CONVERGYS

Design Collaborative

PathMaker

DESIGN COLLABORATIVE

PathMaker

17787 Waterview Parkway

Dallas, Texas 75252

Phone 972.454.8000

Table of Contents

Chapter 1

Introduction

Design is thinking made visual. - Saul Bass

At First Glance

PathMaker is an application which wraps and extends a Microsoft Visio drawing .. Much of PathMaker works and behaves just like Microsoft Visio and isn't covered in this manual. The most obvious difference is that PathMaker works on files with a .vui extension, not .vsd.

The main window is made up of 3 parts: the menu and toolbar area, the stencil and the drawing area.

There is only one set of stencil shapes supported by PathMaker (visible on the left in the screen shot above). The use of other shapes will be flagged and you'll be asked to remove those shapes. In order for PathMaker to work correctly, the only shapes in a design need to come from the supported stencil.

The PathMaker menu items are a combination of Visio functionality as well as custom functions. Likewise, the toolbar is a combination of functions as well. One of the major additions in PathMaker is the ability to add custom data to any shape that you drag out into the drawing. Most

shapes have a dialog box that will pop up when the shape is double-clicked. This dialog box will allow for detailed information to be entered.

Another additional set of functionality allows the design to be exported in different forms. A design can be exported as a Word document or an XML document for use in customer reviews or with other tools.

PathMaker Menu Options

Most of the PathMaker menus support functionality provided by Visio. There are, however, a few custom menu and toolbar options that are PathMaker only.

Validate UI Spec

Validation creates a design "punch list" as it walks all UI paths using every possible data combination and identifies errors and potential design issues. Once a design has successfully validated it is then possible to generate test cases, generate application code, and run automated testing.

This feature is available as a service from Convergys Professional Services when using a Convergys compatible version of PathMaker.

Export High Level Design Doc

Outputs an HLD Word document. This includes only the Visio drawing and space for scenarios to be entered manually.

Export User Interface Spec

Outputs a full User Interface Specification in Microsoft Word. This includes the Visio drawing as well as a table for each dialog state with all the detailed property information.

Import/Export Prompt List

Imports or exports a prompt list. Exporting is most commonly used for getting a list suitable for recording in the studio. It's also used to get a full list of prompts for translation into another language. Importing is used to take a translated list of prompts and change the existing prompts

Deletions are not tracked - meaning PathMaker doesn't keep deleted rows or data around and show it as strikethrough (or some other means). Things that are added or modified are tracked and highlighted appropriately but deletions need to be tracked via the change log.

Chapter

2

Designing

Good design is obvious. Great design is transparent. - Joe Sparano

This chapter will provide the basic knowledge that is needed to get started designing with PathMaker. There are two different sets of information. One is around the design process, which has a couple of steps. The first, creating the High Level Design (HLD) document is essentially an exercise in layout and naming. The second step, filling out the full UI Specification, involves describing every little detail of the design until it is fully specified and can pass validation. We'll take a look at both forms of design, quick and detailed. The second set of information is about components that are used throughout the different dialog states, as well as the states themselves and the other supporting shapes.

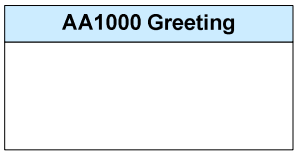
In each editing mode, the dialog states are connected using the dynamic connector shape. Use connectors to create transitions between states. You cannot add transitions using the properties dialog state. You must first create the transition using dynamic connectors and then the transition attributes can be edited using the property dialog.

Quick Editing

PathMaker has several features which allow a basic call flow to be designed without having to enter any detailed information or even open the property sheets for any of the dialog states. This is often the quickest and easiest way to layout and annotate a design for an HLD. All of the information entered this way is saved and will not need to be reentered when you start the detailed design phase.

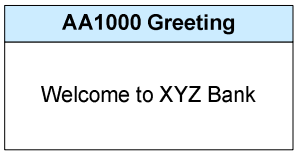
Naming a Dialog State

To initially name or edit the name of a dialog state, first click on it to select it. Then type in the dialog state name. If you want to force a dialog state prefix (see dialog state name under Dialog State Components), you can enter that as well. If you just type the text part of the name, an Id will be automatically assigned. Click outside of the dialog state to deselect it and the dialog state name will be set.



Entering Prompt Wording

To enter an initial prompt for a Prompt or Interaction state, you need to first select the dialog state. Note the highlighted boxes around the entire shape. Wait two seconds and then you can click again in the bottom portion of the shape. This will result in highlight boxes only on the bottom portion of the shape. Now you can type in an initial prompt for the state. Click outside the state to complete the setting of the prompt.



Prompt wording can only be entered in this way up until you open the properties dialog and edit things that way. Once the properties dialog has been used, the quick edit of prompt wording is no longer available. You can still edit the state name however.

Splitting a Transition

If you decide to insert a new dialog state between two others which are connected in a call flow, you can simply drag the new shape out and drop it on top of the connector you want to split. While dragging, you will see a pair of scissors appear when the connector will split. This method will do the right thing with respect to transitions in the states themselves and is often the fastest way to add states to the middle of a call flow.

Labeling Transitions

Labels for transitions can be entered directly on the connectors themselves. Simply select a connector and type in the label for that transition. This will not be used as the condition or option itself but rather as the label that will show up on the connector to make things more readable.

Detailed Editing

The next level of editing is all about entering the details for each state.

Property Sheets

Each dialog state except the terminals and sub-dialog return has a property sheet. It's accessed by double clicking on the state or by using the right mouse button pop-up menu and choosing Properties.

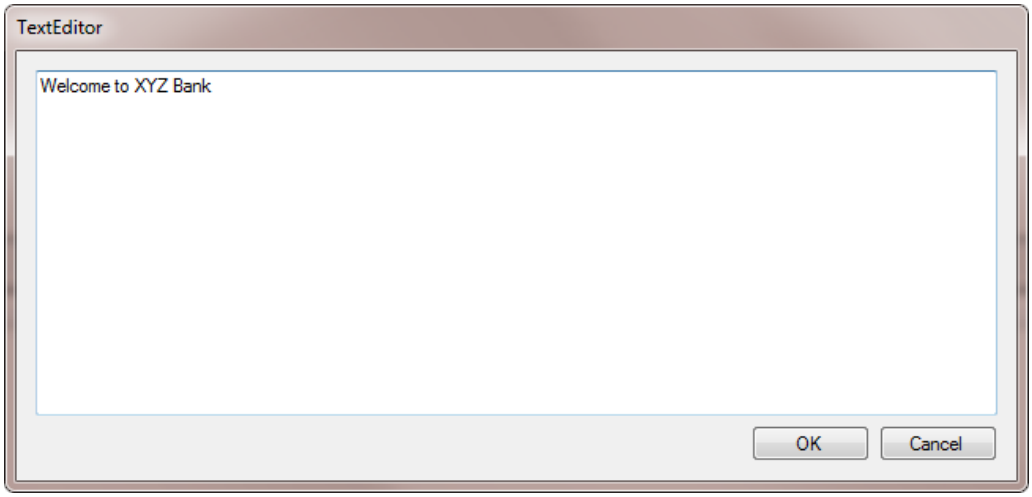
Popup Menus and Hotkeys

Many of the spreadsheet-like grid views used in the property sheets have some hidden functionality available to the designer. In all cases where this functionality exists, there is a pop-up menu available using the right mouse button. The pop-up menu will show the hotkeys which can be used in the grid view.

Examples of grid view actions are moving rows up and down, inserting rows, and accessing the text editor.

Text Editor

In all grid view cells where text entry is allowed, a multi-line text editor can be accessed using the right mouse button pop-up menu or the F2 key. Upon hitting Ok, the contents of the text editor will replace the text in the current cell.



Dialog State Components

Most dialog states are comprised of one or more components of a few basic types. The following is a description of each of those components which are used in more than one dialog state.

State Name

Each dialog state (Play, Interaction, Decision, Data Retrieval, Sub-Dialog) has a dialog state name. Other shapes like terminals can also have names but they are not state names. A state name has two components, a state Id component and a text name component.

The state Id component is a unique identifier containing a two letter alpha and a four digit numeric. Examples would be MM1000, AA1050, etc. This identifier must be unique across the call flow. This uniqueness is enforced (and helped) by PathMaker. For example, when a shape is copied and pasted, PathMaker will change the Id to an unused number.

By default, PathMaker will automatically number Ids if you don't specify your own numbers. The initial 2-letter alpha prefix will also continue to be used until you specify a different 2 letter prefix which will then be used from then on.

The full state name will also include the text name. Spaces are allowed but get transformed to underscores internally so you will sometimes see them expressed as "MM1000_Greeting" when they appear as "MM1000 Greeting" in the properties dialog. When writing conditions that use state names to compare against Current State and Previous State, you must use the version of the name

with underscores in place of spaces. There is always an underscore placed between the id and text portions of the name.

Conditions

Conditions are logic statements used in several places to describe a particular situation. It might be a condition in which a prompt is played or the call flow moves along to another state.

Writing Conditions

The first rule in writing conditions is to use natural language. PathMaker allows you to express the design in common language that will most easily be understood by the client. Here are some examples of simple conditions:

```
if caller is delinquent

if retention offer is available

if payment plan should be offered
```

For readability, reserved words like *if, then, else* are supported but not always necessary. For example, the following two conditions are equivalent:

```
if caller is delinquent

caller is delinquent
```

A blank condition in prompt logic means to reuse the most recent condition:

Condition	Prompt
if account status = gold	Thank you for your service
	We like our gold customers
else if account status = silver	Thank you for calling
	We like our silver customers
else if account status = bronze	Hello

This saves you from having to retype conditions when a whole set of prompts need to be played. If you need to reset the logic and have a prompt which shouldn't be under the previous condition and needs to be played in all situations, use *always*.

Condition	Prompt

if account status = gold	Thank you for your service
	We like our gold customers
else if account status = silver	Thank you for calling
	We like our silver customers
else if account status = bronze	Hello
always	What can I do for you?

This is irrelevant for conditions used in transitions because each transition row needs to have a different condition.

Operators, Reserved Words & Keywords

There are only a few logical and comparison operators allowed in conditions. Logical operators are "and", "or", "not" and left and right parenthesis. Comparison operators are >, <, >=, =>, <=, =<, and =. There are two different allowed representations of greater than or equal and less than or equal. The logic needs to use a single comparison operator:

```
balance > $50
```

~~`$100 > balance > $50`~~

There is also a short list of reserved words which have special meaning in conditions. Those are: if, then, else, otherwise, always. If and then are bookends to start and end a single condition. Else and otherwise are equivalent and provide an alternative to an if statement. Always is used to represent a condition which always equates to true. It is allowed to combine and else and if on the same line as shown in the examples below:

```
if caller is delinquent
else if balance = $0


if balance = $0
else if balance < $1000
otherwise
```

Lastly, there are also a few keywords which have special meaning when used as symbol names or values:

- [illegible]

```
if current state = MM1000_Greeting
```

- [illegible]

```
if previous state = MM1000_Greeting
```

- [illegible]

```
if times here > 1
```

- [illegible]

```
if caller is delinquent = true
```

Nesting

When describing the prompt logic for a play or interaction state, conditions are allowed to be nested. This means that conditions can be used inside of other conditions. It sounds complicated but can really simplify the logic in some cases. See the following long winded example:

```
if caller is delinquent and amount owed > $1000

else if caller is delinquent and amount owed > $2000

else if caller is delinquent and amount owed > $3000
```

Using nested conditions, the same thing can be expressed more simply as:

```
if caller is delinquent

     if amount owed > $1000

     else if amount owed > $2000

     else if amount owed > $3000
```

Because the amount owed conditions are indented under the caller is delinquent condition, it means that they are only tried if the caller is delinquent. There's no limit on the amount of nesting that's allowed but readability does start to suffer at some point. Within PathMaker, the plus and minus buttons surrounding a condition will increase and decrease the

indentation for a condition. You'll notice a chevron symbol appears in the condition text indicating each level of indentation.

Connector Labels

For basic transitions (not command transitions used

Writing Actions

Action entries are multiline but each line can only contain a single action statement. Each line is either a predefined action or an *other* action. Conditions should never be placed in actions. If you feel the need to put a condition in there, you should break the transition into two and put the condition in the condition column.

Predefined Actions

There are several predefined actions which affect the walking of the call flow.

- Play - Plays a prompt type.

  ```
  Play Transition Prompt
  ```

- Assignment - Assigns a value to a symbol.

  ```
  authenticated = true
  ```

- Increment/Decrement - Adjusts a symbol (typically a counter) by 1.

  ```
  increment number of tries
  ```

Other Actions

Outside of the predefined actions, any other lines in the action are simply placed in the generated code for developers to implement.

```
execute payment transaction

send payoff statement fax
```

Prompts

Prompts, sometimes called messages, represent the audio that is played to the caller. They can be auditory icons, recorded messages or text-to-speech (TTS). All are described in the prompt grids of the properties dialogs. Each row of a prompt grid has a condition column, a wording column and a prompt id column.

Conditions

Prompt logic conditions are evaluated as a set. This means that multiple conditions can evaluate to true at the same time and result in multiple prompts being played. The conditions are evaluated top to bottom, taking nesting into account.

Condition	Wording

`if hour of day < 12`	`Good Morning`
`else if hour of day < 18`	`Good Afternoon`
`else`	`Good Evening`
`if DNIS = XYZ Bank`	`Welcome to XYZ Bank`
`else if DNIS = ABC Bank`	`Welcome to ABC Bank`
`else if DNIS = Unknown`	`Welcome to our Bank HelpDesk`
`if account status = Gold`	`It's always good to talk to a Gold customer`
`Always`	`How can I help you today?`

In this case, if the caller calls in at 2PM on the ABC phone line and is not a gold customer, the prompting would go as follows:

Good Afternoon

Welcome to ABC Bank

How can I help you today?

One common confusion when writing prompts is around the use of if versus else or else if. Here are two examples that will hopefully illustrate the behavior.

Condition	Wording
`if hour of day < 12`	`Good Morning`
`if hour of day < 18`	`Good Afternoon`
`if hour of day <= 24`	`Good Evening`

In this example, all three lines are tested and any of them could evaluate to true. If the caller called in after 6PM at night, they would hear the following:

Good Morning

Good Afternoon

Good Evening

In this case the following would be more appropriate:

Condition	Wording
`if hour of day < 12`	`Good Morning`
`else if hour of day < 18`	`Good Afternoon`
`else`	`Good Evening`

In this case, the second condition is only tried if the first fails and the third is only tried if the second fails. In this example, only one prompt would fire at any given time of day.

Wording

The wording entry is used to describe what is played to the caller. For recorded audio, enter the actual text of the prompt. It will be considered recorded audio as long as the first character is not the less-than character.

To attach a recording note to the prompt which will be output when exporting a prompt recording list, enter it between square brackets.

```
Hi, welcome back! [with great joy]
```

The bracketed text can occur anywhere in the wording.

To play data from the backend or TTS, use a <variable prompt> entry. This describes an entry that is unknown at design time and will be "filled in" during the call.

```
Your last payment for…

<last payment amount>

…was received on…

<last payment date>
```

There is one special variable prompt which must be used when confirming a data entry. See Command Transitions for a description of data inputs. Because the confirmation of a variable input (i.e. phone number, zip code, etc.) isn't known at design time, it needs to be variable. But, since the value doesn't come from the backend but instead comes from the recognizer, we use the special notation <result> to show that.

```
I thought you said…

<result>

…is that correct?
```

The prompt wording column can also be used to indicate reuse of a prompt type. This is called a prompt type macro. To reuse a prompt type in the current state or from the Start state, use the curly bracket notation.

```
The last transaction was …

<transaction type>
```

Prompt Ids

Every recorded audio prompt needs to have a unique identifier. While those identifiers can be of any format, the PathMaker tool supports a few common ones automatically.

- Partial (AA1000_i_00) - Numbers each prompt with the state Id, a single letter representing the prompt type and a number.
- Full (AA1000_statename_i_00) - Similar to partial, except the entire state name is included.
- Numeric (0) - All prompts have a unique numeric Id. This format is for backward compatibility and doesn't support automatic numbering although you can renumber all prompts.
- Disabled - No automatic numbering.

Prompt Types

A prompt type is a labeled set of conditions, prompts and prompt Ids. Labeling them as a group allows them to be referenced and played as a unit. Standard prompt type names exist for default interaction state behavior but user-defined ones are also allowed.

Standard Types for Interactions

- Initial - Identifies the set of prompts which are played when a state is transitioned to from another state.
- Retry - Identifies the set of prompts to play following a no match recognition error.
- Timeout - Identifies the set of prompts to play after a no input recognition error.
- Disconfirm - Identifies the set of prompts to play following a negative response to a confirmation.

All of the standard types except initial can be combined with a number to identify which error the set is for: Retry 1, Timeout 2, Disconfirm 3. Simply specifying the type name without a number (i.e. "Timeout") will work for any error number.

Custom Prompt Types

Non-standard prompt types are created simply by typing in a text label. If the text doesn't match a standard prompt type, it's considered custom. Custom prompt types are played using the play action:

```
Play Hold Message
```

Where "Hold Message" is a custom prompt type.

Using Prompt Types

In PathMaker, prompt types can be selected from the pull down list of commonly used types or entered manually. Multiple prompt type names can be specified using a forward slash (/) separator

```
Retry 1 / Timeout 1
```

In those cases, the prompts will apply for any prompt types listed.

The prompt type name only needs to be on the first line of the set of prompts. A blank name indicates the current set of prompts is continuing.

Extending Global Prompt Types

Reusing a global prompt type name defined in the Start state does not override it, it extends it. The global prompts of a given type will play before the individual state prompts of the same type. This allows for common prompt prefixes like "I'm sorry" to be played before any retry prompts defined in an interaction state.

Transitions

A transition describes when and how a state will go to another state. All transitions have a condition, an action and a goto. Command transitions have some additional information and transitions used for Max Handlers have predefined conditions.

Creating Transitions

All transitions except for global commands in the Start state and Max Handler ones are created by placing a connector between two states in the Visio drawing. This will automatically add a new row to the transition table of the originating state (non-arrow side of the connector).

New rows can simply be entered in the command transition table in the Start state and the Max Handling rows are already populated with their predefined conditions. Simply fill them out to make them active. For both of these, there is no connector to show the next state so it is simply picked from a pull down list in the goto column. Included in that pull down are some predefined goto targets that can be chosen:

- Current State - This causes the walking to stay in the current state, it doesn't transition out of the state. See the tips section for more information on how this works.
- Return - Returns from the current sub-dialog.
- Hang up - Results in an application hang up.
- Transfer - Transfers the call.
- PlaceHolder - Temporary place holder. Behaves like a hang up but shows up differently in documentation.

Conditions

Conditions for transition are evaluated top down and the first one that evaluates to true wins. Only one transition can fire. Nesting is not allowed for transition conditions so the evaluation is solely based on the order the conditions are placed in.

Tips

A design can avoid transitioning to another state using the Current State goto target. A transition will also be avoided if a state provides a transition to itself using a connector. The rule is that, if a state goes to itself, it will not transition but will instead stay where it is. It does not re-enter the state, does not play any prompts by default and all error counters (timeout, retry, disconfirm, total) stay as they are.

This sounds confusing but is a necessary and powerful ability that allows designers to implement commands like repeat and help in interaction states. In order to play a prompt when staying within a state, use the Play statement in an action to note the prompt type you want played.

Command Transitions

A command transition wraps a transition with additional information for each option available to the caller. Each option represents a grammar item that the caller can speak or enter via a DTMF key. The new fields are Option, Vocab, DTMF and Confirm.

Option

The option is a simple identifier for a single user input. It's also used as a connector label for the transition. As with condition labels, you can use square bracketed entries to supply an alternate label.

```
Make a Payment [Pmt]
```

A single option can have multiple rows if there are conditions which would result in different actions or gotos for that command. In those cases, the option name should be on each line but only the first row should have Vocab and/or DTMF entries.

Dynamic Commands

Typically PathFinder will expect that a command is available under any and all circumstances and will raise an error if the conditions for an option do not cover 100% of all situations. In some cases, however, the designer only wants a command available under certain situations. This is commonly called a dynamic grammar entry. To designate that an option should only be available under some conditions, use the <dynamic> notation.

```
Make a payment <dynamic>
```

In those cases, PathFinder will assume that the option is only available when the condition or conditions for that command evaluate to true. In all other situations, the option will be removed from the grammar.

Vocabulary

The speech vocabulary column (labeled "Vocab") is used to notate one or more example entries for the speech grammar developers. Multiple examples need to be comma-separated.

```
make a payment, pay my bill
```

Where a data input is expected, for example, a zip code or phone number, use the variable bracket format.

```
<zip code>
```

Additional information can be put inside the brackets as well.

```
<zip code: 5 or 9 digits>
```

DTMF

The DTMF column is used for the DTMF key grammar entry. Outside of the special cases listed below, this should be a single digit, * or #.

As with vocabulary, data inputs are noted with the <> entries.

```
<zip code: 5 or 9 digits>
```

A dynamic assignment of a DTMF key at runtime should be noted using a string which will not be confused with a single digit key or a data input.

```
X
```

Confirmations

Designing confirmations is a two-step process. First the command transition's Confirm column entry must be set appropriately. Then the confirmation prompts must be entered for that command. There are three different confirm settings for an option:

- Never - No confirmation will be done.
- If Necessary - Confirmation will be done if the recognition result score doesn't meet the accept threshold. Has no effect on a DTMF entry.
- Always - Confirmation will always be performed.

As described in the prompt section of this document, data inputs have one special variable prompt, "<result>", which is used to refer to the recognized input.

Overriding Global Commands

Global commands are specified in the Start state but sometimes require overriding to provide different behavior in a particular state. To do that, you override the global simply by using the same option name within an interaction state. Every transition row from the Start state is replaced with any entries in the current state.

Max Handler Transitions

Max handler transitions define what happens when the maximum number of errors is reached in any one state. The default transitions are defined in the Start state but can be overridden in any interaction state. The difference between a max handler and a standard transition is that the error conditions are predefined and each error condition has a counter specifying the maximum number of those errors that are allowed.

Error Conditions

- Max Retries - Evaluates to true when the number of no match errors for the current state *exceeds* the count.
- Max Timeouts - Evaluates to true when the number of no input errors for the current state *exceeds* the count.
- Max Disconfirms - Evaluates to true when the number of negative responses to confirmations for the current state *exceeds* the count.
- Max Total Errors - Evaluates to true if the total number of errors which are configured to count towards total errors *exceeds* the count. Which error types count towards total errors is configured on the default settings tab of the Start state.

Goto

Because max handling transitions are not created using connectors, the goto states need to be selected from a pull down list. That list will contain all of the states in the VUI as well as some predefined entries: Current State, Return, Hang up, Transfer and Place Holder.

Overriding Default Handlers

The default handlers specified in the Start state can be overridden in individual interaction states in one of three ways.

- Entering a new count will keep the rest of the default behavior, only changing the count.
- Entering an action or goto overrides the action and goto in the default behavior - they always work in tandem with each other.
- Entering a new count and action and/or goto will completely override any default behavior.

Developer Notes

This is simply a comment field that is passed through to the developer.

Special Settings

This is also a comment field passed through to the developer but is meant to describe special settings requested for a recognition. Things like different speech timeouts, etc. should be specified here.

Dialog States and Other Shapes

Each shape in the PathMaker Visio Stencil represents a dialog state or other supporting shape. This section will describe each as well as the properties it may have. For any shapes which have property dialogs, the properties are accessed by double-clicking on the shape or using the right mouse button pop-up menu and selecting Properties.

Document Title

This shape contains basic information used in printing out the HLD and UI Specifications: client name, project name, client logo. Only one Document Title shape is allowed per UI.

Change Log

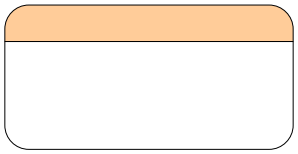
The change Log shape tracks the dates and versions of changes made to the document. It requires manual updating to create a version and allows for the highlighting of changes for a particular version. Only one Change Log shape is allowed per UI.

Start

The Start shape is used to specify application defaults and the start state of the call flow. It contains default settings for the application, variable initializations, global prompts and commands and default max error handling. Only one start shape is allowed per UI. Key points are as follows:

- Global prompt types are played prior to but in addition to any of the same type in the current state.
- Global commands are available in every interaction state which does not have an option with the same name.

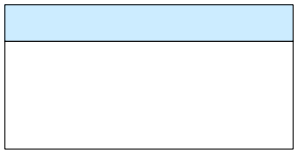
Interaction



An interaction state is used to interact with the caller using speech or DTMF recognition (or a combination). It contains prompt types, command transitions and confirmations, max error handling transitions, special settings and developer notes. Key points are as follows:

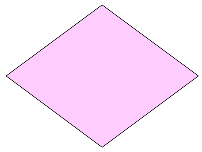
- Reusing a global prompt type name doesn't override it, it extends it. The global prompts will be played prior to the state prompts.
- Global commands can be overridden by creating a command transition using the same option name as the global.
- Default max handlers can be overridden in two ways. Just the count can be overridden or the entire transition can be overridden.

Play



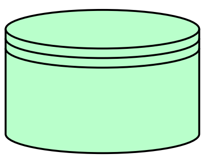
The play state is used to play prompts to the caller. It contains prompts, transitions, special settings and developer notes.

Decision



The decision state is used when branching to different states based on conditional logic. Typically, branching can be done directly in any of the other play, interaction or data retrieval states to avoid an extra shape in the drawing. Sometimes, however, a decision is used to clarify or simplify the logic. Each decision contains transitions and developer notes.

Data Retrieval



There's a lot of confusion around the use of the data retrieval element in the PathMaker VUI design tool. This document is meant to clarify the purpose of that element and when it should and should not be used in user interface designs.

Purpose

The purpose of the element is to show places in the user experience where the result of a backend data request could fail and that failure would result in a unique user experience for the caller. By unique, we mean that we do not need to design every place where the catastrophic "backend down, transfer to rep" behavior is used.

One example use of the data retrieval element is where a payment is being made. The element can be used where the user interface waits for the transaction to post. Based on the result, the UI reads the caller a transaction id or tells them that the payment failed for one reason or another.

This behavior can be demonstrated in any play, interaction or branch on condition state with the use of a simple business rule condition "if payment was committed." However, the data retrieval element is best used to communicate that the retrieval may take some time and cause a delay in the experience.

Don'ts

- [illegible]

- [illegible]
- [illegible]
- [illegible]

Dos

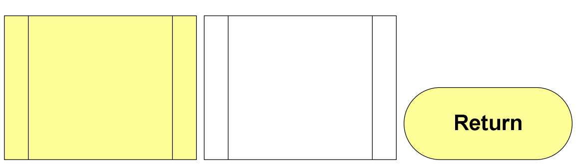
- [illegible]
- [illegible]

Terminals: Transfer, Hang up, PlaceHolder

These states all behave the same but have different meanings. All are used to terminate a call flow. They have no properties other than their simple name (not a state Id). That name will show up in the UI specification for readability purposes.

- Transfer - Shows a call transfer or a transfer to another application.
- Hang up - Shows application termination and hang up on the caller.
- Place Holder - Shows a temporary place holder. None of these should exist in the final design.

Sub-Dialogs: Define, Call, Return



Sub-dialogs are used to identify a reusable group of call flow states. There are three components related to sub-dialogs:

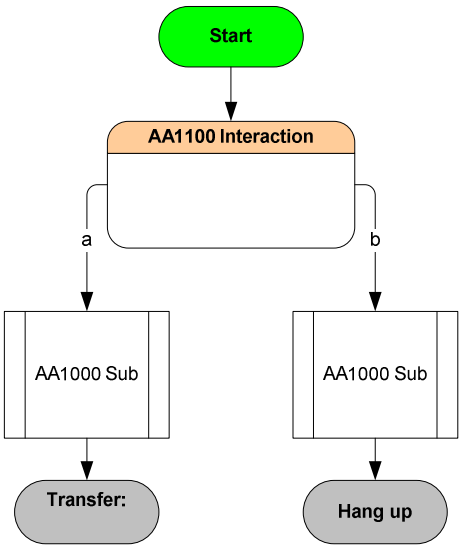
- The *definition* is the design of the reusable group itself. Each definition starts with a Define Sub-Dialog shape.

- The *call* is where the main call flow calls the sub-dialog definition. It's represented by a single Call Sub-Dialog shape.

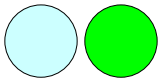
- The *return* is where the sub-dialog goes back to where it was called from.

Sub-dialogs are best used for isolated, repeatable functionality like authentication, address collection, etc.

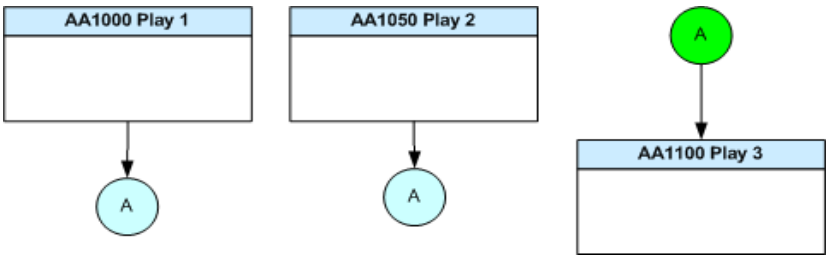
There's one key restriction around the use of sub-dialog calls. You will get an error from PathFinder if a state has transitions which call the same sub-dialog two (or more) times and has different next states as shown below.



On-Page References



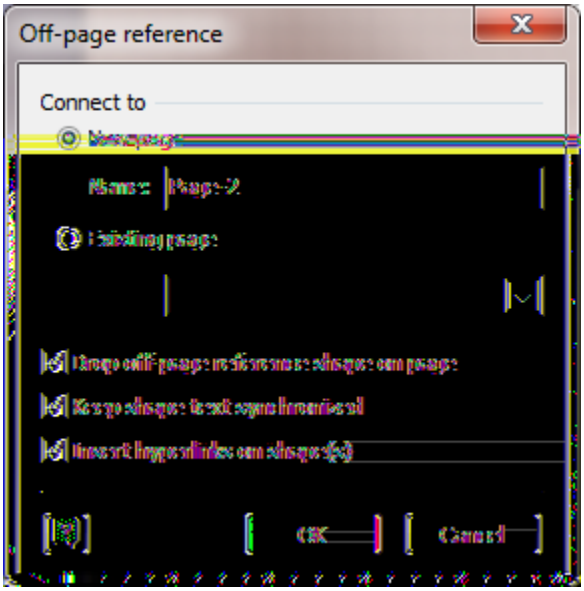
On-page references simplify a drawing, replacing long or confusing connections. Incoming and outgoing references are paired using the shape text label. Multiple incoming references can point to a single outgoing reference.



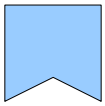
Off-Page References

Off-page references allow states on one drawing page to connect to shapes on another page. Visio manages the relationship between references. When you place an off-page connector on a page, Visio will bring up the Off-page reference dialog and let you choose the page to connect to (or create a new one).

The off-page reference shape names do not have to match, but it is highly recommended to aid readability. Visio will keep the names in sync if you choose the "Keep shape text synchronized" option.



One common technique used by designers to distinguish incoming versus outgoing off-page references is to use the right mouse button pop-up menu and choose Incoming to change the shape of the reference.



Dynamic Connector

Connectors are used to show the call flow transitions from one state to another. Adding a connector automatically adds a transition row to the state on the non-arrow side of the connector. Removing the connector will remove the transition row. It's important to note that even temporarily disconnecting a connector to move it to another part of the shape will remove the transition and add it back. This will result in the clearing of all the transition data except for the goto. To move the non-arrow side of a connector on a shape without losing the transition data, use the Shape->Move connector being point menu options or their associated hotkeys.

Connector labels are automatically created using the condition or option text from the state being transitioned from. Those default labels can be overridden by adding a label in square brackets [] along with the option name or condition.

```
if caller is delinquent [Delinquent]
```

Comment

The comment shape is used to place clarifying notes on the drawing page. They have no effect on the call flow.

Chapter

3

Symbology

The definition of an expert is someone who knows what not to do. - Charles Wilson

Conditions and actions are written using *symbols*. A symbol is a set of characters, starting with an alpha, which represents a single variable or business rule. A *variable* is created in PathMaker for use in designing the user interface. A *business rule* is referenced in PathMaker to identify something which exists in the client backend or the platform (outside of the UI).

Examples of symbols:

```
caller is delinquent

balance

counter
```

Case and whitespace are irrelevant, the following are all equivalent:

```
Caller is delinquent

caller is delinquent

callerIsDelinquent
```

And the symbol name must start with an alpha character. The following is not a valid symbol name:

```
~~10 days overdue~~
```

Symbols cannot include the special characters " [ ] ( ) = < > but they can include any other non-alphanumerics. For example, the following are all valid:

```
Caller isn't allowed access
```

```
caller_is_delinquent
```

Types & Values

There are four types of symbols allowed in PathMaker conditions and actions: Boolean, Number, Money and Strings.

Boolean

Boolean symbols equate to true or false. They can be used in a condition stand alone or using an equals sign and the reserved words "true" or "false". A Boolean symbol can only be assigned a value of "true" or "false".

Examples of setting a symbol value in an action:

```
authenticated = true

first payment = false
```

Examples of testing Booleans in conditions:

```
if authenticated

if authenticated = false

if not authenticated
```

Number

Number symbols are used to represent positive (0 included) integer values. In addition to setting a value, number symbols can also be incremented and decremented in actions.

Examples of setting a number symbol in an action:

```
counter = 1
```

Increment/Decrement examples:

```
increment counter

decrement counter
```

Examples of testing a number symbol in a condition:

```
if counter = 0

if counter >= 5
```

Money

Money is really no different than a Number symbol under the hood but allows for a preceding dollar sign and option decimal point followed by two numbers. Here are a few examples:

```
balance = $50

if payment amount < $49.99
```

Strings

String symbols are used to represent lists of values - including non-mathematical numerical things containing numbers like zip codes and phone numbers. If a string value is comprised of numbers, use double-quotes to identify the value as a string value and not a number. You should also use double quotes to preserve spaces within a value.

```
phone number = "8005551212"

if authentication type = "2 stage"
```

Otherwise, assignment and checks for equality are no different than other symbols.

```
language = English

if language = Spanish
```

The design should explicitly check for each value that a string symbol can have. By default, the algorithm only tries the different values for a string symbol that it finds in the design (in conditions or via assignment in actions). For example, the following will not generate correct or complete test cases.

```
if string = a

else if string = b

else
```

That last else will never generate a test case because no other values are known for "string" other than "a" and "b".

This behavior is the default. See the section on Testing Strings in the Advanced Topics chapter.

Initialization

All symbols representing variables (not business rules) must be initialized prior to using them in a condition. Otherwise the platform or backend will be asked to supply the data.

Variables can be initialized using actions or using the initialization tab of the Start state.

Units Crossover

When units are added to a numeric comparison for readability, the underlying symbol names are modified.

```
if last payment made > 30 days
```

is equivalent to

```
if last payment made days > 30
```

and results in a number symbol being created instead of trying to change "30 days" into a string value.

Under the Hood

Internally, PathMaker takes all the natural language text used for symbol names and makes "crunched" versions of them internally for storage and comparisons. This results in symbol names which have white space removed and uses mixed case (sometimes called Camel-Case) naming. For example:

```
caller is delinquent
```

crunches to

```
CallerIsDelinquent
```

This is important to understand for two reasons. The first is that many different text values can refer to the same underlying symbol:

```
CallerIsDelinquent

callerisdelinquent

caller is delinquent
```

The other reason this is important to understand is that all of the PathFinder output including validation results and debugging aids, use the crunched versions of the names.

Tips

Avoid Client Backend Details

Probably the most important rule of thumb to use when designing is to always strive to keep the business rules at the highest possible level. It's often convenient to just reproduce the client backend logic and details in the design but it can have a significant negative impact:

- The readability of the design is decreased
- The complexity of the logic is increased
    - Which impacts the amount of testing required
    - And the amount of time to process the VUI
    - And the amount of code that needs to be written
- Debugging and fixing issues in the design also increases in complexity

Always try to describe the function, rather than show the code that's required.

```
if atflag = "a" or atflag = "b"
```

should be

```
if account type = business
```

Use Natural Language

Just because you know what the names of the variables used in the underlying code are or what the client backend code calls them, doesn't mean you should use them in the design. Just as developers over the years have moved towards coding with more and more descriptive variable names, so should you. PathMaker supports natural language symbols so use them. Don't try to make it read like code with underscores and prefixes, etc.

```
caller_status

accttype

sFlag
```

is always better as

```
caller is delinquent

account type
```

```
account status
```

Single Symbol, Multiple Values

Use a single symbol when representing different options for the same thing. For example:

```
if English

else if Spanish
```

should be

```
if Language = English

if Language = Spanish
```

Since English and Spanish are just different values for the same thing. If you don't tie these together using a single symbol, it looks like English and Spanish are different Boolean symbols which means they could both be true at the same time…

Single Symbol Name

When referring to the same business rule or variable throughout the design, you must use the same text. For example:

```
if business user

if caller is a business user
```

are two different symbols. By not using the same phrasing, this sets up a situation where business user could be true and caller is a business user could be false…

Different Symbol Names

You do, however, need to use different symbol names to distinguish unique situations. For example, if you have two different data retrieval states and you want to check for transaction success in each of them, you *must* use a different symbol value. Otherwise, PathFinder will assume those two symbols refer to the same check for success.

```
if transaction success
```

should be

```
if transaction success in state 1

if transaction success in state 2
```

Don't Use Strings as Booleans

There's more work involved in the design, development and QA phases if string symbols are used as Booleans. Instead of a simple true/false value, Boolean strings must always be checked against the same values. If written incorrectly, this will lead to unintentional extra values. Also, a simple Boolean is implemented and tested as true and false. A string needs to be coded to be set to each off the values found in the design by the developer.

```
if switch = on

if switch = off
```

should be

```
if switch is on

if switch is on = false
```

Chapter

4

Advanced Topics

If I'd asked people what they wanted, they would've said a faster horse. - Henry Ford

Duplicate Prompts

PathMaker is set up to handle prompts with duplicate wording in a specific manner. The goal is to minimize the work for the designer, developer and voice talent at the expense of a little more file management work by the recording studio. But, this technique also minimizes the chance of mistakes being made when changes are made to a design. Here are the steps:

1. All prompt wordings are entered individually in PathMaker and assigned unique prompt Ids.

2. The prompt list output will show a single row for all prompts which have the same wording. Multiple prompt Ids will be listed along with the single prompt wording.

3. The voice talent will record a single version of the prompt.

4. The recording studio will create copies of the audio file for each of the prompt Ids listed.

Following this process results in the following benefits:

- Changing the wording in a single spot in the design doesn't require giving a prompt a new prompt Id or searching to make sure that the prompt Id is unique throughout the design.

- The tools do not need to check every prompt wording entry and modification to see if another prompt wording exists with the same text. This would significantly slow down prompt wording entry during design.

- The risk of making a change which inadvertently affects several spots in the design is eliminated. This might occur in design or even in development if a patch is made to a deployed application.

So, what happens if you decide to handle duplicate prompt Ids in another manner? In that case, you are responsible for managing the prompt Ids, file naming and recording lists yourself. In addition, you'll need to communicate to others that the prompt files are not unique and changing one could result in unintended behavior.

Error Handling Details

There are four predefined error handling conditions used in max error handling. Two of these - retries and timeouts - refer to the recognition errors received from the platform. Disconfirms are manufactured errors that occur whenever a caller says "no" to a confirmation request. The last - max total errors - is a counter that corresponds to a combination of the other errors.

Count

Maximum Allowed

In all cases, the counters used in the max handling transitions dictate the maximum number of errors of a particular type that are *allowed*. For example, if you have max retries set to 2, you should supply a retry 1 and retry 2 set of prompts.

Individual Counters

Each error type has its own counter, including the total errors. Having independent counters means that the prompt types selected are based on the individual counters, not an overall count. For example, if a retry is followed by a timeout, the prompt types that would be played would be a Retry 1 followed by Timeout 1.

Calculating Total Errors

The Start state has configuration settings for which of the retry, timeout and disconfirm errors are used to calculate total errors. This was done because some designers prefer to not have disconfirms count towards the total error count.

Given the configuration settings, the total error count formula is as follows:

total errors = 0

if retries count towards total errors, total errors = total errors + retries

if timeouts errors count towards total errors, total errors = total errors + timeouts

if disconfirms count towards total errors, total errors = total errors + disconfirms

Max Handling Transition Order

There can also be confusion around what happens when one of the error counters causes more than one max handling condition to be true at the same time. The behavior in this case is just like other transitions - the conditions are evaluated top to bottom and the first one the evaluates to true is exercised. Here's the detailed logic:

> if retries > max retries, exercise transition
>
> else if timeouts > max timeouts, exercise transition
>
> else if disconfirms > max disconfirms, exercise transition
>
> else if total errors > max total errors, exercise transition