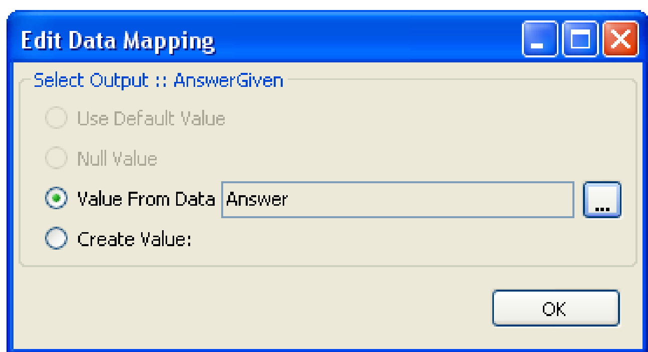
Mapping output data

Data mapping refers to a transfer of value, where one variable is pointed at another variable which takes the value of the first. For example, if a variable called **`Variable1`** with a value of "person" is mapped into a variable called **`Variable2`**, "person" becomes the value of **`Variable2`**.

Before the process can use a variable from your linked model, you must map the variable into its corresponding process variable. This mapping is done in the Linked Model component editor. You can open the Linked Model component editor by right-clicking the component in the workspace and selecting **Edit Component**.

Although mapping output data is similar to mapping input data, except that it has to be done in two places: the Linked Model component editor, and the end component(s) inside the linked model. Mapping output data from the Linked Model component editor creates a "contract" between the linked model and the parent model. Mapping output data from the linked model end component actually exposes the variables to the parent model.

While mapping output data, you will see a dialog like this:



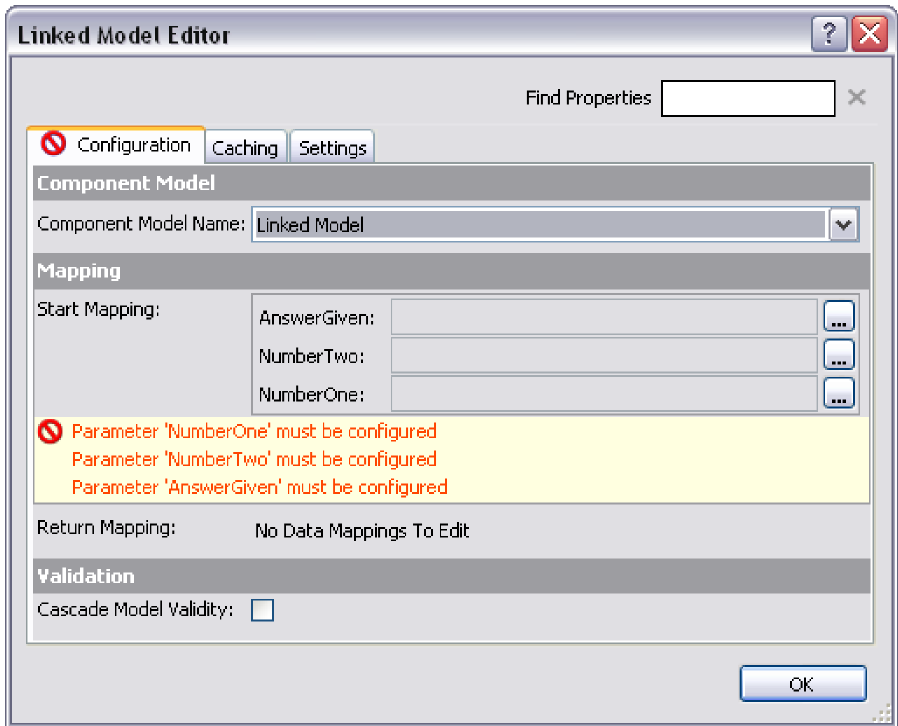
This dialog lets you decide where data comes from. The Value From Data option is used most frequently. Below is an explanation of each option.

Use Default Value	Uses the default value of the variable in question (in this case **CorrectAnswer**). This is an option only if you set a default value on an output variable.
Null Value	Designates a null value of the variable in question (in this case **CorrectAnswer**). This is an option only if you set an output variable to allow a null value.
Value From Data	Lets you pick a variable from the Embedded Model to map into the variable in question (in this case **CorrectAnswer**). This is the most commonly used option. Use this option to map the value of a variable in the Embedded Model into an output variable.
Create Value	Assigns a constant value to the variable in question (in this case **CorrectAnswer**).

To map output data from the Linked Model editor

1. Open the Linked Model component editor

   To do this, right-click on the component in the workspace and select **Edit Component**. When the editor opens, in the Configuration tab under Mapping, you will see all of the variables you added to output data.

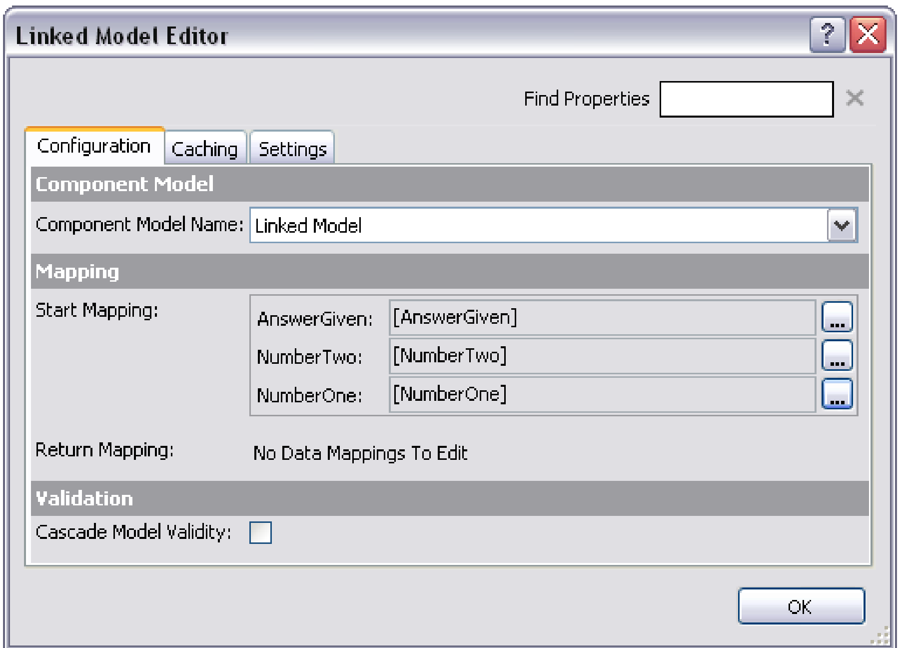


Here we see a Linked Model component editor showing three mapped input variables: **AnswerGiven**, **NumberTwo**, and **NumberOne**. We also see the same three variables as unmapped output variables. In this screen, all the variables in the

left column represent values added to a Linked Model's input or output data. The variables in the right column represent variables outside the Linked Model in the parent model.

Start mapping refers to input variables; return mapping refers to output variables.

2. Click the **[...]** button for a variable you want to map.
3. Select **Value From Data** and click the **[...]** button.
4. Select the process variable you want to map into your Linked Model variable and click **OK.**



Here we see all the input and output variables mapped. Again, the variable names on the left side represent the variables created in the Linked Model input and output data, and the matching variable names on the right side represent the process variables whose values are being overwritten.

Start mapping refers to input variables; return mapping refers to output variables.

5. Click **OK.**

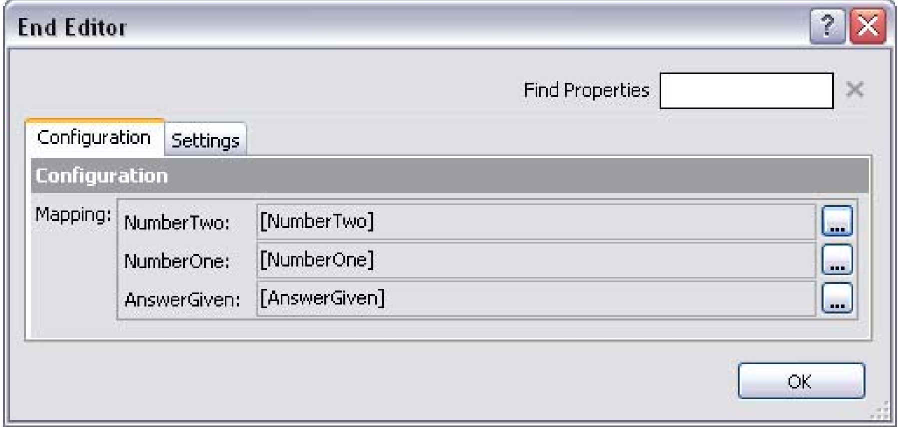
To map data from the Linked Model end component

1. Open the linked model

   To do this, click on the model to which the Linked Model component points.
2. Open the end component in the model.

   To do this, double-click on the end component inside the model.
3. Click the **[...]** button for a variable you want to map.
4. Select **Value From Data** and click the **[...]** button.

5. Select the variable to which you want to map your model variable and click **OK.**



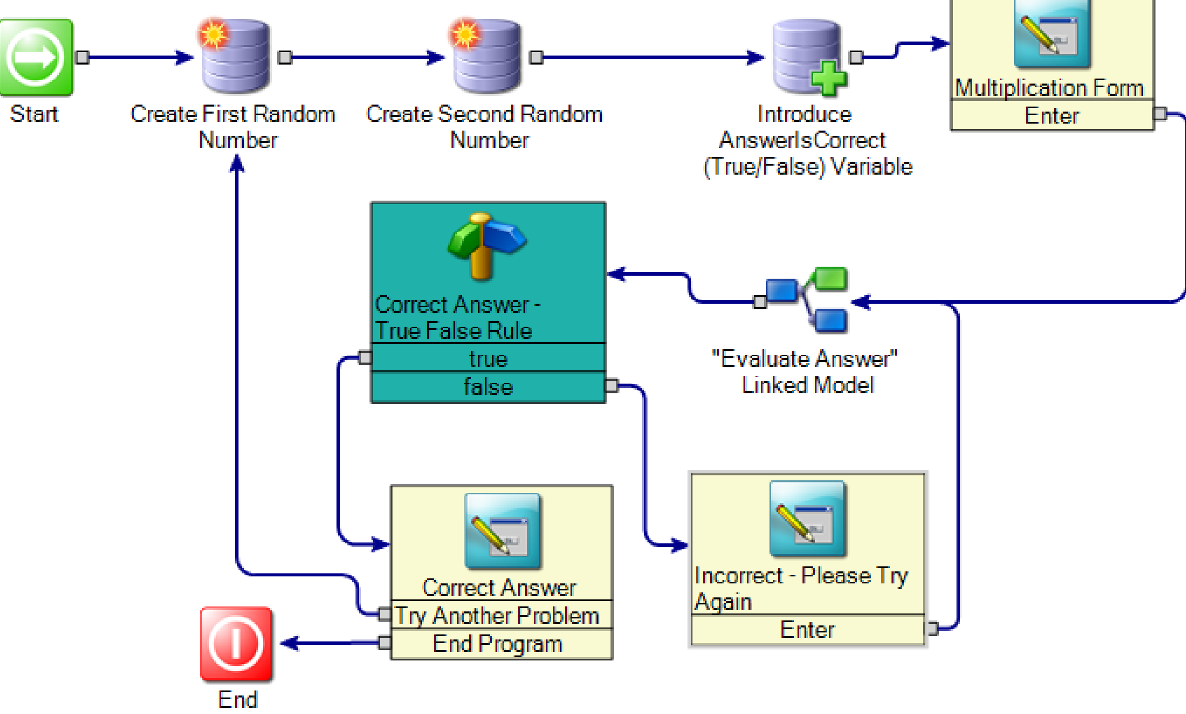
Here we see the editor of an End component in a Linked Model. It shows three mapped variables: **`AnswerGiven`**, **`NumberTwo`**, and **`NumberOne`**. The variable names on the left side represent three variables created in the linked model output data. The matching variable names on the right side represent the process variables whose values are being overwritten.

6. Click **OK.**

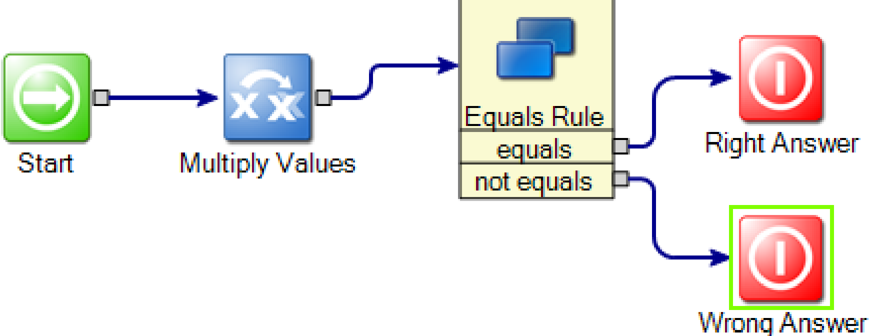
Example use of the Linked Model component

Scenario: A flash card application helps elementary students learn multiplication tables. Students are shown two random numbers and are asked to multiply them and provide the answer. Each time a student clicks the "Enter" button, the application must determine whether the student's answer was correct. If it was not, the student is given another chance to answer.

Process: This process is built in a Webforms-type project. The process generates two random numbers between 0 and 10, displays the numbers in a Webform, and asks the student to multiply the numbers and provide the correct answer. Using a separate model (the Linked Model), the process evaluates the answer and sets the value of a variable called **`CorrectAnswer`** to either true or false. The two models used in this workflow are called "Primary Model" and "Evaluate Answer." Here's what the primary model looks like:



Here's what the Linked Model looks like:



NOTE
In this scenario, a Linked Model component is be used simply to "clean up" the process by isolating a chunk of logic. Normally, an amount of logic this small would not require using a Linked Model. This is a simple example aimed at showing step-by-step configuration, and not necessarily best practice.

Primary Model (parent model)

Components Used in this Model:

- Create Random Number
- Add New Data Element

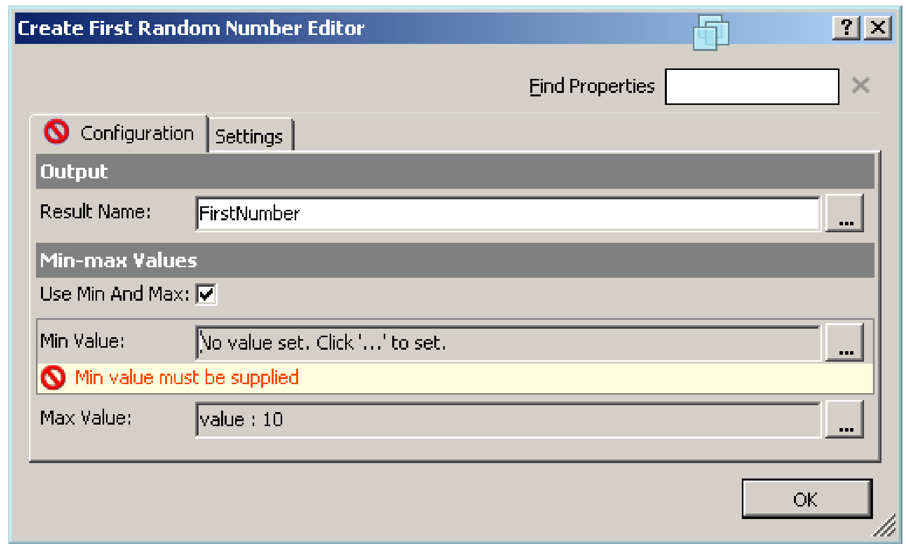
- Form Builder
- Linked Model
- True False Rule

For more information on these components, see the Workflow Solution Component Examples guide.

www.altiris.com/support/documentation.aspx

The process begins with two Create Random Number components. These components create the numbers that the student is asked to multiply.

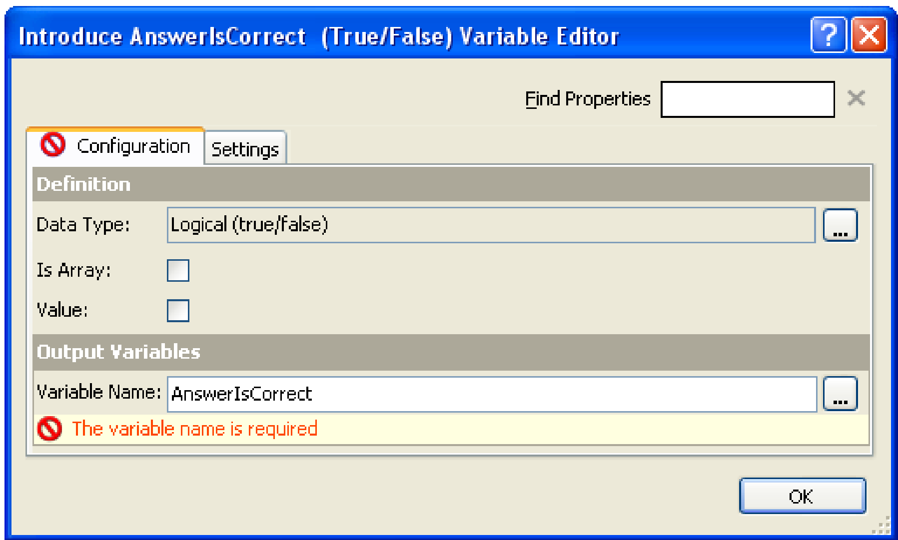
Let’s go into the first Create Random Number rule:



Both Create Random Number components are configured the same way (except for the "Result Name," which must be unique). We've checked the "Use Min and Max" feature because we want to limit the value of the numbers to be between 0 and 10. The "Min Value" by default is null. We will assign 0 as a constant value for this field.

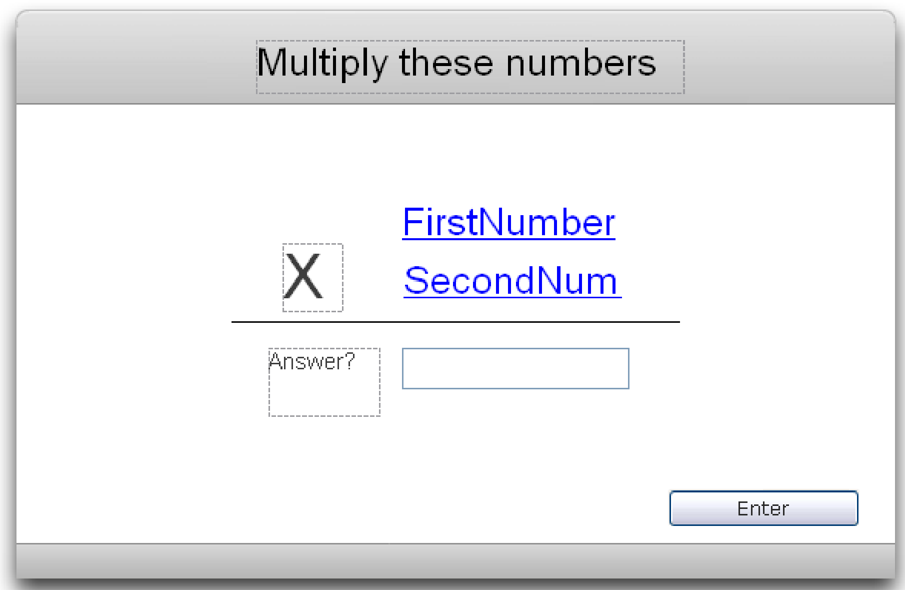
Next, an Add New Data Element component introduces a variable into the process that will be used later. The variable is called **`AnswerIsCorrect.`** It is a Logical (True/False) type variable because it represents the accuracy of the student’s answer (which is either true or false). This variable will receive the data that is passed back from the Linked Model after the Linked Model has evaluated the student’s answer.

Here’s what the editor looks like:



Next, a Form Builder component presents the multiplication problem to the student. This process uses three From Builders throughout this process. They are named "Multiplication Form", "Correct Answer" and "Incorrect - Please Try Again" to describe what their purpose is in the workflow.

Here is the first form at design-time:



The textbox takes in the student's answer and outputs it as a variable called **AnswerGiven**.

Linked Model ("Evaluate Answer")

Components Used in this Model:

- Multiply Values

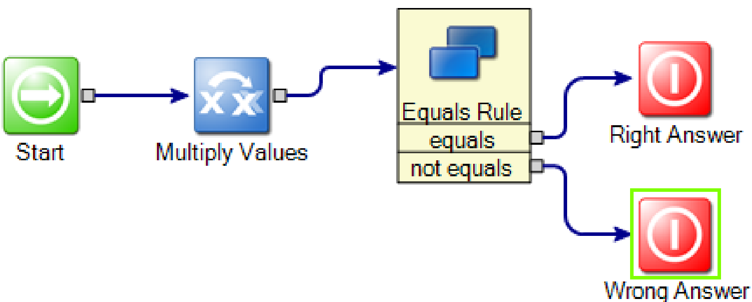
- Equals Rule

For more information on these components, see the Workflow Solution Component Examples guide.

www.altiris.com/support/documentation.aspx

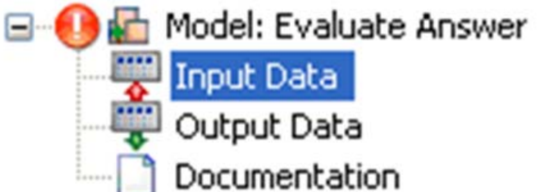
In this scenario the parent model is "Primary Model" and the child model is the Linked Model "Evaluate Answer." In a real world scenario, a linked model will likely carry out a more complex process. However, as this is an example project only, our linked model is very small and is carrying out a relatively simple task.

Let's take another look at the Linked Model process:



Linked Model data

This Linked Model calculates the correct answer and compares it with the user-supplied answer. Before this process can do anything, it needs variables from the process. These are supplied through the Linked Model input data.



Input data

In this case, the purpose of the child model is to evaluate whether the answer to a multiplication problem is correct. Therefore, the data we have to supply to the Linked Model are the randomly created numbers (**`FirstNumber`** and **`SecondNumber`**), and the number the user supplied as an answer (**`Answer`**).

For clarity, here is a list of parent model variables that the Linked Model needs to perform its function:

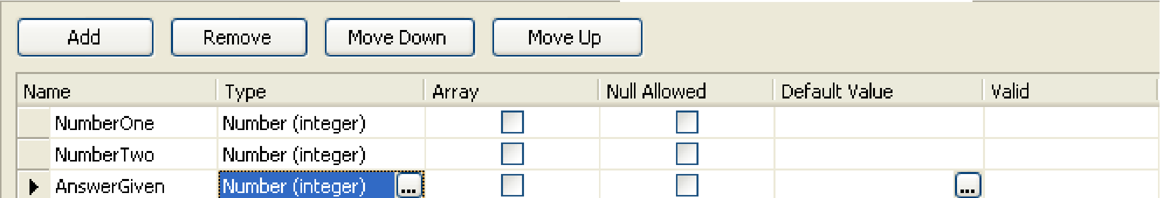
- FirstNumber
- SecondNumber
- Answer

All of these parent model variables are of type "Number (Integer)."

Now, let's take a look at the configured input data on the Linked Model. Remember, variables here cannot be selected from a list of process variables; new variables must be added to the Linked Model's input data. These variables can have the same name as the process variables they correspond with, or different names. Either way, they must be mapped to communicate with their corresponding process variables. In this example, we use different names so the mapping function is seen more clearly.

See *Adding data to input data* (page 13)

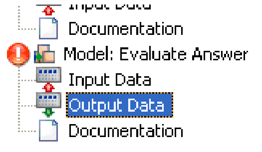
See *Mapping input data* (page 13)

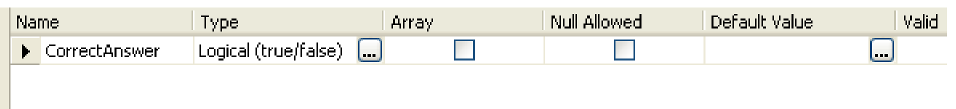


Because the three process variables required in the Linked Model are integer type data, the added input variables shown here are also integer type data.

Output data

After we've configured all of the input variables, we must configure the Linked Model's output data. In our scenario, the child model will output a variable called **`CorrectAnswer`** of type "Logical (true/false)" (True = the user's answer was correct, False = the user's answer was incorrect). This will make more sense when we see the function of the End components.





This variable has to be mapped back into the parent model variable **`AnswerIsCorrect`**.

NOTE
At this point in the setup we recommend configuring the End components of the linked model. However, for the sake of organization, we will discuss the End component configuration later.

See *Adding data to output data* (page 16)

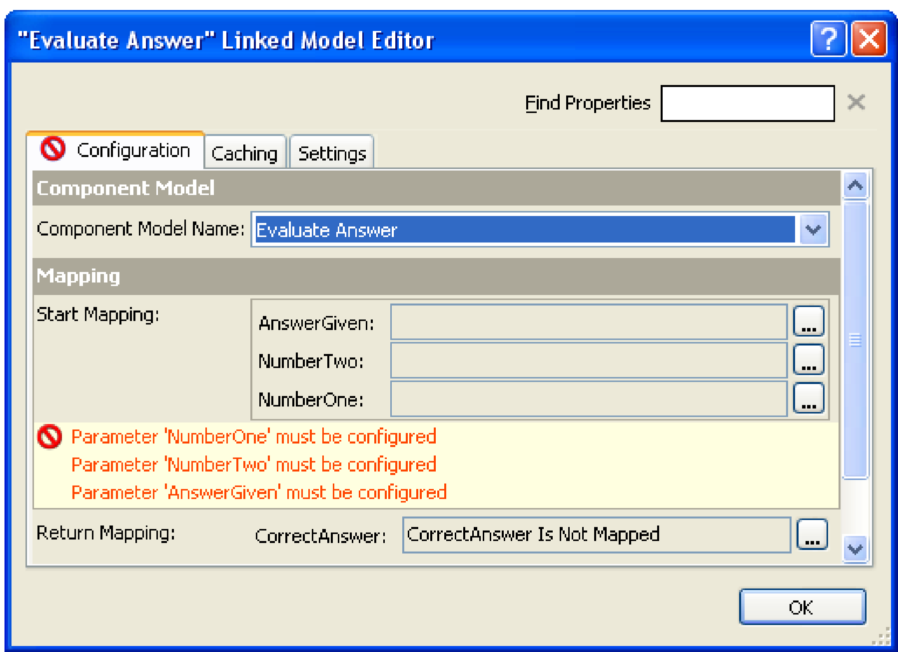
See *Mapping output data* (page 17)

Data mapping

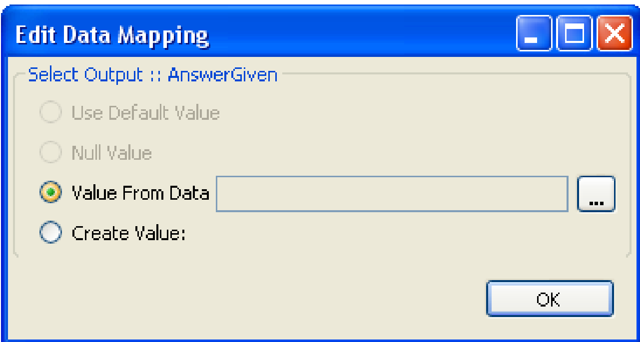
After the input and output data on the Linked Model is added and configured, it must be mapped into the primary model.

Instead of viewing pre-mapped data, we will go through one detailed mapping from beginning to end. Below is a series of screen shots that shows how to do this. You must repeat this process for all input and output variables.

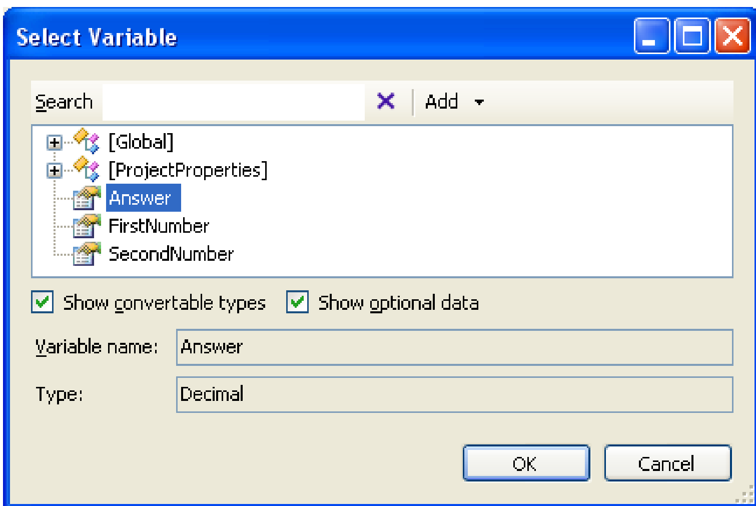
In the Linked Model editor, we see that data has been added, but not mapped:



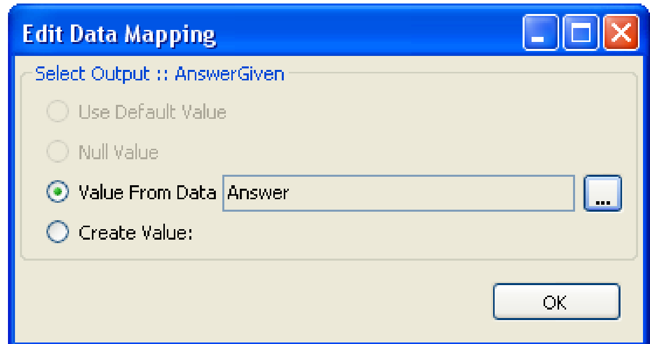
After you click the ellipse button next to the child model variable name, you are prompted to choose the source variable to which the input variable is mapped.

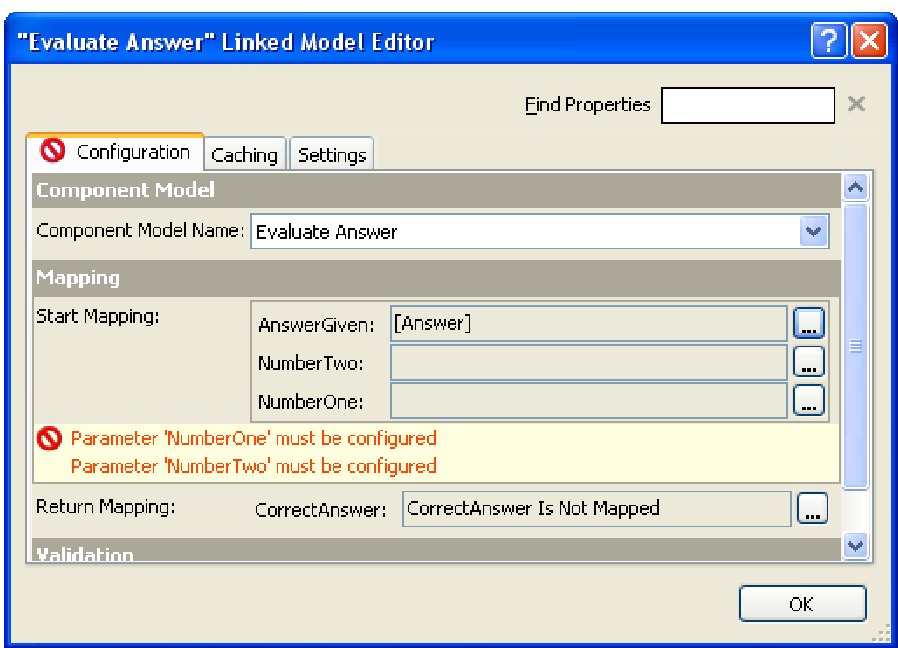


In this case we want to map the parent model variable **Answer** to the child model variable **AnswerGiven**, so we select "Value from Data" and click the ellipse button in line with that option.

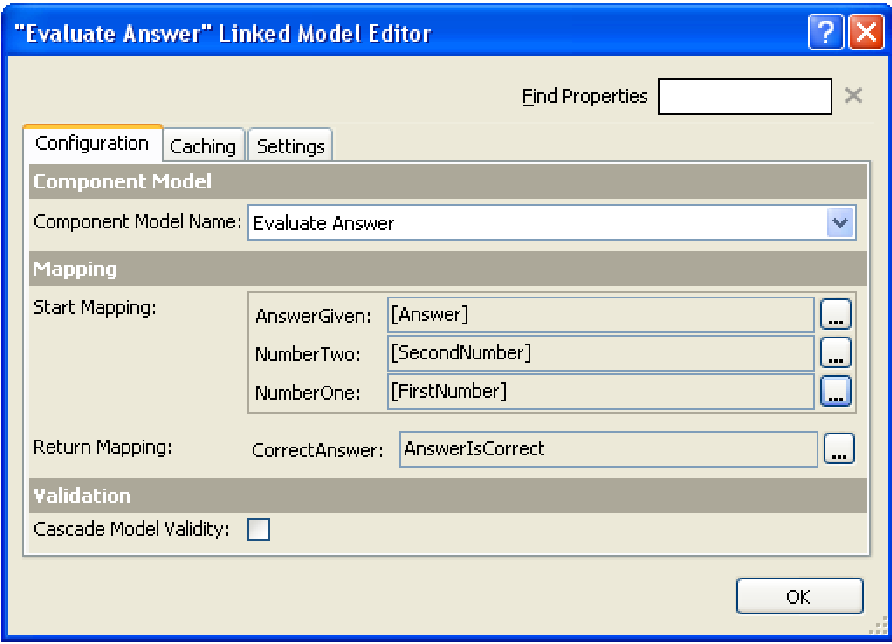


Remember that the variable we are mapping to (**`AnswerGiven`**) is of type number (integer). When we click the ellipse button we are able to see only those parent model variables of compatible types. In this case, we choose the corresponding variable **`Answer`**.





Here we see the input variable **AnswerGiven** is mapped into the process variable **Answer.** After repeating the mapping process for each of the input and output variables that we configured for the child model, the end result looks like this:



Here we see the input and output data variables on the left, and the original process variables on the right.

In review, what we've done above is mapped several variables from the parent model to the corresponding variables for the child model. The child model's variables were setup in its Input Data and Output Data screens.

Linked Model process

After the Linked Model's input and output data has been added, configured, and mapped, the components inside the Linked Model can perform their functions.

The Linked Model process starts with a Multiply Values component. This component takes in two variables (**NumberOne** and **NumberTwo**), multiplies them together, and produces an output variable (**Product**).

Next, an Equals Rule component compares the computer-generated answer (**Product**) with the multiplication problem answer provided by the user (**AnswerGiven**).

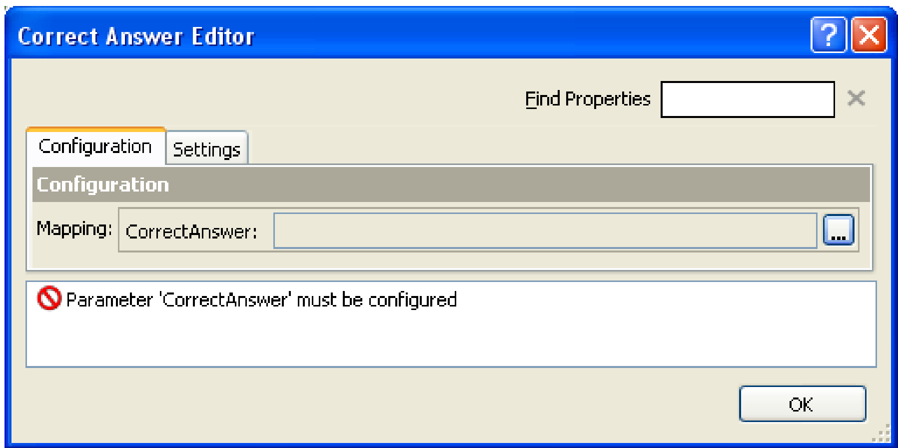
For more information on the Equals Rule component, see the Workflow Solution Component Examples Guide.

www.altiris.com/support/documentation.aspx

Finally, two End components complete the Linked Model process. The Evaluate Answer model needs to return the True/False variable **CorrectAnswer** which tells the parent model if the user's answer was correct (True=correct, False=incorrect). The Linked Model is configured with one output variable called **CorrectAnswer**. This variable's value is set in one of the two End components, depending on the outcome path of the Equals Rule component.

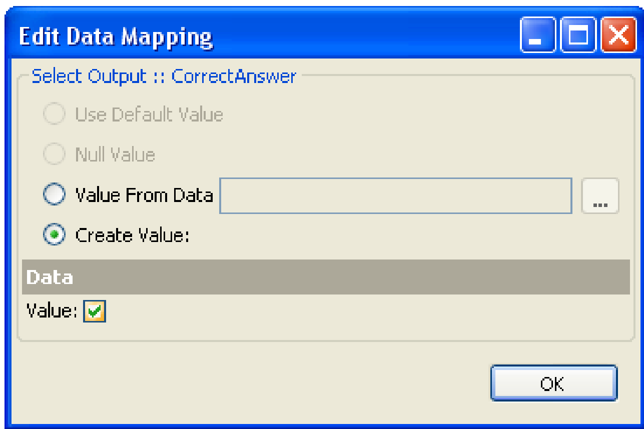
If the Equals Rule component finds that the variable **Product** and the variable **AnswerGiven** are equal, then it exits through the "Equals" path to an End component called "Right Answer." If **Product** and **AnswerGiven** are not equal, then the component exits through the "Not Equals" path to an End component called "Wrong Answer." The End component named "Right Answer" sets the variable **CorrectAnswer** to True; the End component named "Wrong Answer" sets the variable **CorrectAnswer** to False.

Let's go into the "Right Answer" End component:



Here we see the variable **CorrectAnswer** waiting to be mapped. However, it will not be mapped into another variable, but be assigned a constant value.

After clicking the ellipse, we choose "Create Value." Why "Create Value" instead of "Value From Data"? In this case, we want each of our End components to provide only a single outcome. "Right Answer" provides the value True, "Wrong Answer" provides the value False. We want no other possible outcomes. Therefore each End component is configured to provide only one constant value.



After choosing "Create Value," a check box labeled "Value" appears. If we check this box we will set the value of the variable "CorrectAnswer" to true, if we leave it unchecked we set the value to false.

Because we are configuring the End component "Right Answer" we want to check it (setting it to true). When configuring "Wrong Answer," we want to leave it unchecked (setting it to false).

When the Linked Model completes its evaluation of the student's answer and returns the variable **`CorrectAnswer`** (which is mapped into **`AnswerIsCorrect`**), the primary model can finish its work.

Back in the primary model, the process evaluates the variable **`AnswerIsCorrect`**. A True False Rule component performs the evaluation. If it finds the variable is set to true, the True False Rule component exits out the "True" path. If this is the case, the "Correct Answer" form opens, which congratulates the user on the correct answer and lets the user return for more questions or end the program.

If **`AnswerIsCorrect`** is false, the component exits through the "False" path and opens the form "Incorrect - Please Try Again." This form tells the user that he or she answered incorrectly, and asks them to retry the problem. The process continues until the user supplies a correct answer, or closes the browser.

Extra notes

The most difficult part about the Linked Model component is correctly configuring the input and output variables. It can be very confusing! To reduce confusion, use the same name for corresponding variables within the parent model (primary model) and child model (Linked Model).

The example presented here does not use the same names for variables in an effort to elucidate the mapping function. The chart below shows the corresponding variable names we used in our example:

Primary Model	Linked Model
FirstNumber	NumberOne
SecondNumber	NumberTwo

Primary Model	Linked Model
Answer	AnswerGiven
AnswerIsCorrect	CorrectAnswer

Each variable name represents a unique variable. However, because the variables are being mapped into each other, it is acceptable to use identical names and think of matching variables as one. For our illustration, it was important that we use different variable names to help the reader distinguish one set of variables from the other.

Chapter 4
Embedded model

About the Embedded Model component

Embedded Model

This component contains its own model. Think of the Embedded Model component as an isolated chunk of business logic that has a data contract with its parent model. You can use embedded models in your primary model, or even other embedded models. Using Embedded Models lets you break down larger processes into smaller, distinct sub-processes. This helps maintain organization and makes the main workflow more readable.

You can drag and drop an Embedded Model component from the component toolbox as you would any other component, and use it immediately in your process.

The concepts of a "parent model" and "child model" are very important when working with Linked Models.

See *Parent and child models* (page 6)

Embedded Model setup

When using the Embedded model component there are two phases of setup:

1. Configure the Embedded Model component's output variables.

   See *Configuring output data* (page 32)

2. Creating the Embedded Model process.

   See *About the Embedded Model process* (page 36)

Configuring output data

No input data needs to be added to Embedded Models because they can see all process data. However, Embedded Models do not automatically make their data available back to their parent models. Any data from the Embedded Model that the parent model needs to use must be configured as output data.

Adding data to output data

If an Embedded Model needs to pass data back to the parent model, it must be configured with output data. All data that needs to come out of an Embedded Model must be added to output data.

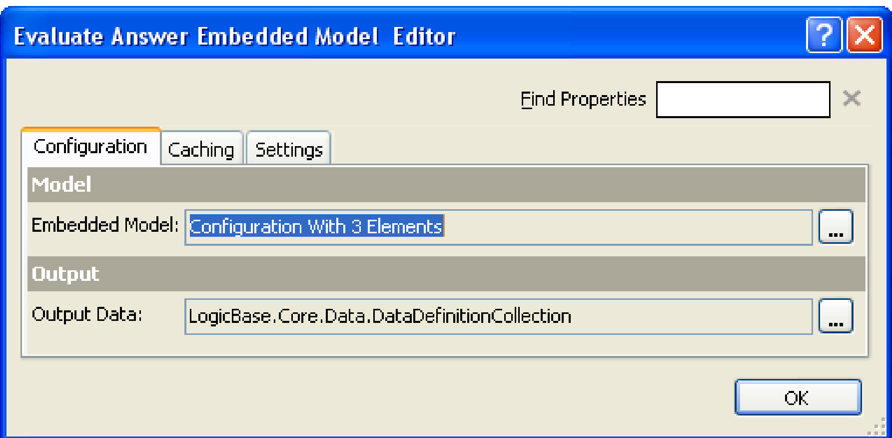
There are some scenarios in which output data is not necessary. For example, if the child model writes data off to a database instead of handing it back to the process, there is no need to use output data.

To determine what variables to add to output data, consider this question: "What variables exist inside this Embedded Model that the parent model needs to know about in order to accomplish its work?" (For example, if the Embedded Model compares two variables and renders an outcome to be used in the parent model, that parent model must have access to the outcome data.) Once you have answered this question, you are ready to add data to the Embedded Model.

To add output data

1. In the parent model of your process, right-click the **Embedded Model component** and click **Edit Component**.

   You will see a dialog box like the one below:

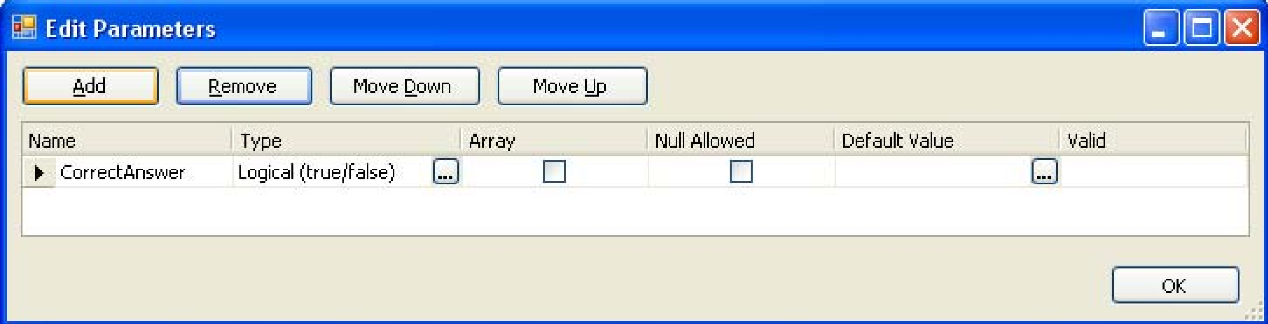


2. Click the **[...]** button next to the Output Data field.
3. Click **Add.**

   Create one or more variables to match variables in the Embedded Model. If you have not yet created the process inside the Embedded Model, you may not know what variables you need to output. In this case, make the process first, then return to this step at the end.

   See *About the Embedded Model process* (page 36)
4. Configure each variable to match its corresponding variable.

   Each variable being created as output data is a piece of data that must be passed back to the parent model. For example, if your Embedded Model contains a logical (true/false) data element named **`CorrectAnswer`** that the parent model needs to see, create an output variable of the same name and type as seen below:



5. Click **OK.**

NOTE
If you want to output data to a variable that already exists in the parent model, you "recreate" it as output from the Embedded Model component. This overwrites the value that previously existed for that variable.

Mapping output data

Data mapping refers to a transfer of value, where one variable is pointed at another variable which takes the value of the first. For example, if a variable called **Variable1** with a value of "person" is mapped into a variable called **Variable2**, "person" becomes the value of **Variable2.**

You need to map data in the End component(s) of the Embedded Model only if you have configured output data. If the Embedded Model does not need to output any of its data, you do not need to configure its End components.

See *Adding data to output data* (page 32)

After you have added output data to the Embedded Model, the End components inside the model gain a data mapping capability. Any output data from the Embedded Model must be mapped to existing model variables.

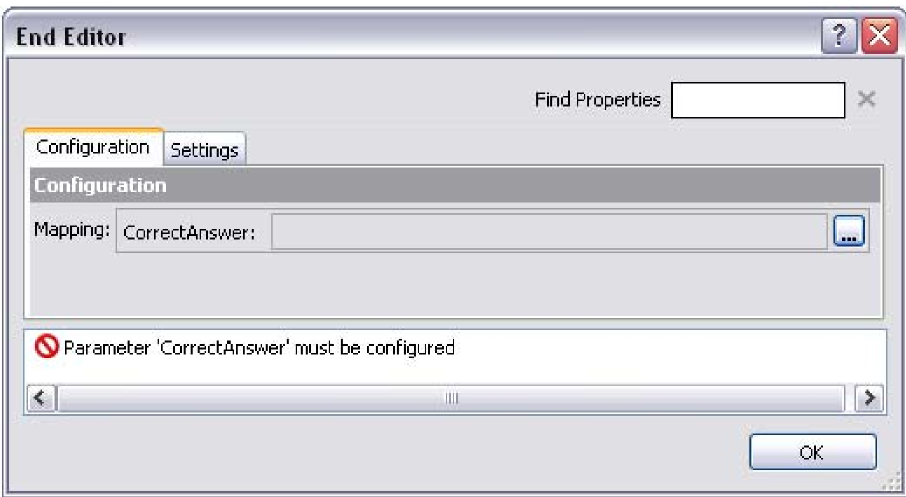
To map data in the Embedded Model End components

1. Open the Embedded Model.

   To do this, double-click the **Embedded Model** component.

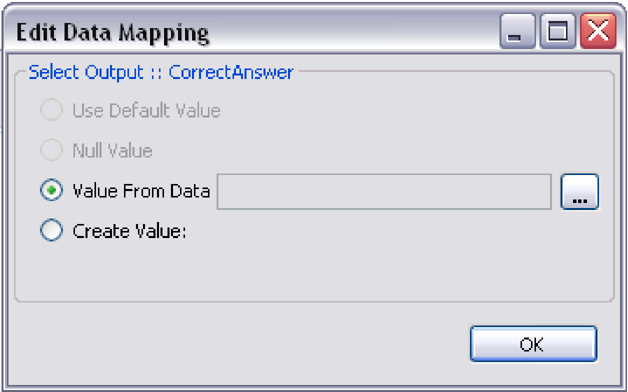
2. Open the End component.

   To do this, double-click the **End** component.



Here we see an example of an End component with only one available variable (**`CorrectAnswer`**). It is not yet mapped.

3. Select **Value From Data** and click the **[...]** button.



Below are descriptions of the other options.

Use Default Value	Uses the default value of the variable in question (in this case **`CorrectAnswer`**). This is an option only if you set a default value on an output variable.
Null Value	Designates a null value of the variable in question (in this case **`CorrectAnswer`**). This is an option only if you set an output variable to allow a null value.
Value From Data	Lets you pick a variable from the Embedded Model to map into the variable in question (in this case **`CorrectAnswer`**). This is the most commonly used option. Use this option to map the value of a variable in the Embedded Model into an output variable.
Create Value	Assigns a constant value to the variable in question (in this case **`CorrectAnswer`**).

4. Locate and select the variable from which you want to map data, and click **OK.**

About the Embedded Model process

An embedded model can make use of nearly all components available within Workflow Designer, but there are a few notable exceptions. Linked Models, Form Builders, and Workflow components (for example, Dialog Workflow) cannot be used in embedded models. Keep these limitations in mind as you build the model.

Build the Embedded Model exactly as you do the primary model. Embedded Models require no special component configuration. One exception to this is the End components. If you want to make data from the Embedded Model available to the outside process, you must map that data out of the End component.

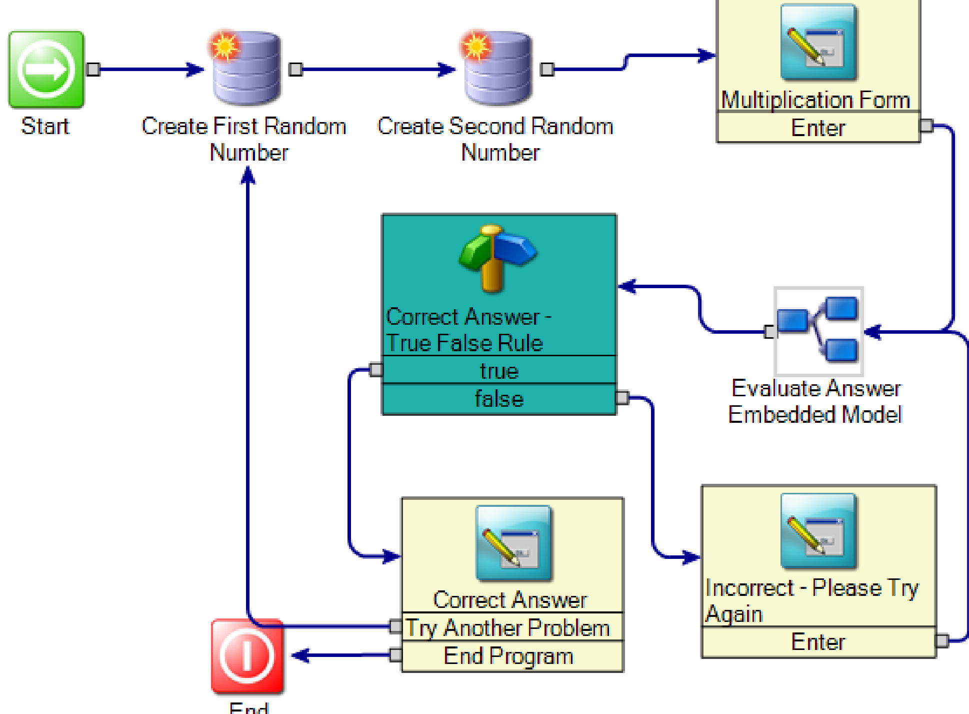
See *Configuring output data* (page 32)

In some scenarios (such as an Embedded Model that performs a true/false decision), using two End components is preferable to using one.

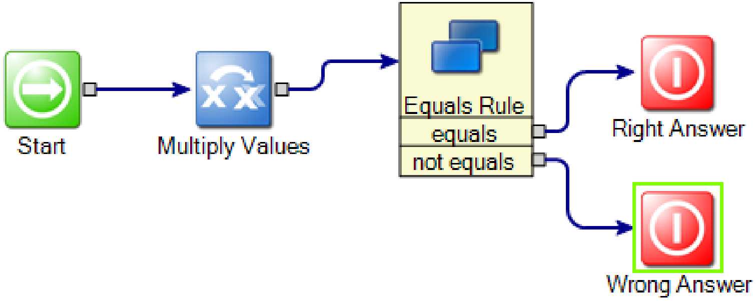
Example use of the Embedded Model component

Scenario: A flash card application helps elementary students learn multiplication tables. Students are shown two random numbers and are asked to multiply them and provide the answer. Each time a student clicks the "Enter" button, the application must determine whether the student's answer was correct. If it was not, the student is given another chance to answer.

Process: This process is built in a Webforms-type project. The process generates two random numbers between 0 and 10, displays the numbers in a Webform, and asks the student to multiply the numbers and provide the correct answer. Using a separate model (the Embedded Model), the process evaluates the answer and sets the value of a variable called **`CorrectAnswer`** to either true or false. The two models used in this workflow are called "Primary Model" and "Evaluate Answer." Here's what the primary model looks like:



Here's what the Embedded Model looks like:



Primary model (parent model)

Components Used in this Model:

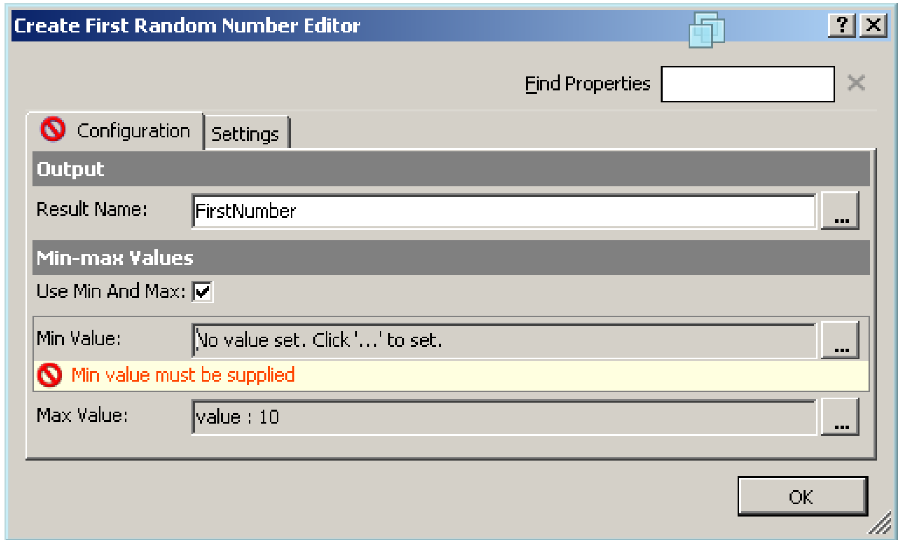
- Create Random Number
- Add New Data Element
- Form Builder
- Embedded Model
- True False Rule

For more information on these components, see the Workflow Solution Component Example guide.

www.altiris.com/support/documentation.aspx

The process begins with two Create Random Number components. These components create the numbers that the student is asked to multiply.

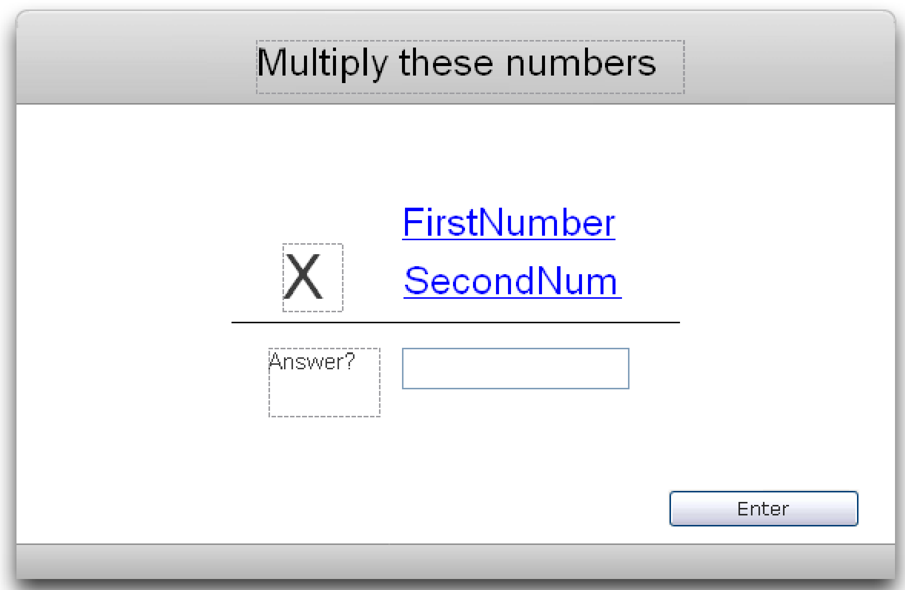
Let's go into the first Create Random Number rule:



Both Create Random Number components are configured the same way (except for the "Result Name," which must be unique). We've checked the "Use Min and Max" feature because we want to limit the value of the numbers to be between 0 and 10. The "Min Value" by default is null. We will assign 0 as a constant value for this field.

Next, a Form Builder component presents the multiplication problem to the student. This process uses three From Builders throughout this process. They are named "Multiplication Form", "Correct Answer" and "Incorrect - Please Try Again" to describe what their purpose is in the workflow.

Here is the first form at design-time:



The textbox takes in the student’s answer and outputs it as a variable called **AnswerGiven**.

Embedded Model (“Evaluate Answer”)

Components Used in this Model:

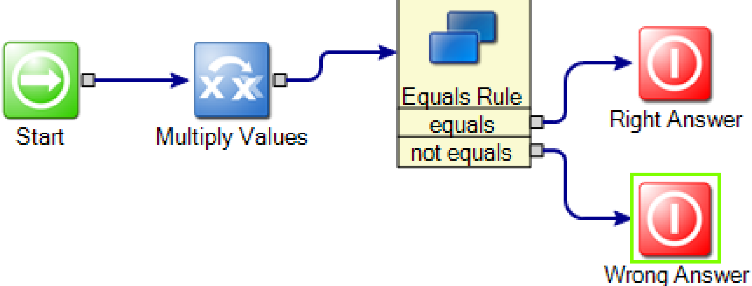
- Multiply Values
- Equals Rule

For more information on these components, see the Workflow Solution Component Example guide.

www.altiris.com/support/documentation.aspx

In this scenario the parent model is "Primary Model" and the child model is the Linked Model "Evaluate Answer." In a real world scenario, a linked model will likely carry out a more complex workflow. However, as this is an example project only, our Embedded Model is very small and is carrying out a relatively simple task.

Let’s take another look at the Embedded Model:



This model calculates the correct answer and compares it with the user-supplied answer.

Embedded Model data

This Embedded Model calculates the correct answer and compares it with the user-supplied answer. Because Embedded Models can see data outside of themselves, no data needs to be added as input data. The Embedded Model needs to pass the result of its calculation back to the primary model, so it needs one piece of output data.

Output data

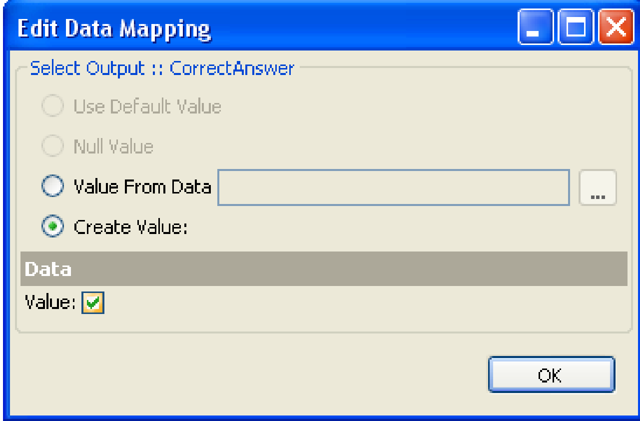
In our scenario, the child model outputs a variable called **CorrectAnswer** of type "Logical (true/false)" (True = the user's answer was correct, False = the user's answer was incorrect). This will make more sense when we see the function of the End components.

Output data is declared in the Embedded Model component in the parent model. Here’s what it looks like:

In this case, only one logical (true/false) variable is declared. This variable corresponds to the variable **AnswerIsCorrect**. **CorrectAnswer** tells the parent model whether the user's answer was correct or not.

Data mapping

Data from the Embedded Model is not mapped back into parent model in this case. However, data from the Embedded Model still needs to be configured in the data mapping editor. In this case, the Embedded Model uses two End components, that each set **CorrectAnswer** to a constant value. Let's go into the "Right Answer" End component:



Here we see that this End component sets **CorrectAnswer** to a constant value of "true." The End component "Wrong Answer" is configured like this, except with a false value.

This configuration yields the following function: if the Embedded Model exits through the "Wrong Answer" End component, **CorrectAnswer** is set to true; if the Embedded Model exits through the "Right Answer" End component, **CorrectAnswer** is set to false.

Whether True or False, **CorrectAnswer** is exposed to the parent model through the Embedded Model's output data.

See *Output data* (page 39)

Embedded Model process

The Embedded Model process starts with a Multiply Values component. This component takes in two variables (**NumberOne** and **NumberTwo**), multiplies them together, and produces an output variable (**Product**).

Next, an Equals Rule component compares the computer-generated answer (**Product**) with the multiplication problem answer provided by the user (**AnswerGiven**).

Finally, two End components complete the Embedded Model process. The Evaluate Answer model needs to return the True/False variable **CorrectAnswer** which tells the parent model if the user's answer was correct (True=correct, False=incorrect). The Embedded Model is configured with one output variable called **CorrectAnswer**. This variable's value is set in both of the End components. The final value of this variable depends on which End component is invoked.

If the Equals Rule component finds that the variable **Product** and the variable **AnswerGiven** are equal, then it exits through the "Equals" path to the End component called "Right Answer." If **Product** and **AnswerGiven** are not equal, then the component exits through the "Not Equals" path to the End component called "Wrong Answer." The End component named "Right Answer" sets the variable **CorrectAnswer** to True; the End component named "Wrong Answer" sets the variable **CorrectAnswer** to False.

Back in the primary model (parent model), the process evaluates the value of **CorrectAnswer.**

A True/False Rule component makes the evaluation. If it finds **CorrectAnswer** has a true value, then the component will exit through the "True" path. If **CorrectAnswer** has a false value, the component exits through the "False" path. Depending on the result, the user sees one of two forms - a "congratulations" form, or an "incorrect, please try again" form. The latter form lets the user choose to return to the problem and try again.

Chapter 5
Other model components

The Linked and Embedded Model components each have a derived component that has a slight variation in function.

There are two derived model components:

- *Dynamic Linked Model component* (page 42)
- *Embedded Rule Model component* (page 52)

Dynamic Linked Model component

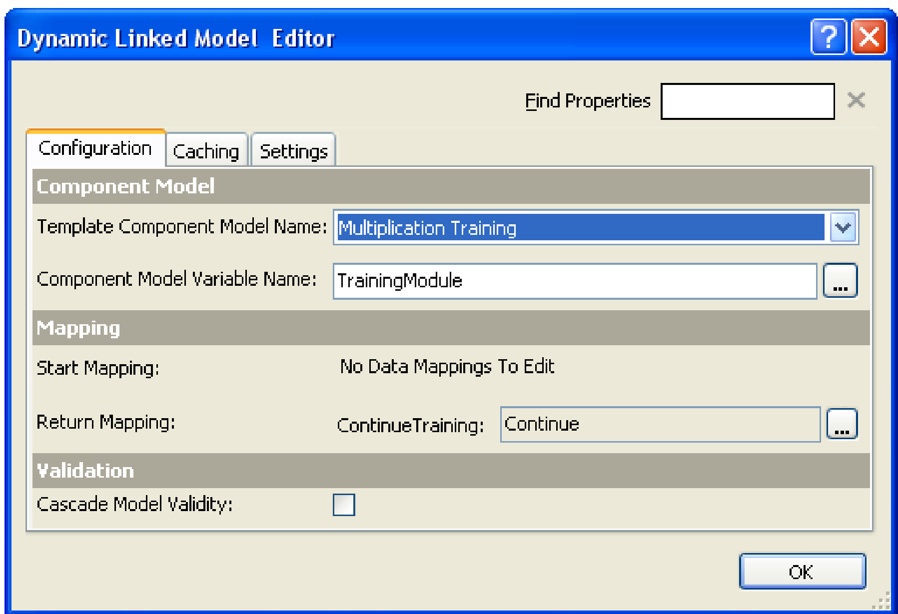
The Dynamic Linked Model component takes the concept of linked models one step further than basic Linked Model components. Whereas the basic Linked Model component represents only one secondary model (the model in the project tree structure to which it corresponds), the Dynamic Linked Model can represent any secondary model in the project tree structure. It uses a process variable to dynamically select which secondary model it represents, rather than a constant setting. Thus, the Dynamic Linked Model can choose a secondary model programmatically, adding a great deal of flexibility to your process design.

About the Dynamic Linked Model Component

The majority of the process for setting up the Dynamic Linked Model component is identical to that of the basic Linked Model component. However, the Dynamic Linked Model component has two properties that the basic Linked Model component does not have: Template Component Model Name and Component Model Variable Name.

About the template component model

Below is an example of a configured Dynamic Linked Model editor:



Here we see the Dynamic Linked Model's two unique properties in the top of the editor: Template Component Model Name and Component Model Variable Name. The first of these, Template Component Model Name, refers to a "template" model from which the Dynamic Linked Model borrows data mapping definitions. A Dynamic Linked Model must use a template model because it can map only one set of Start and Return Mapping variables.

This is the most difficult concept in the Dynamic Linked Model component. The Dynamic Linked Model requires a template model for data mapping because data mapping cannot be defined dynamically. Dynamic Linked Models do not support different mapping configurations for different secondary models. During design-time, a Dynamic Linked Model does not know which secondary model it represents; thus, it also does not know what input and output data it should have, or how those values should be mapped.

This means that every possible secondary model that a Dynamic Linked Model may call must have identical input and output variables. Instead of requiring you to exactly duplicate the configuration of the secondary models it may call, the Dynamic Linked Model lets you select one as a template. The name of the model selected as the template becomes the Template Component Model Name. Once set, the template model tells the component which input and output variables to look for, and how to map them.

About the component model variable name

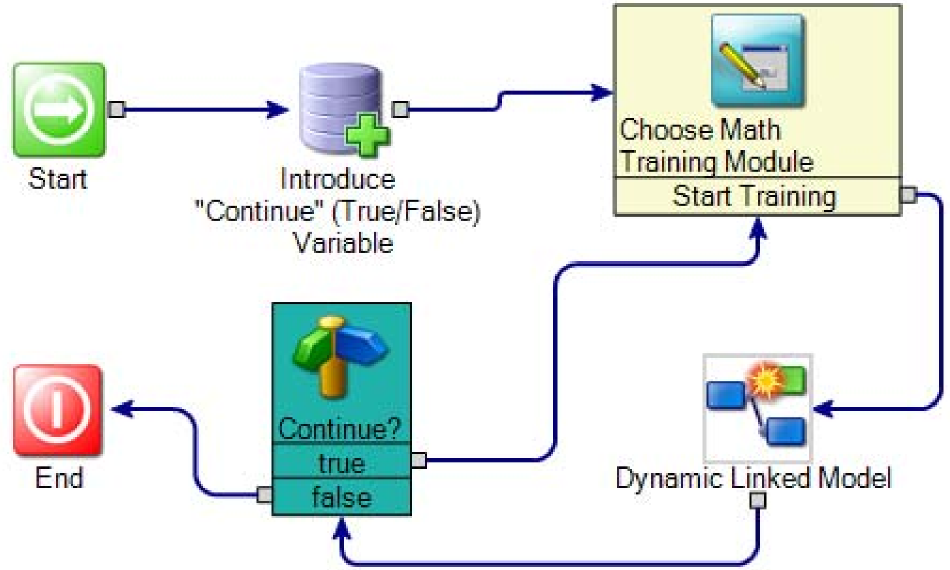
The second property unique to the Dynamic Linked Model component, Component Model Variable Name, tells the component which secondary model it represents. In the screen shot above, the Dynamic Linked Model uses the variable **`TrainingModule`.** It is very important to note that whatever model name is passed to this variable must correspond exactly to one of the model names listed in the project model tree.

Once you provide a text variable in the Component Model Variable Name field, you can set the value of this variable programmatically. Thus, you can make the secondary model selection dynamically.

Example use of Dynamic Linked Model component

Scenario: An elementary student uses a flash card application to learn various mathematical functions. The student chooses which function she wants to study (addition, subtraction, multiplication or division), is shown two random numbers in a math problem, and is asked to answer. Each time the student clicks "Enter," the application determines whether the student's answer was correct. If it is not, the student is given another chance to answer. In this scenario the parent model is Primary Model and the secondary model is either Division Training, Subtraction Training, Multiplication Training or Addition Training.

Process: This process is built in as a Webforms-type project. Here's what it looks like:



After the desired mathematical function is chosen, the process launches the corresponding secondary model and generates two random numbers between 0 and 10. It displays these numbers to the student and asks them to provide the correct answer. The process evaluates the answer and then gives the student different choices based on the result.

This project was developed from the example used for the Linked Model component and each of the secondary models is a derivative of that example. For more information on setup and configuration, see the Linked Model chapter.

See *Linked model* (page 11)

Primary Model (parent model)

Components Used in this Model

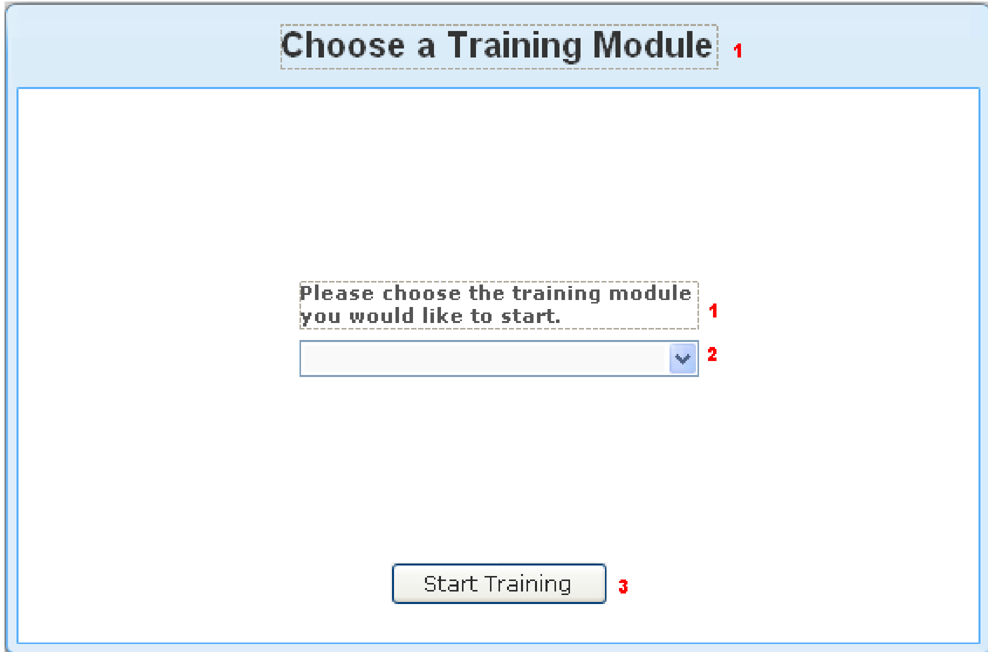
- Add New Data Element
- Form Builder
- Dynamic Linked Model
- True False Rule

For more information on these components, see the Workflow Solution Component Example guide.

www.altiris.com/support/documentation.aspx

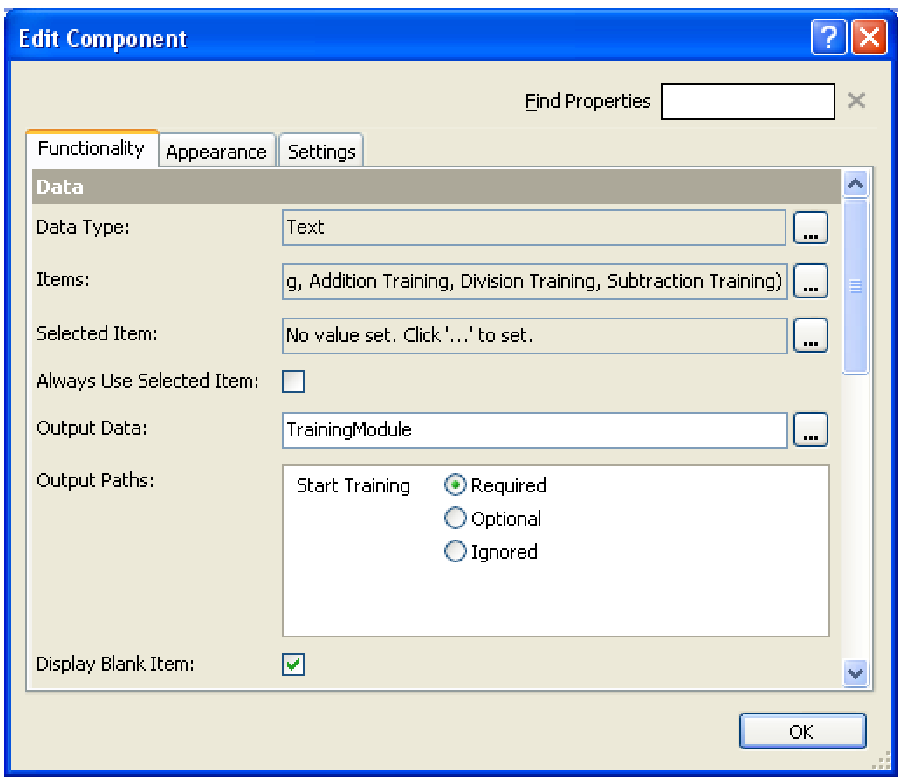
An Add New Data Element component starts the process by creating a text variable called **Continue**. The student is then presented a form with a list of mathematical functions. The Dynamic Linked Model component then launches the appropriate secondary model process based on the user's selection. It also passes back a value for the variable "Continue" when the user is done. If the student wants to continue, he or she can choose a new mathematical function on which to train.

Let's take a look at the Form Builder component:



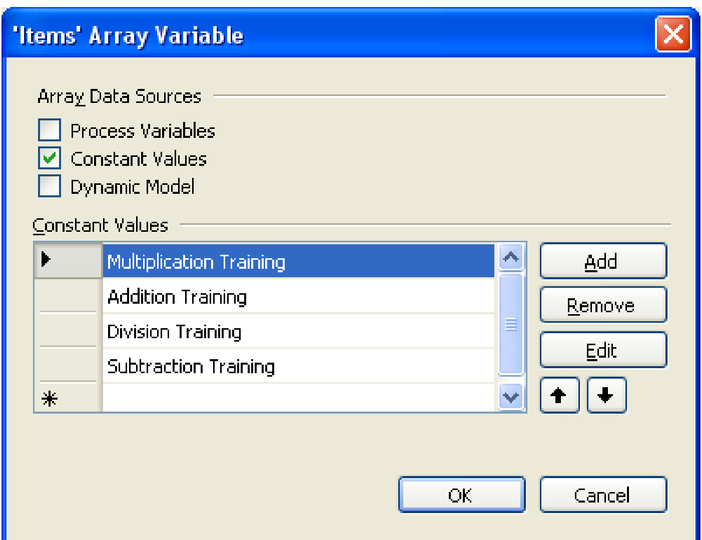
A drop down list component (#2) contains a list of math functions (addition, subtraction, multiplication, division), and produces a variable called **TrainingModule** that contains the student's choice.

Let's go into the drop-down list component:

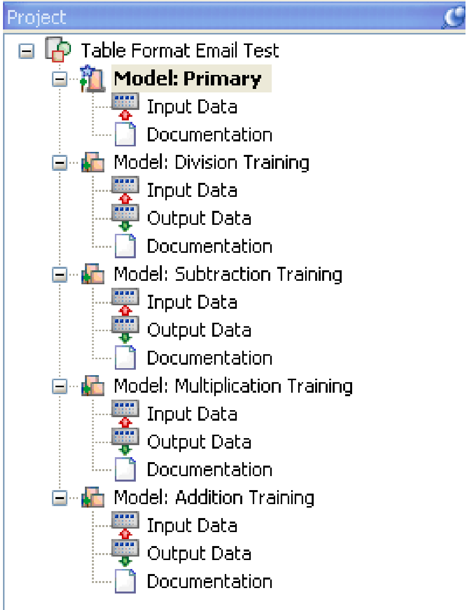


In this screenshot, of primary concern to us are the Items field and the Output Data field. In the screenshot above you can see that in the Output Data field, we've assigned the variable **`TrainingModule`** to hold output from this component. In the Items field we've configured a list of constant values that correspond exactly to the names of the secondary models we want the user to choose from.

If we click the ellipse button for the items, we see the list of constant values:



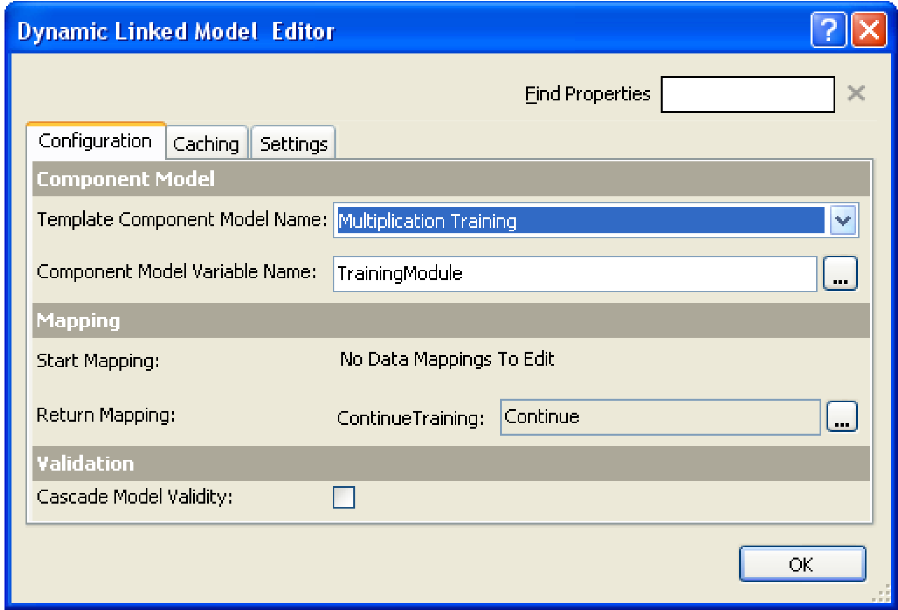
These item names directly correspond to the names of the math models in the process. You can see in the screenshot below that the list of model names we just configured are taken directly from the model names in the project model tree.



Dynamic Linked Model component

Although we've seen in detail how the Dynamic Linked Model component is setup, let's review by looking at the configurations for this project.

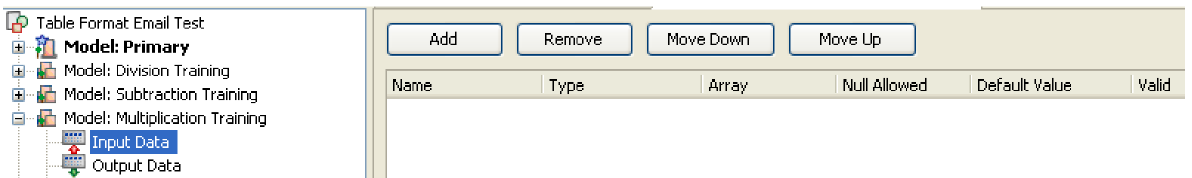
Here's what the editor looks like:



First, notice the variable in the Component Model Variable Name field. This is the output variable from the drop-down list in the form, and holds the name of the secondary model we want to launch.

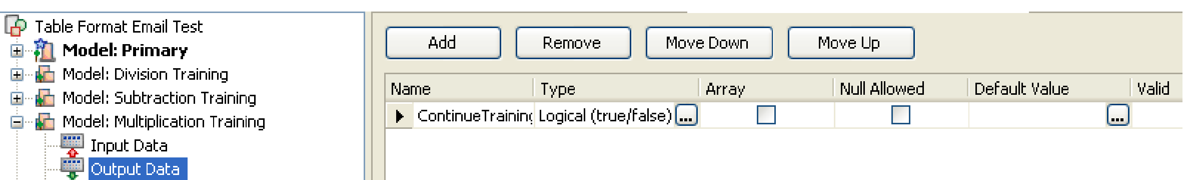
Next, notice that we have selected Multiplication Training as the Template Component Model Name. Again, the Template Component Model Name is the model the component uses to determine what Start and Return Mapping variables need to be configured.

Let's look at the input and output variables for the Multiplication Training model and how that corresponds to what we see in the component:



Here we see that the Multiplication Training model has no Input Data configured. In other words, it is not expecting to receive any data from the parent model when it launches. With no input data, there is nothing to configure under the Start Mapping header of the Dynamic Linked Model component (previous screenshot).

Now let's look at the output data:



Here we see that this model is configured to return one piece of output data - a logical type piece of data called **ContinueTraining**. This output variable tells the parent model to expect a single piece of data of type Logical (True/False). This variable will remain empty unless it has a value mapped into it. Mapping is configured under the Return Mapping header in the Dynamic Linked Model component. **ContinueTraining** should be mapped into the **Continue** variable that we introduced at the very beginning of the Primary Model process.

NOTE
Remember that each input and output variable configured for a secondary model requires a corresponding variable in the parent model. The mapping process that we just reviewed tells the models how they relate to one another.

Finally, the End components for each of the four secondary models in this project need to be configured. This is done exactly as outlined for the Linked Model component.

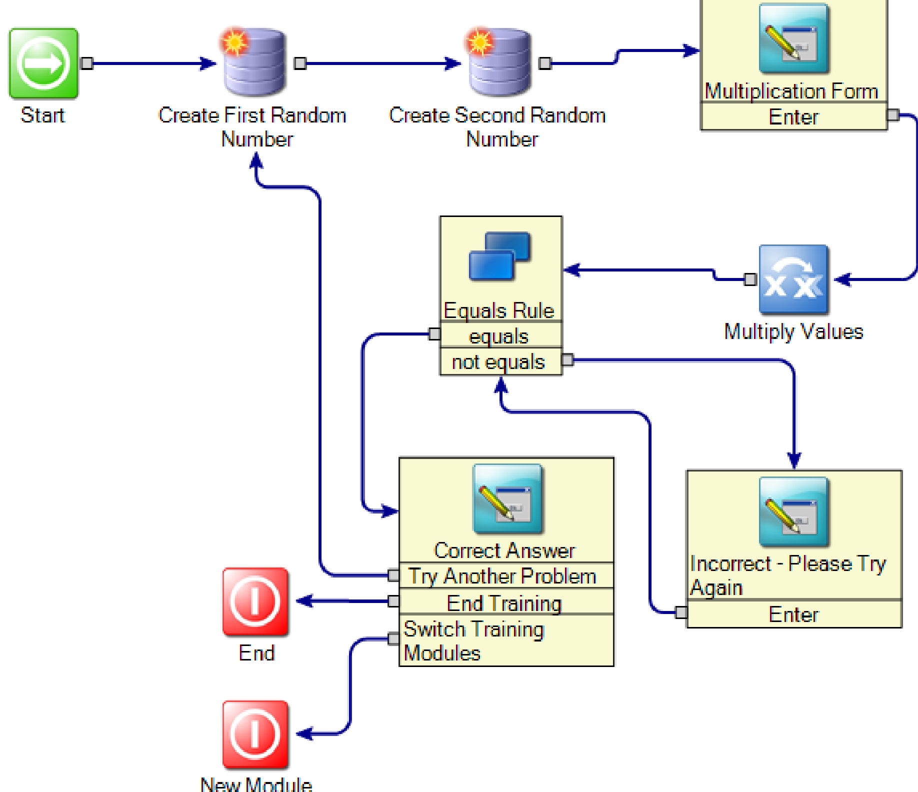
See *To map data from the Linked Model end component* (page 19)

Potential secondary models

Below is the overview for each of the secondary models that the Dynamic Linked Model could potentially call. We will not discuss them in detail as they are all derived from the example project for the Linked Model component.

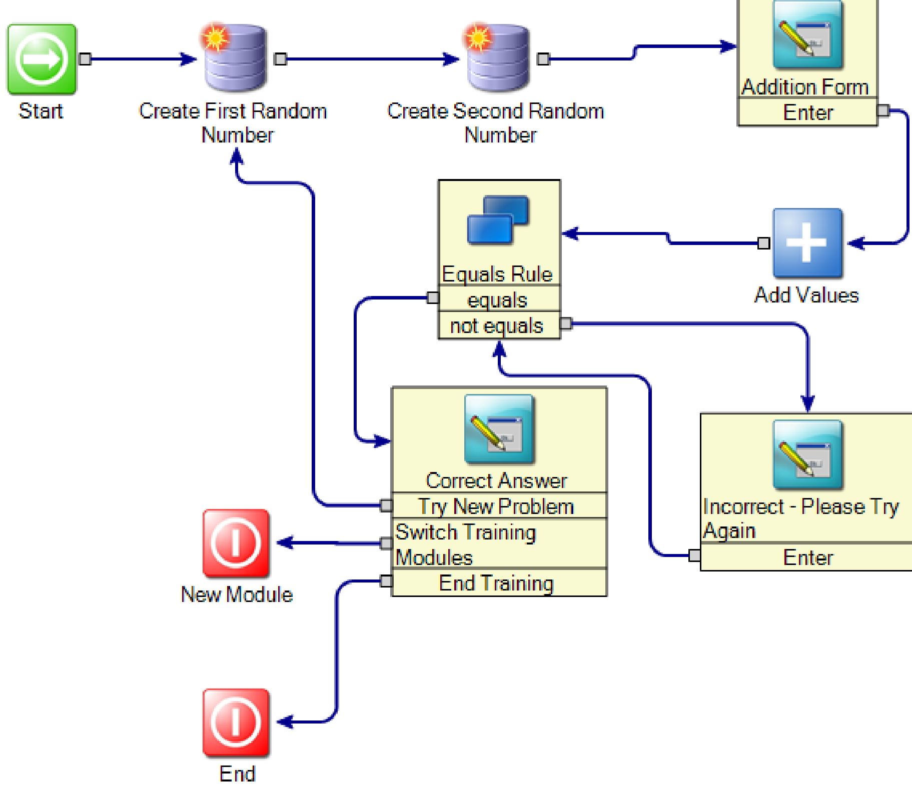
Multiplication Training

Here we see the Multiplication Training model. Out of the four possible training models, this model is invoked if the user chooses multiplication training in the first form. If this is the case, the Addition Training, Division Training, and Subtraction Training models do not run.



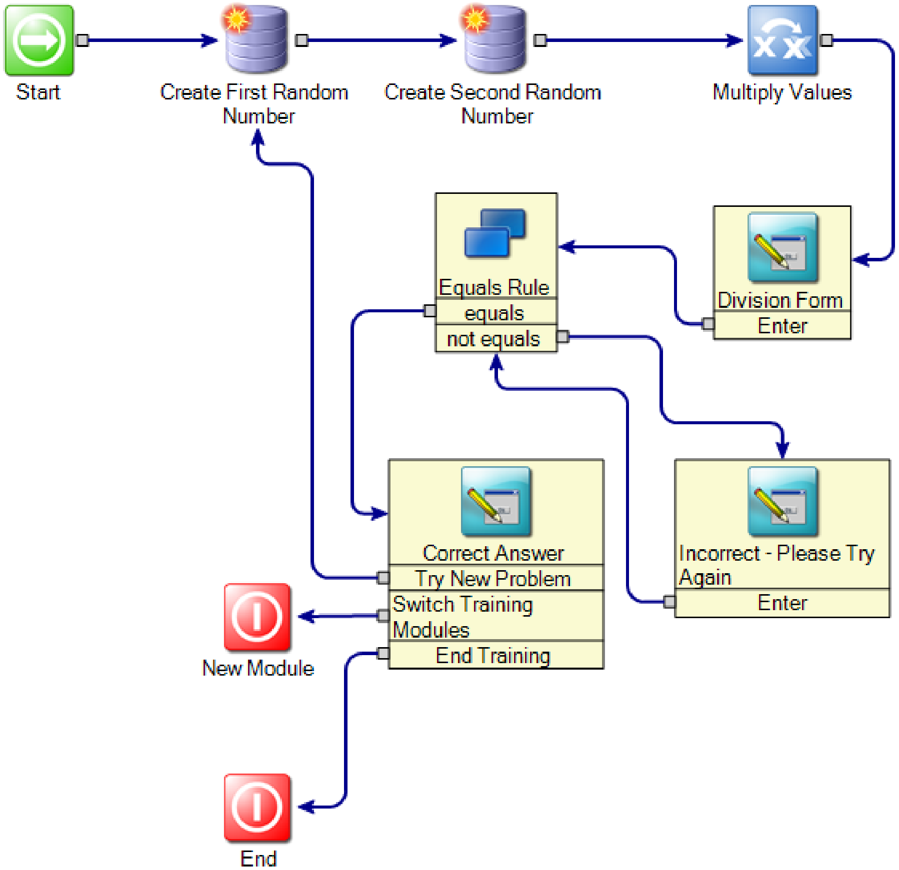
Addition Training

Here we see the Addition Training model. Out of the four possible training models, this model is invoked if the user chooses addition training in the first form. If this is the case, the Multiplication Training, Division Training, and Subtraction Training models do not run.



Division Training

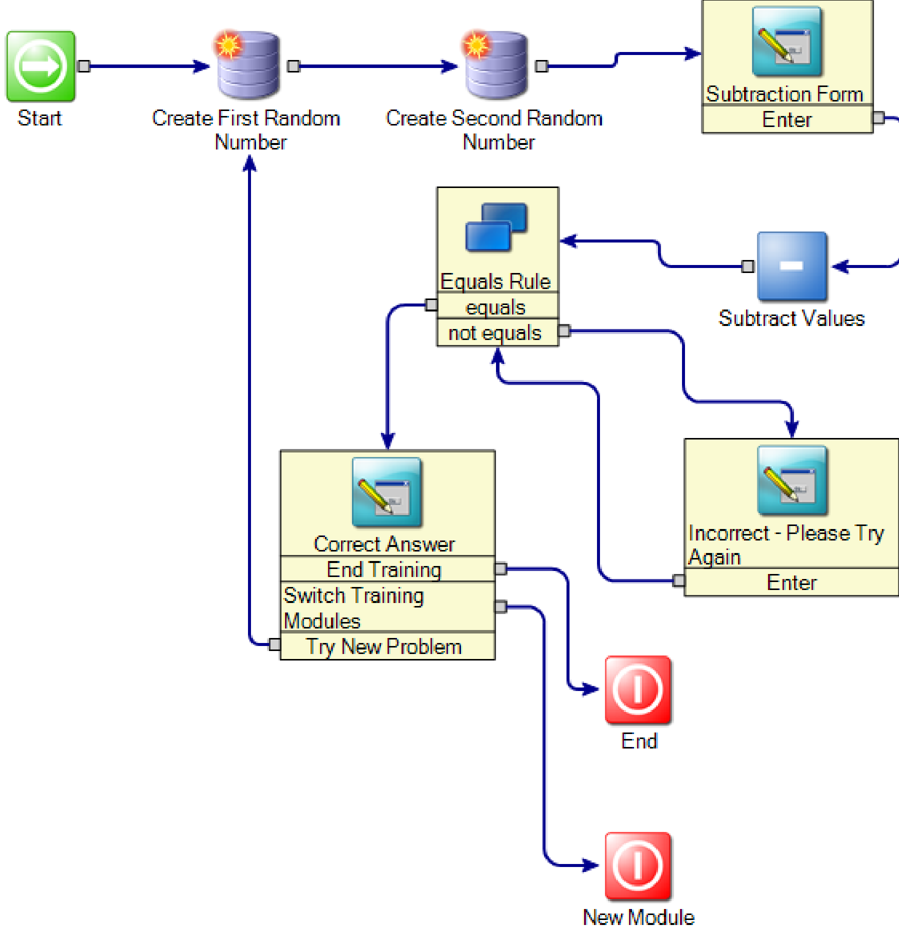
Here we see the Division Training model. Out of the four possible training models, this model is invoked if the user chooses division training in the first form. If this is the case, the Addition Training, Multiplication Training, and Subtraction Training models do not run.



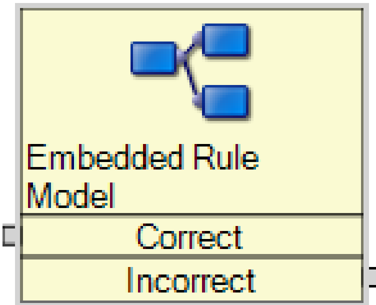
There are some differences in this model that are worth noting. First, we multiply the randomly generated numbers prior to the Division Form (which presents the mathematical problem for the user to solve) and then present a problem which divides the multiplied answer by the **SecondNumber** variable. This ensures that our division problems always divide evenly. Additionally, the Create Second Random Number component generates numbers from 1 to 10 instead of 0 to 10. Because **SecondNumber** is used as the divisor in out mathematics problem we can avoid dividing by zero.

Subtraction Training

Here we see the Subtraction Training model. Out of the four possible training models, this model is invoked if the user chooses subtraction training in the first form. If this is the case, the Addition Training, Division Training, and Multiplication Training models do not run.



Embedded Rule Model component



The Embedded Rule Model component takes the concept of embedded models one step further than basic Embedded Model components.This component operates exactly like an Embedded Model component, but with one additional feature: multiple outcome paths. This component is designed to be a custom rule component that functions according to the components and outcome paths that you add. You can add any components to an Embedded Rule Model that you can to a basic Embedded Model; you

are not restricted to only rule components. You can also add as many outcome paths as you want.

You can use Embedded Rule Models in your primary model or any sub-models. Using Embedded Rule Models, you can break down larger processes into smaller, distinct sub-processes. This helps maintain organization and generally makes the main workflow more "readable."

About the Embedded Rule Model component

For basic information on model components, see the introduction to this guide.

See *Introduction* (page 4)

When using the Embedded Rule Model component there are two distinct phases to the setup:

1. Build the Embedded Rule Model process.

   See *About the Embedded Rule Model process* (page 53)

2. Configure the End component(s) of the secondary model.

   See *Configuring the Embedded Rule Model's End Components* (page 53)

About the Embedded Rule Model process

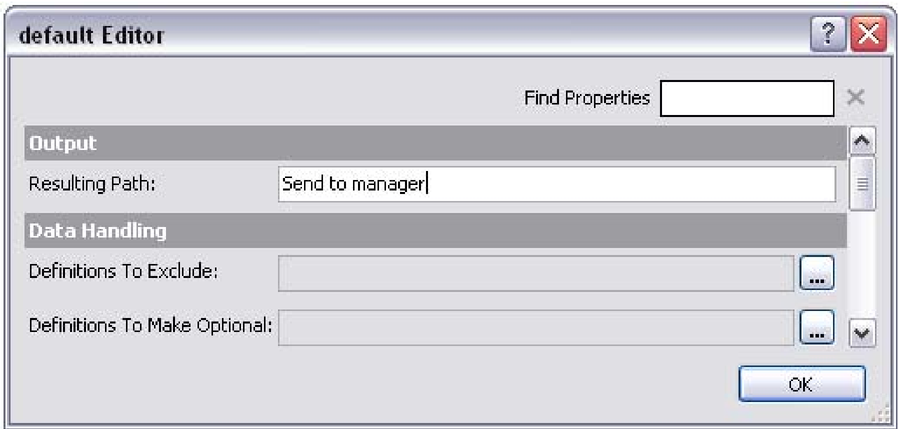
An Embedded Rule Model can make use of nearly all components available within Workflow Designer, but there are a few notable exceptions. Linked Models, Form Builders, and Workflow components (for example, Dialog Workflow) cannot be used in Embedded Rule Models.

Build the Embedded Model exactly as you do the primary model. Embedded Models require no special component configuration. One exception to this is the End component. Each End component must be configured to represent an outcome path. If you do not need more than one outcome path, use a regular Embedded Model component rather than an Embedded Rule Model component.

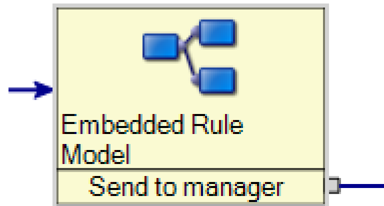
Configuring the Embedded Rule Model's End Components

The End components in an Embedded Rule Model give the component its multiple paths. Each End component contains a property called "Resulting Path." This value is the outcome path to which an individual End component points. For example, if an End component has a Resulting Path of "Send to Manager," that End component points to an outcome path called "Send to Manager." When an End component is added to the Embedded Rule Model process, a matching output path is automatically generated.

For example, here is an End component configured with a resulting path of "Send to manager":



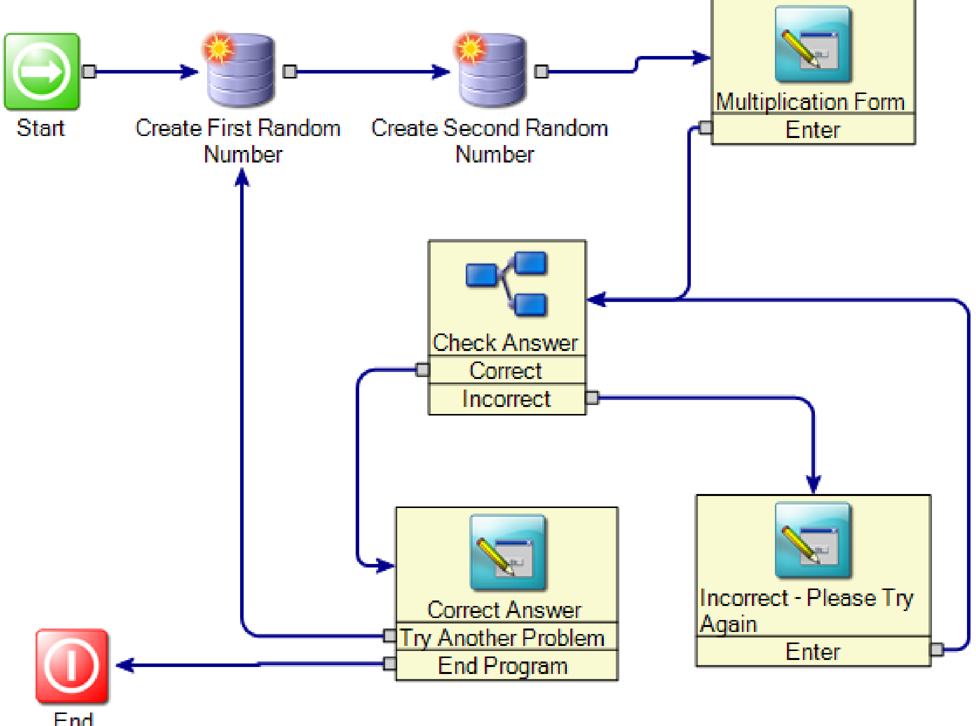
This End component produces a new outcome path on the Embedded Rule Model with the same name - “Send to manager”:



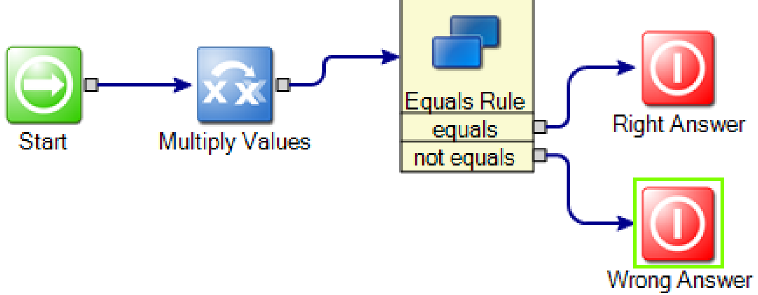
Example use of the Embedded Rule Model component

Scenario: A flash card application helps elementary students learn multiplication tables. Students are shown two random numbers and are asked to multiply them and provide the answer. Each time a student clicks the “Enter” button, the application determines whether the student's answer was correct. If it was not, the student is given another chance to answer.

Process: This process is built in a Webforms-type project. The process generates two random numbers between 0 and 10, displays the numbers to a student and asks the student to multiply the numbers and provide the correct answer. Using a separate model (the Embedded Rule Model), the process evaluates the answer and sets the value of a variable called **`CorrectAnswer`** to either true or false. The two models used in this workflow will be called "Primary Model" and "Evaluate Answer." Here's what the primary model looks like:



Here's what the Embedded Model looks like:



Primary model (parent model)

Components Used in this Model

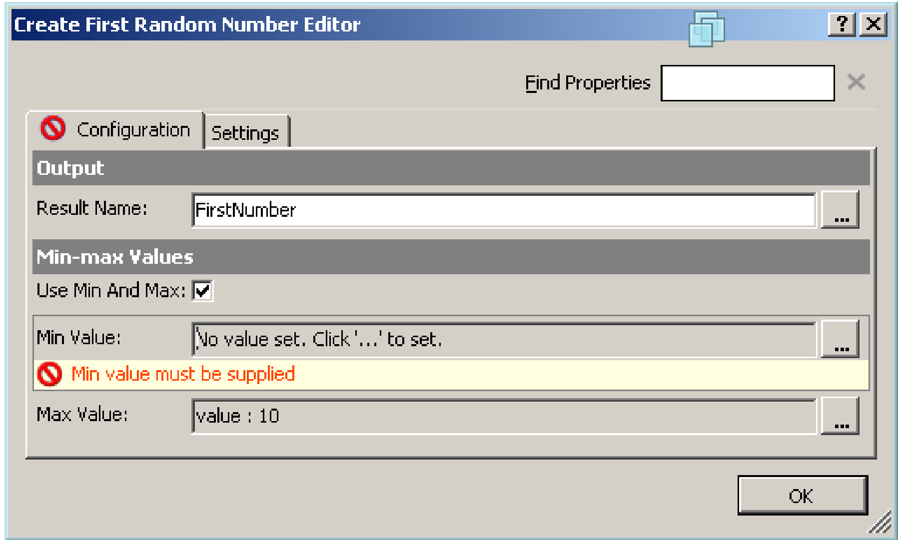
- Create Random Number
- Form Builder

- Embedded Rule Model

For more information on these components, see the Workflow Solution Component Example guide.

www.altiris.com/support/documentation.aspx

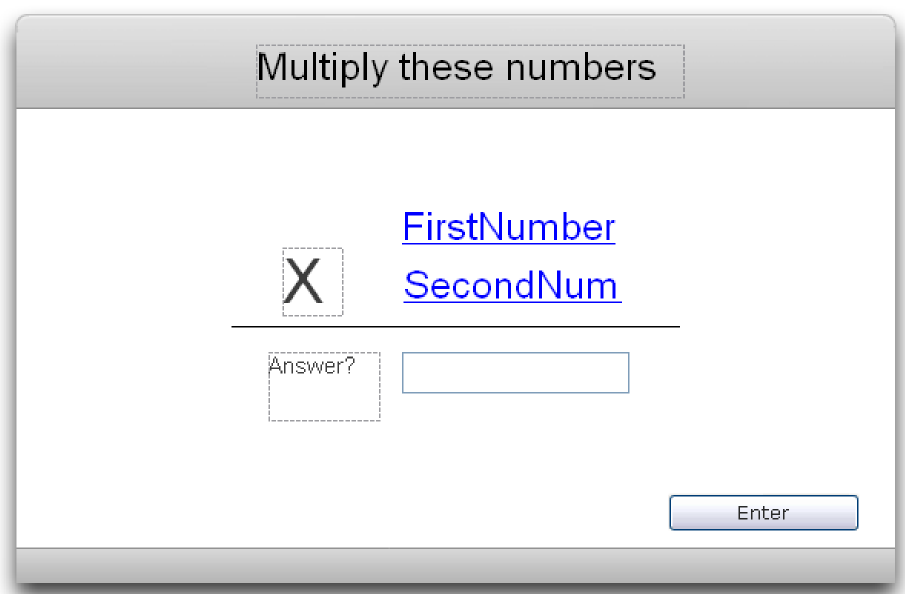
The process begins with two Create Random Number components. Here is what one of them looks like partially configured:



Both Create Random Number components are configured the same way (except for the "Result Name," which must be unique). We've checked the "Use Min and Max" feature because we want to limit the value of the numbers to be between 0 and 10. The "Min Value" by default is null. We will assign 0 as a constant value for this field.

Next, a Form Builder component presents the multiplication problem to the student. This process uses three From Builders throughout this process. They are named “Multiplication Form”, “Correct Answer” and “Incorrect - Please Try Again” to describe what their purpose is in the workflow.

Here is the first form at design-time:



The Textbox takes in the student's answer and outputs it as a variable called **`AnswerGiven`**.

Embedded Rule Model (child model)

Components Used in this Model:

- Multiply Values
- Equals Rule

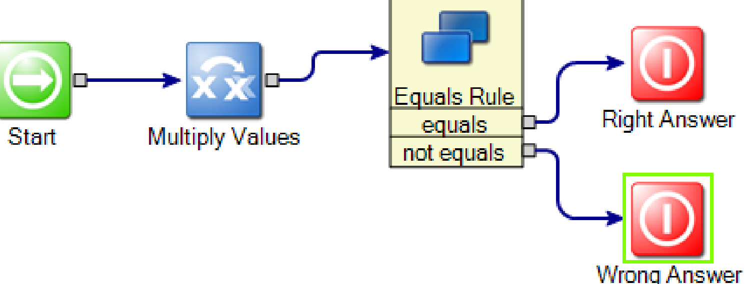
For more information on these components, see the Workflow Solution Component Example guide.

www.altiris.com/support/documentation.aspx

In a real world scenario, a Embedded Rule Model would likely carry out a much more complex process. Because this is an example project only, our Embedded Rule Model carries out a relatively simple task.

In this scenario the parent model is the Primary Model and the child model is the Embedded Rule Model. The Primary Model receives information from the user, and the Embedded Rule Model evaluates the information. Because this component does not ever require input or output data, the only necessary configuration is building the Embedded Rule Model process.

Let's take another look at the Embedded Rule Model:



This model calculates the correct answer and compares it with the user-supplied answer.

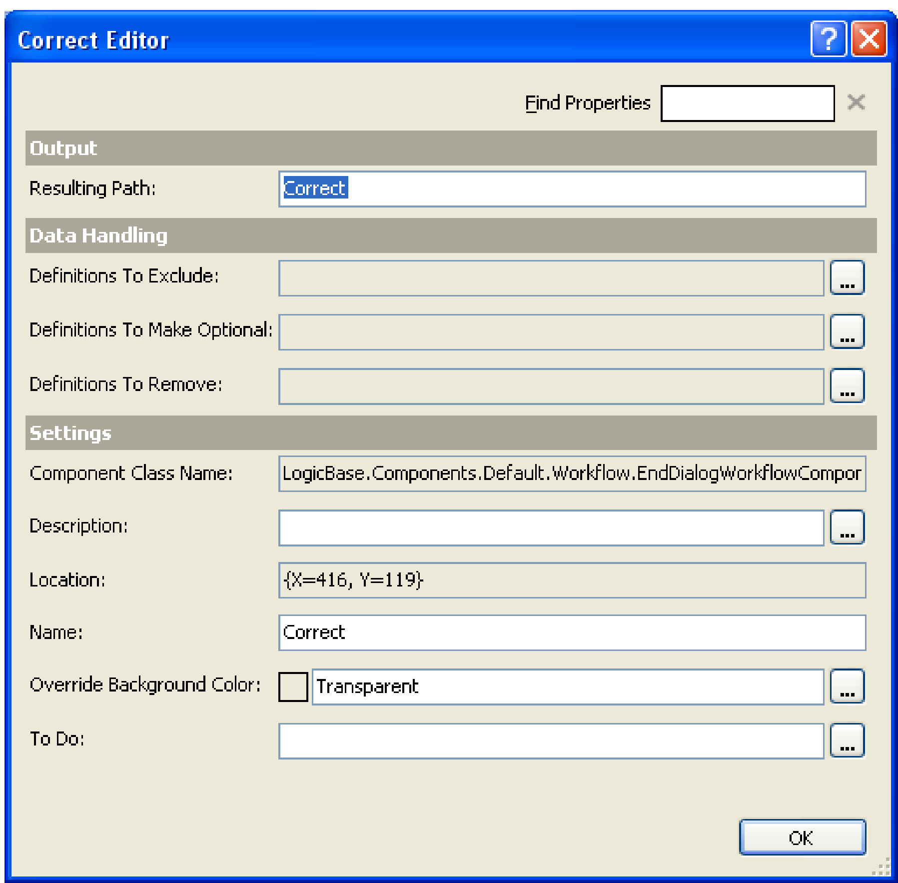
Because Embedded Rule Models can see data outside of themselves, no data needs to be added as input data.

The Embedded Model process starts with a Multiply Values component. This component takes in two variables (**`NumberOne`** and **`NumberTwo`**), multiplies them together, and produces a variable (**`Product`**). The values **`NumberOne`** and **`NumberTwo`** come from user input in the form.

Next, an Equals Rule component compares the computer-generated answer (**`Product`**) with the multiplication problem answer provided by the user (**`AnswerGiven`**).

Finally, two End components complete the Embedded Rule Model process. If the Equals Rule component finds that the variable **`Product`** and the variable **`AnswerGiven`** are equal, then it exits through the "Equals" path to the End component called "Right Answer." If **`Product`** and **`AnswerGiven`** are not equal, then the component exits through the "Not Equals" path to the End component called "Wrong Answer." Each End component points to one of the Embedded Rule Model's outcome paths. The End component named "Right Answer" points to the "Correct" outcome path. The other End component, named "Wrong Answer," points to the "Incorrect" outcome path.

Here is the editor for the "Right Answer" End component:



Here's what the outcome paths on the Embedded Rule Component look like: