UNIFY Harmonize your enterprise

Creating NLU IVR Applications with OpenScape Media Portal and Fusion Application Builder plus supporting Grammar Studio

Summary

This paper describes how to deploy on a Win64bit OS the so called **'Media Server Starter Kit'** which includes the **OpenScape Media Server** (without Symphonia Framework) and the **OpenScape Fusion Application Builder** for generation of IVR Call flows, examples, **TTS and ASR engines** and more as Not-for-Resale (NFR) versions. This Starter Kit shall facilitate access to a test and development environment while the sold version supported for productive IVRs needs the OSMS as UC Application module / with a UC setup running on Suse Linux. Of course, without a UC based installation Controls interacting with UC Applications cannot be used herein.

This document provides a tutorial describing step by step how to build a first simple IVR application - how to generate it, how to test it and how to deploy it.

A second part of this paper describes the solution's **Natural Language Understanding** (NLU) capabilities and the so called **Grammar Studio**, a tool which facilitates generation, extension and testing of NLU applications and seamless works with the Application Builder. Grammar Studio is today only available from within the Starker Kit.

Tutorials describing how to generate NLU applications, best practices and more complete this document to a comprehensive guide.

Version: 1.1

Date: 08/04/2014

Author: Wolfgang Schiffer et. al.

Unify Software and Solution GmbH & Co. KG
München
Deutschland

Table of Content

2. History of Changes

< 8/2011	Wiki Pages as successor/source	W.Schiffer + team
8/5/2011	First Wiki Export: Version not including NLU	J.H.Krüger
8/11/2011	Second Wiki Export, now complete - including Grammar studio and NLU tutorials etc.	J.H.Krüger
8/04/2014	Updates + Reskinning to Unify Brand	J.H.Krüger

3. Introduction

The *Media Server Starter Kit* contains an IVR development environment, which consists of the following components:

- OpenScape Media Server (without Symphonia Framework)
- OpenScape Application Builder
- OpenScape Grammar Studio
- OpenScape Desktop Client (OptiClient)
- Nuance ASR/TTS
- Sample Applications

With the *Application Builder* and the *Grammar Studio* you can create, test and deploy IVR applications.
The applications are deployed on the local *Media Server*. In order to call the applications the *OpenScape Desktop Client (OptiClient)* can be used.

This Starter Kit shall facilitate access to a test and development environment while the version supported for productive IVRs needs the OSMS as UC Application module / with a UC setup running on Suse Linux. Of course, without a UC based installation these Controls which are interacting with UC Applications cannot be used.

This document provides a tutorial describing step by step how to build a first simple IVR application - how to generate it, how to test it and how to deploy it.

A second part describes the solution's Natural Language Understanding (NLU) capabilities and the so called Grammar Studio, a tool which facilitates generation, extension and testing of NLU applications and seamless works with the Application Builder. Grammar Studio is today only available from within the Starker Kit.

This document contains the following sections:

- Prerequisites
- Installation & Startup of Media Server Starter Kit including Media Server, TTS, ASR, the Application Builder and an OpenScape Desktop Client (OptiClient) to test with
- Generation, test and deployment of an example IVR application
- Description of the Grammar Studio and Grammars with DIANE engine
- Tutorials
- Knowledge Base and Best Practises
- Q&A
- Known Issues
- List of example applications

The Prerequisites section provides some general information regarding the development environment. Please make sure to read that section before the installation of the Media Server Starter Kit.

The Installation & Startup section guides you through the installation of the Media Server Starter Kit and the OpenScape Desktop client. In addition it describes how to configure the OpenScape Desktop client to operate with the Media Server.

4. Prerequisites

This section describes some prerequisites to be considered before the installation of the Media Server Starter Kit.

4.1. Operation System

The Media Server component requires a 64 Bit Windows operating system. The Media Server Starter Kit was created and tested based on Windows 7 64-Bit operating system, but older Windows operating systems, as long as they are 64-bit, should work as well.

4.2. Multiple Network Adapters

If more than one Network Adapter is available on your workstation, please make sure that during the operation of the Media Server all but one Network Adapter is disabled. Especially in case you are using virtualization software (e.g. VMWare) make sure to disable the additional Network adapters.

4.3. Hardware Requirements

The Media Server Starter Kit requires a minimum of 4GB RAM during operation and at least 4GB available hard disk space. More hard disk space may be required depending of the size of your applications.

4.4. Microsoft DirectX

The OpenScape Desktop Client requires Microsoft DirectX for operation.

4.5. SIP Configuration, Ports

The OpenScape Desktop client will use the port 5060 on the workstation.
The Media Server will use by default the ports 5066 & 5067 on the workstation.
The Nuance Speech Server will use the ports 5062 & 5063.

Please take care that these ports are not used by other applications on your workstation.

4.6. Additional requirements to use certain features

Some applications may require internet access (e.g. Weather Application sample)

5. Installation & Startup

This section provides information regarding the installation, configuration and start of the Media Server Starter Kit and the required software components

5.1. Install the Media Server Starter Kit

Please download the following setup components to *one* directory on your local workstation:

- Setup Media Server Starter Kit
- Setup Nuance Speech Server
- Setup Nuance TTS
- Setup Nuance ASR

After the download, start the installation of the Media Server Starter Kit with ms_starterkit_setup.exe.

Please note: The Nuance setup components are installed automatically by the Media Server Starter Kit setup. They cannot be executed directly.

During setup only the target directory needs to be provided. Once the setup is finished the following directories are created in the provided target directory:

```
+---application_builder
+---application_host
+---grammar_studio
+---jre
+---nuance
+---samples
\---workspace
```

5.1.1. Description of Directories

- application_builder: The Application Builder
- application_host: The Media Server
- grammar_studio: The Grammar Studio
- jre: JRE, shared between Application Builder, Media Server and Grammar Studio
- nuance: Nuance ASR/TTS and Nuance Speech Server
- samples: Sample, which can be imported into the Application Builder
- workspace: Default workspace, shared between the Application Builder and the Grammar Studio
-

5.1.2. Menu Entries

During setup a new folder 'Unify' is created in the start menu. It contains the following entries:

- OpenScape Application Builder
- OpenScape Grammar Studio
- OpenScape Media Server Starter Kit Uninstall
- Start Server components

5.2. Uninstall the Media Server Starter Kit

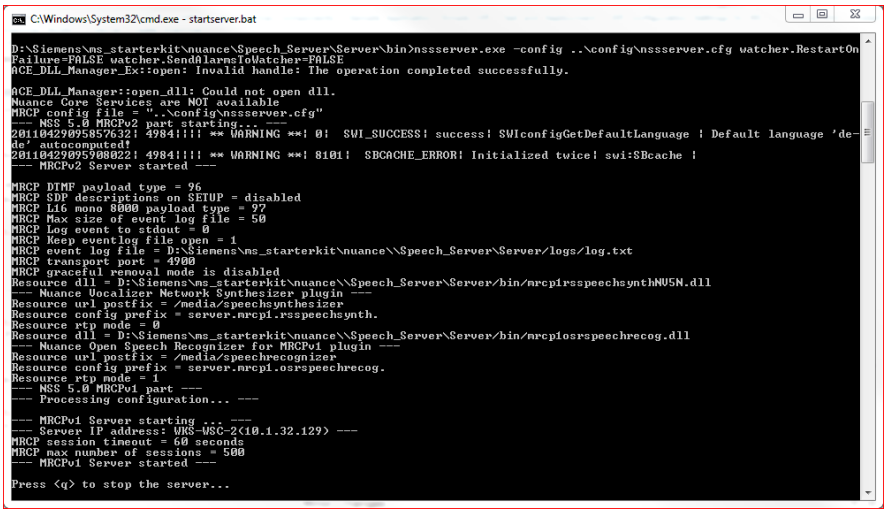
In order to uninstall the Media Server Starter Kit start the uninstaller 'OpenScape Media Server Starter Kit Uninstall' from the Start Menu or from the 'Programs and Features' control panel plugin.

The Start Menu entries and the target folder will be removed.

5.3. Startup of Server Components

After the installation of the Media Server Starter Kit the server components (Media Server & Nuance Speech Server) can be started by means of the 'Start Server components' entry in the Unify folder of the start menu.

It will start the Nuance Speech Server in one console window and the Media Server in a second.



5.4. Shutdown of Server Components

In order to shutdown the Nuance Speech Server, type 'q' in the console window.
In order to shutdown the Media Server, type 'exit' in the console window.

5.5. Startup of Server Components separately

In order to start only the Nuance Speech Server, execute the script '..\nuance\startserver.bat' in the folder where the Media Server Starter Kit is installed.

In order to start only the Media Server, execute the script '..\application_host\bin\start-host.bat' in the folder where the Media Server Starter Kit is installed.

5.6. Install OpenScape Desktop Client

Please download the OpenScape Desktop client (OpenScapeClient-V40-R0.1.6.zip) to your workstation. Unzip the file to a directory and install the OpenScape Desktop Client by executing 'setup.exe'.

Details regarding the installation can be found in the document 'OpenScapeClient_Release_Notes.doc' located in the same directory as the 'setup.exe'.

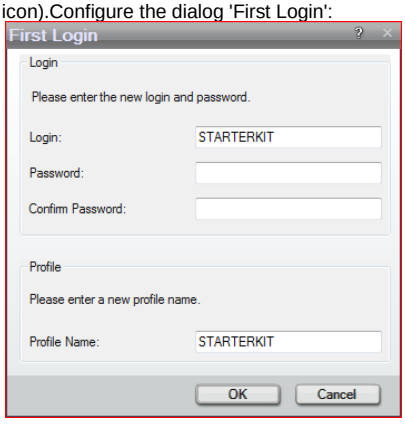
During installation you should use the default values. More precisely:

- Choose 'Personal Edition'
- Choose 'SIP Provider' as Standard Provider Module
- Do not use 'Central configuration'

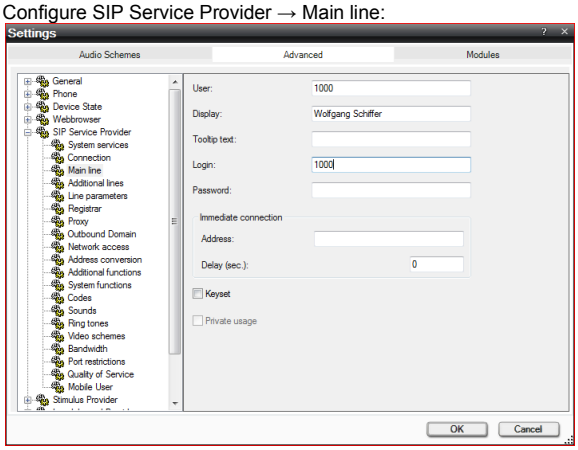
5.7. Configuring OpenScape Desktop Client

After the installation, the OpenScape Desktop Client needs to be configured:

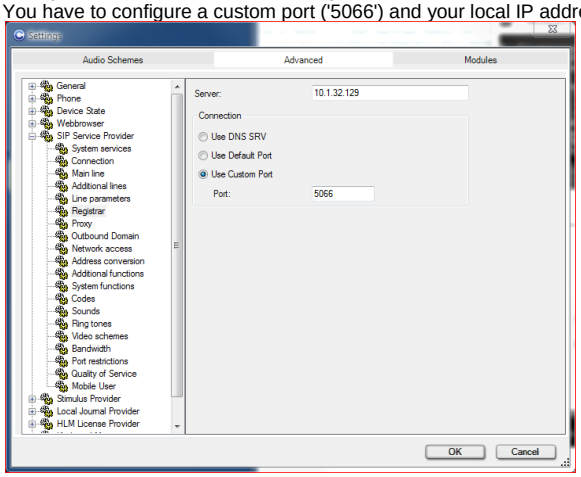
- Start the OpenScape Desktop Client (from the Unify folder in the Start Menu or the Desktop icon).Configure the dialog 'First Login':



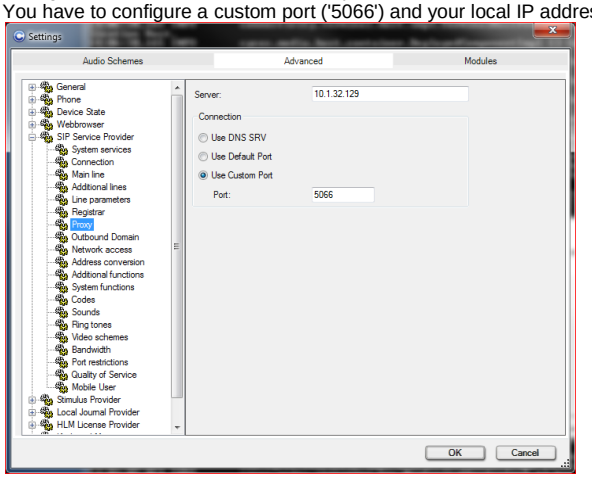
Configure SIP Service Provider → Main line:



Configure SIP Service Provider → Registrar.
You have to configure a custom port ('5066') and your local IP address:



Configure SIP Service Provider → Proxy.
You have to configure a custom port ('5066') and your local IP address:

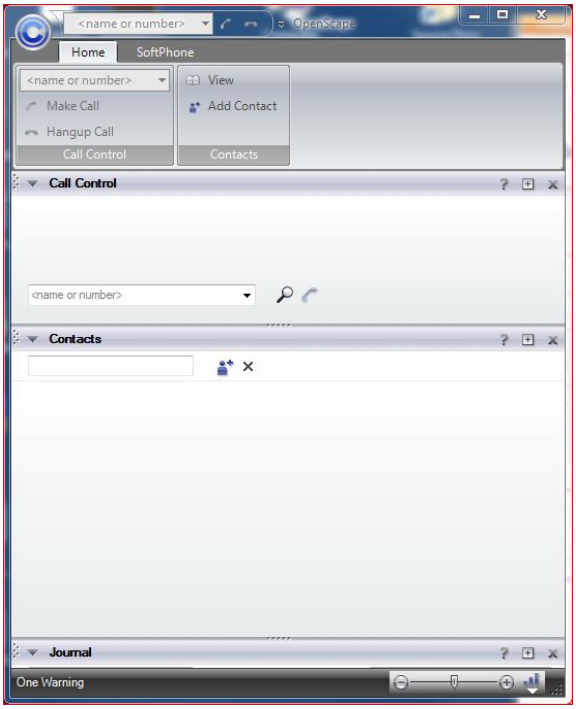


5.8. Start of OpenScape Desktop Client

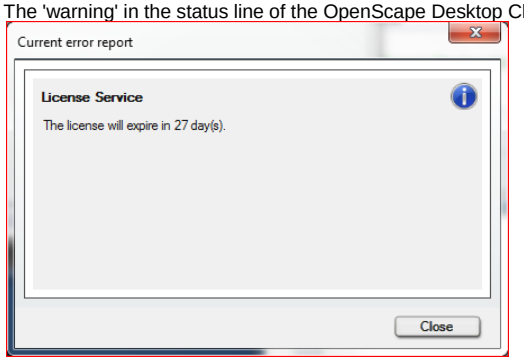
Start the OpenScape Desktop Client (from the Unify folder in the Start Menu or from the Desktop icon). Assuming that the configuration described above is made correct, you should notice the registration of the Softphone in the Media Server console window.

The OpenScape Desktop Client configuration as described here is not licensed, so the full feature set is limited to a grace period of 30 days. After this grace period only the most basic SIP features (make call, hangup) will work.
However, this is sufficient for the Media Server Starter Kit.

Once started, the OpenScape Desktop Client should look this:



The 'warning' in the status line of the OpenScape Desktop Client is the license warning, which can be ignored.

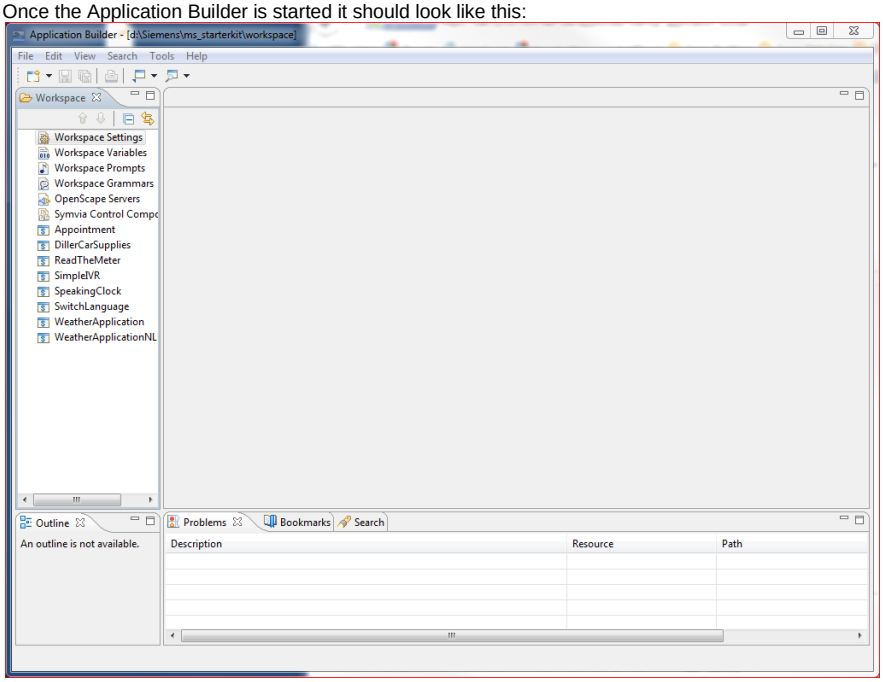


In case any other SIP Soft phone should be used, please note that the Media Server is using the ports 5066, resp. 5067 for the SIP communication here.

5.9. Startup of OpenScape Fusion Application Builder

Start the OpenScape Application Builder (from the Unify folder in the Start Menu). During the setup a 'default' workspace is installed on the workstation. The workspace already contains a couple of sample applications.The Application Builder automatically starts with this default workspace so that nothing else needs to be configured. Please note that the workspace is shared between the Application Builder and the Grammar Studio.

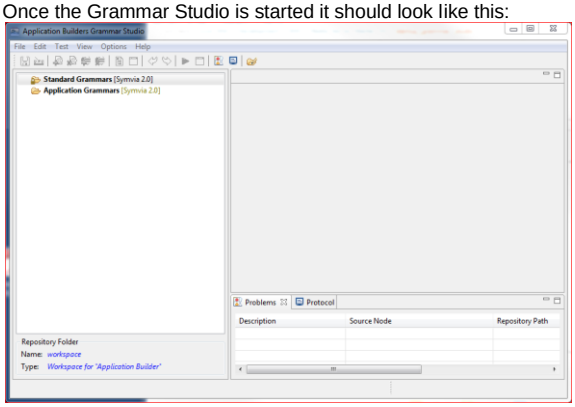
Once the Application Builder is started it should look like this:



5.10. Startup of OpenScape Grammar Studio

Start the OpenScape Grammar Studio (from the Unify folder in the Start Menu). During the setup a 'default' workspace is installed on the workstation. The workspace already contains a couple of sample applications. The Grammar Studio automatically starts with this default workspace so that nothing else needs to be configured. Please note that the workspace is shared between the Grammar Studio and the Application Builder.

Once the Grammar Studio is started it should look like this:



6. Fusion Application Builder

The *Application Builder* is designed to create IVR applications for OpenScape.

This chapter contains the following contents:

- **General Overview**
- **Tutorials** (Test and deployment of a sample application, create a new application)

The general overview will try to illustrate the features of the *Application Builder*, whereas the tutorials will try to guide you through with the help of examples.

6.1. Application Builder Overview

The Application Builder is an application designed to create IVR Applications. IVR Applications are created by composing Application Controls.

Application Controls are categorized in the so called palette within different groups. Main goal of this was to separate controls for UC from all other controls to let users see their dependency to UC, or, vice versa, let them know which controls can be used without UC/Symphonia Framework in a classical IVR scenario.

Like UC the ACD controls are separated, too – but these ones are visible only in case they are enabled in the properties of an application (which is not default as it is for IVR and UC controls) and do need an OpenScape Contact center configured in the Media Server ACD provider for their execution.

Furthermore functionality for call flow design can be found in the palette: call flow direction, operators and potentially custom controls (controls which are imported with or for an application as seamless extension).

6.1.1. General Controls

- **Start:** Every call flow must start somewhere.
- **End:** Endpoint of an application. Optional.
- **Assign:** Assign values to variables, including conditions (Rule editor).
- **Compare:** Make a decision based and variables and rules. Allows recursion.
- **System Info:** Provides system data to be used in the application (current date, time, etc).
- **Web Service:** Provides access to RESTful Web Services.
- **Database Write:** Provides write access to databases (JDBC driver required).
- **Database Read:** Provides read access to databases (JDBC driver required).
- **Time Profile:** Allows configuring time profiles. Each time profile corresponds to an exit path of the control.
- **Delay:** Waits for a configurable time.

6.1.2. IVR Controls

- **Prompt:** Plays a prompt.

- **Backing Prompt:** Plays a prompt in a loop until the next prompt is played. Allows semi-parallel execution.
- **Dtmf Input:** Collect Dtmf Input provided by a user.
- **Dtmf Menu:** Provides a DTMF Menu. Every DTMF key creates a exit path of the control.
- **Dtmf Selection:** Advanced Dtmf Menu. Menus are created automatically based on list elements.
- **Language:** Switch the language of an application.
- **Transfer:** Transfer the user to another extension (Blind transfer, Consultation (Supervised) Transfer).
- **Disconnect:** Disconnects the caller.
- **Create Call:** Creates an outbound call in a workflow triggered by an event.
- **Connect Call:** Connects a call which was started with Early Media or alerting to avoid charging the user.
- **Deflect Call:** Deflects to a target or rejects a call in a workflow started by an alerting call.
- **Record:** Record what the users says.
- **Transition:** Makes a transition to another application.
- **NLU:** Allows creating flexible and complex voice controlled applications.
- **Speech menu:** Allows creating a voice controlled menu (more simple to use than the NLU control)
- **Speech input:** Allows getting a speech input from the caller ((more simple to use than the NLU control for directed speech applications)

6.1.3. UC Controls

- **Send:** Sends a voice message recorded with the Record control.
- **Message:** Deals with different types of messages in groupware's message store, e.g. playback, forward, delete, reply emails, set status of messages, reply to meeting requests
- **Authentication:** Authenticates a user on the OpenScape UC Server. Required for most UC controls.
- **Change PIN:** Changes the PIN (numeric password).
- **Presence:** Allows to get the presence state of a user or to set the presence state of a logged in user.
- **Contact Search:** Search for contacts in the OpenScape UC Contact database.
- **User Search:** Search for users in the OpenScape UC user database.
- **Conference:** Allows to get, create, start and join conferences.
- **CallJournal:** Retrieves the call journal of a logged in user.

Please note: UC Controls require an OpenScape UC Server, which is not part of the Media Server Starter Kit. Applications which utilize UC Controls must be deployed on an OpenScape UC Server of the same release.

6.1.4. ACD Controls

- **AcdInit:** Registers a call in contact center which is precondition for other ACD controls and starts OSCC's reporting.
- **AcdExit:** Unregister a call from the contact center.
- **AcdStart:** Starts the the assignment of a call to an agent in a waiting queue of the connected contact center.
- **AcdStop:** Removes the call from the waiting queue.
- **AcdCallback:** Creates a callback job in OSCC.
- **AcdContactData:** Attaches data (key value pairs) to a call before this is put in a queue and transferred to an agent.
- **AcdRoutingInfo:** Fetches information of the routing status from OSCC for announcements or optimization.
- **AcdQueueInfo:** Allows to get, create, start and join conferences.
- **AcdCallInfo:** Retrieves the call journal of a logged in user

Please note: Executing ACD Controls require an OpenScape Contact Center Server v8 R1 or R2, which is not part of the Media Server Starter Kit. The Starter Kit does not have a UI to configure such connection – in the productive OpenScape Media Server this is done using the CMP for the Media Server ACD provider.

6.1.5. Control Compositions

Control Compositions do consist of other controls. They can be compared with subroutines which can be reused easily. Drawn from the palette to the canvas (work area) they look like other controls.
The Starter Kit comes with 2 examples:

- **Change numeric password:** Change the PIN of UC user
- **Logon:** does allow a UC user to logon – this includes the change of the user's PIN

If you edit a composition you find specific start and end controls (Composition Entry, Composition Exit) on the palette.

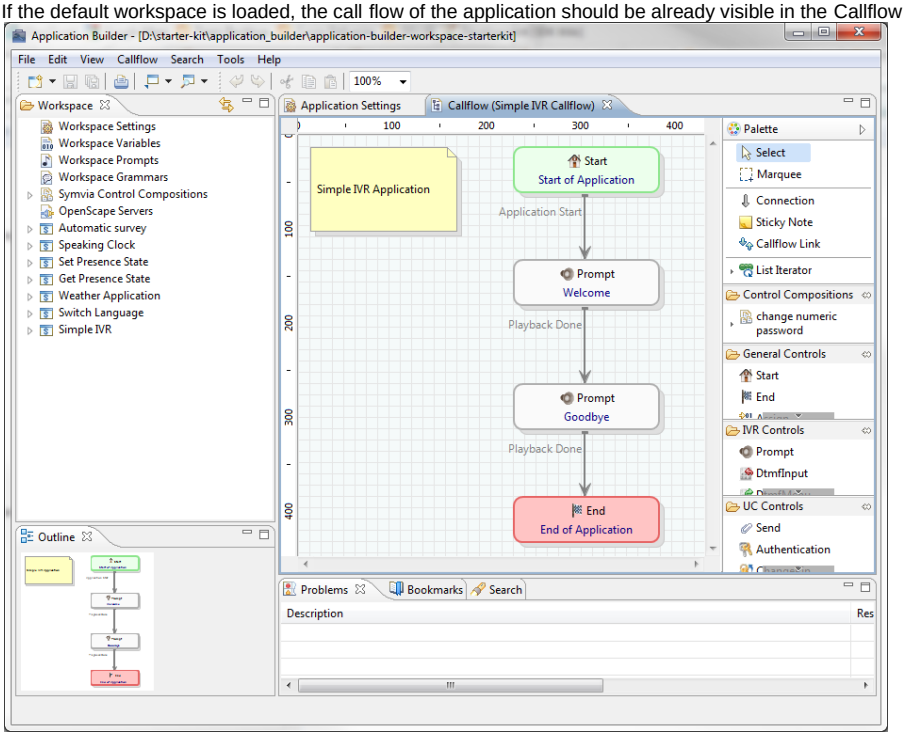
6.1.6. Lists, Operators and Flows

- **List Iterator:** A List Iterator control serves for running through step by step and to execute actions programmed in a subflow for each of the variables.
  The List Iterator is defined in a specifc subflow with the specific controls Subflow Entry, Subflow Exit, Subflow Loop
- **List Sorter:** Sort a variable list with sort order and the sort criterion.
- **List Modifier:** Allows to add or delete elements to or from a variable list.
- **String Operator:** This control allows to modify strings with 15 operations to select from
- **Time Operator:** Allows modification of time variables
- **Date Operator:** Allows modification of date variables
- **Parallel Flow:** Allows to split a callflow in 2 which can be separately modeled. Example: a callee does have to accept a calling parties' transfer to him

6.1.7. Test and Deployment with Sample Application 'Simple IVR'

This section describes how to test and deploy the sample application "SimpleIVR", which is included in the Application Builders sample applications bundle. The "Simple IVR" application does not require a TTS configured on the Media Server, since all prompts are pre-recorded.

If the default workspace is loaded, the call flow of the application should be already visible in the Callflow editor:



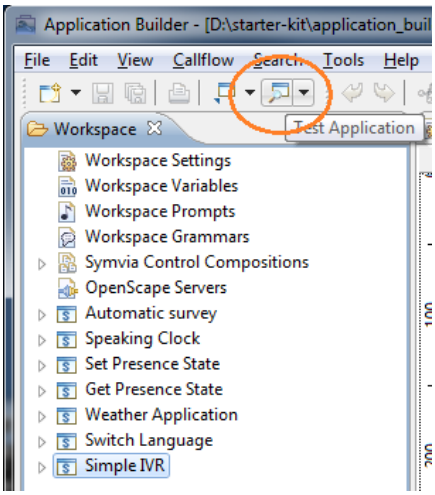
The application consists of two Prompt Application Controls (Welcome & Goodbye), which in this case, play a pre-recorded prompt to the user. The prompt is pre-recorded so that the application can run without a TTS.

6.1.8. Test

The Test feature within the Application Builder allows simulating how the application will run on the server. Therefore support for the simulation mode is a core feature of each Application Control.

The simulation mode allows running the applications offline, without the need for a TTS or ASR, a UC server or even a Media Server. The complete application logic can be modeled and tested *before* deploying the application to the Media Server. Even error conditions can be tested, by specifying the corresponding exit path of an Application Control. For example you may explicitly select that a logon fails, in order to test applications call flow under that condition.

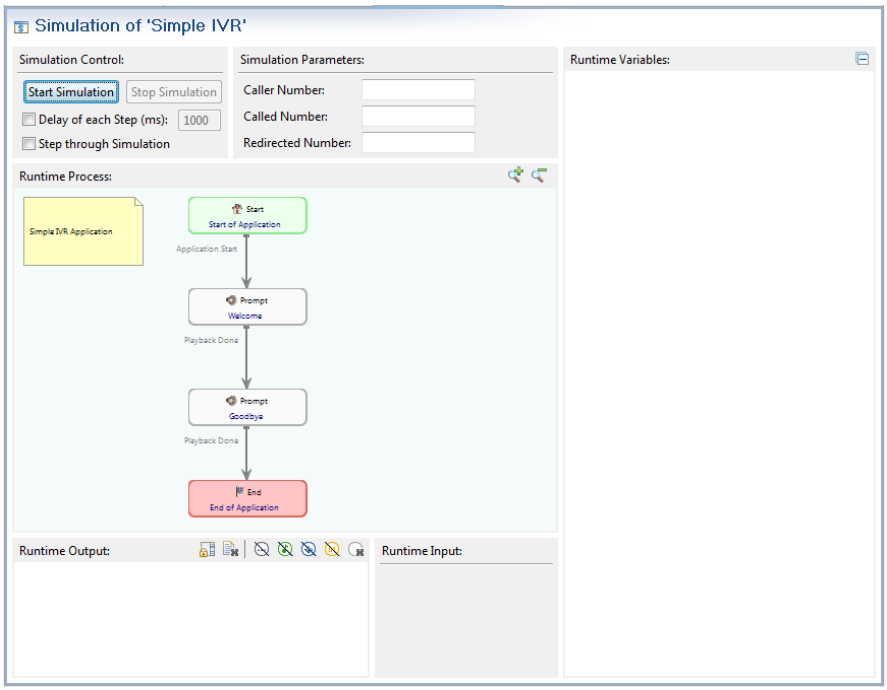
In order to test an application, select the Application project you would like to simulate (“Simple IVR”), and click on the Test Button on the menu bar:



Alternatively you can the select “Test Application…” from the context menu when right clicking the Application Project or by clicking on arrow beside the test button in the menu bar. You can also select the project to test from the drop-down list.

Please note that you can only test applications which are syntactically correct. Otherwise the “Test Application…” functionality is disabled for this application. In addition the errors are displayed in the Problem view.

The Application Simulator is separated in different areas:



6.1.8.1. Simulation control

With the Simulation control you can start the simulation and control how the simulation will be executed.
Without providing any further options, the application will run until user input is required or, if a control provides more than one exit path, the exit path for the simulation needs to be provided (default mode).

Alternatively you can choose between the following options:

- Delay of each Step: The simulation will run as in the default mode, but with a configurable pause after each application control executed
- Step through Simulation: Manually run through the application. Every step needs to be explicit triggered. Please note that even an assignment to a variable is a single step.

6.1.8.2. Simulation Parameters

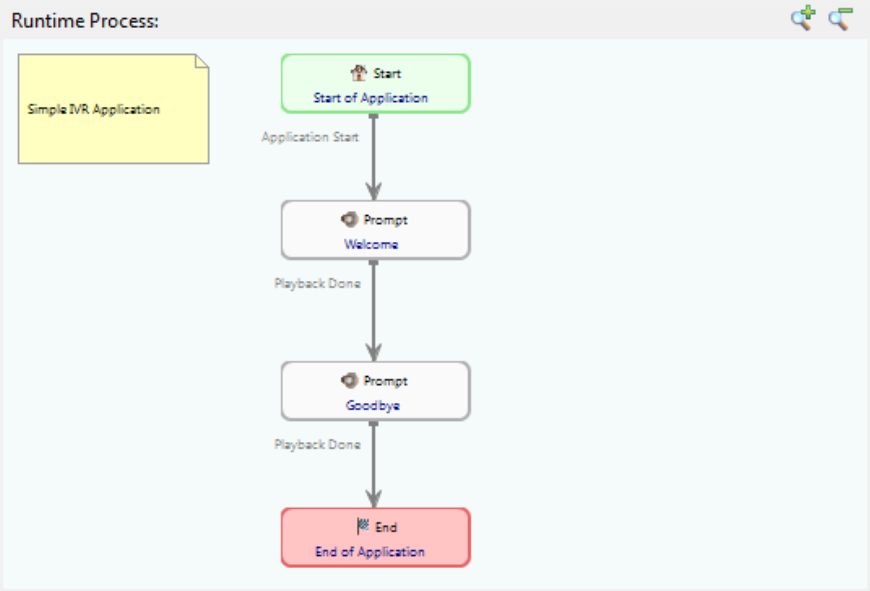
Every IVR application can access the following set of runtime parameters:

- Caller Number: The number of the originating phone device (if available)
- Called Number: The number the caller dialed
- Redirected Number: The redirected number information

If the application provides business logic based on these parameters, the data for the simulation can be applied in this section.

6.1.8.3. Runtime Process

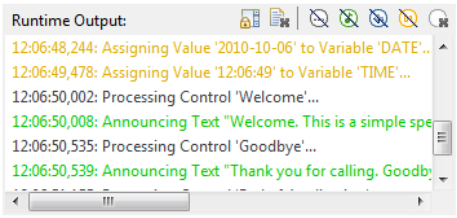
In this section the currently active call flow is displayed:



In case of multiple call flows per application or of sub-call flows in control composites, the 'active' call flow is loaded. On the loaded call flow the Application Control which is currently processed is highlighted.

6.1.8.4. Runtime Output

The runtime output section shows logging information from the application, including information which prompts are played, etc.:



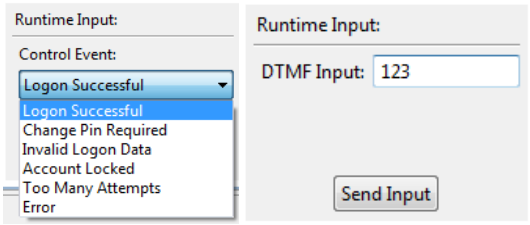
The following filter options are available:

- Status Messages
- Prompt Messages
- Input Messages
- Variable Messages

Typically, if you are only interested in the application input and output, you would hide Status Messages and Variable Messages.

6.1.8.5. Runtime Input

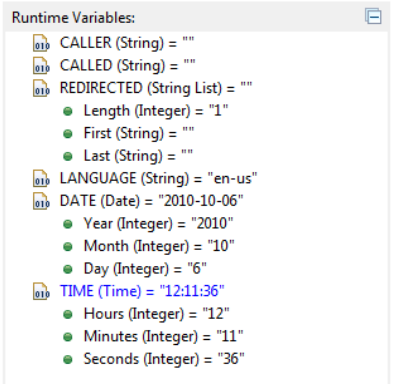
The runtime input section allows providing input to the application during the simulation. You can either provide input as a normal user would do, like providing a PIN Number, or you can select the exit path of an Application Control. The later can be used to explicitly simulate the behavior of the application in case of an error.



Please note: the screenshots shown above are taken from another example application.

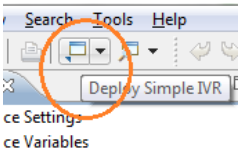
6.1.8.6. Runtime Variables

The runtime variables section shows every variable which is used in the application. Variables are shown when they are created during runtime - when values were assigned to the variables.



6.2. Deployment

In order to deploy an application, select the Application project you would like to deploy (“Simple IVR”), and click on the Deploy Button on the menu bar:



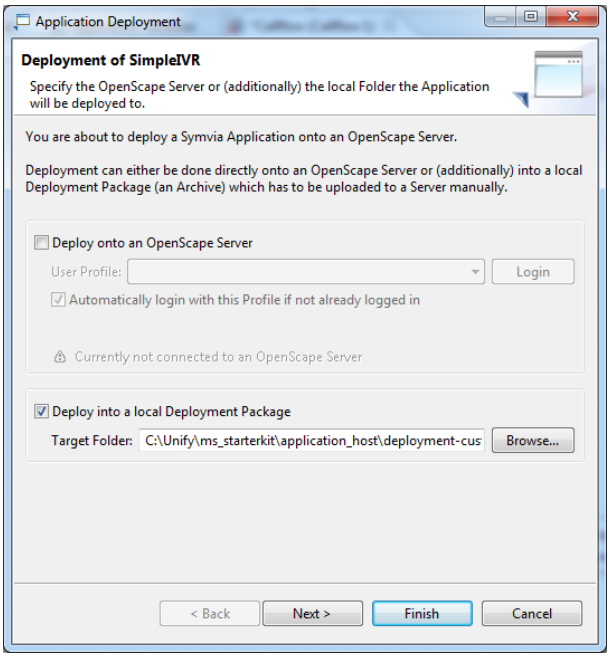
Alternatively you can the select “Deploy Application…” from the context menu when right clicking the Application Project or by clicking on the arrow beside the deploy button in the menu bar. You can select the project to test from the drop-down list.

Please note that you can only deploy applications which are syntactically correct. Otherwise the “Deploy Application…” functionality is disabled for this application. In addition the errors are displayed in the problem view.

The deployment process, which is the process of creating a Media Server deployment package, is supported by the deployment wizard.

6.2.1.1. Deployment Wizard

On this first page of the Deployment wizard is the folder specified, in which the Media Server deployment package is created:

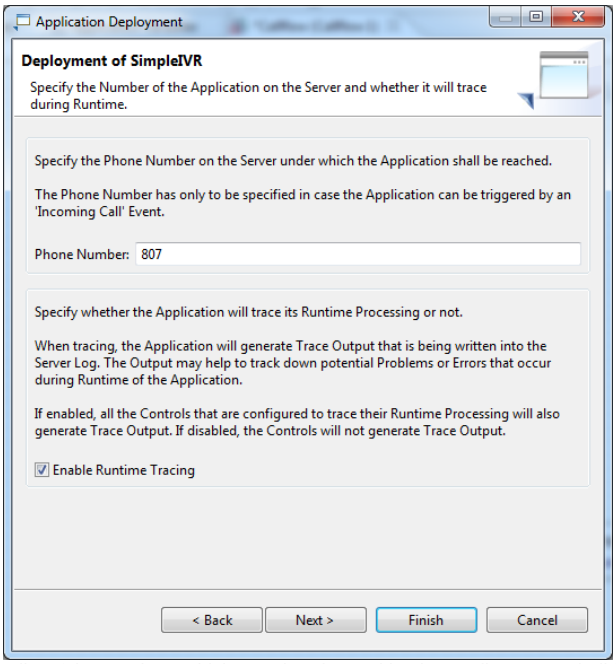


The sample applications of the Starter Kit are preconfigured, so the output directory of the deployment process is, at the same time, the deployment folder of the Media Server. This way the applications are available on the Media Server once the deployment process has been finished. This is typically a process of few seconds and can be checked on demand in the Media Server command line console as described later in this document.

In case of deployment to an OpenScape UC Server it is possible to either upload a deployment package as created above or, a direct deployment can be used. The latter needs a connection profile which can be created at a workspace's OpenScape Server settings.

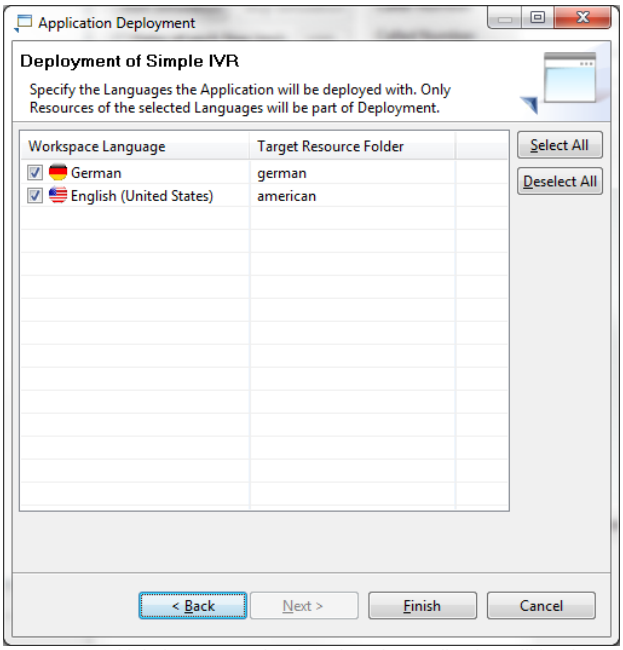
In the next step, the phone number can be configured which should be used to call the application after it has been deployed to the Media Server. Since the Starter Kit contains a stand-alone Media Server without a management UI you need to keep track of the applications deployed on the Media Server.[1]

[1] In case you need to start from scratch or change e.g. a language bound to a given extension and application you can delate all or the specific application from ..\Unify\ms_starterkit\application_host\deployment-custom



A list of the numbers which are already used by the Media Servers build-in applications and by the sample applications can be found in the Appendix.
On this second page the application can be configured to create trace information additionally.

On the Wizard's next page the language resources which are deployed on the server can be configured. For example you may decide only to include the English resources even if the Application contains resources for multiple languages:

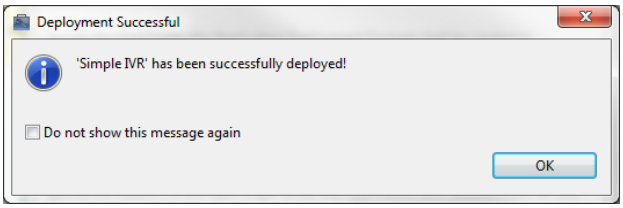


In case of multiple languages the default of the application will be the active one for new deployments.

The last step of the wizard allows determining if the complete application with all resources shall be exported.

Kommentar [JHK1]: Was ist damit? Obsolete: The last parameter on this page is the Server Address. This parameter needs to be provided if application requires access to an OpenScape UC Server, for example if UC Controls are used within the application (Presence, Logon, etc.). - This parameter will go away in later versions.

Click on 'Finish'. This creates the Media Server deployment package.



Changes made in the deployment wizard are saved, so that you can directly click on 'finish' if you plan to deploy a new version of the application with the same deployment settings again.

After the Application Builder has deployed the Application to the Media Server Deployment folder, the process of activating the application within the Media Server takes some time.

Before calling the application, check the log for lines like:

```
13:32:35,250 INFO       cycos.connectivity.terminal.impl.TerminalProviderImpl [] Adding
terminal 'SimpleIVR', applet='application:/SimpleIVR'

13:32:35,263 INFO       cycos.media.host.tomcat.WebContextComponent [] Deploying Tomcat
WebContext:[SimpleIVR] at:[file:/D:/starter-kit/application_host/work/SimpleIVR-
1.1.1/webcontext]

13:32:35,326 INFO       cycos.media.host.container.DeployedComponentImpl [] Compo-
nent:[SimpleIVR-webcontext] initialized in [63] ms!
```

6.2.2. Calling the Application

Start the VoIP softphone. Make sure that it is connected to the Media Server. Check the Prerequisites for more details in regard to the softphone configuration.

Dial the number `807` to call the application. Besides hearing the two prompts of the application, the Media Server log should show the call as well:

```
13:49:21,309 INFO       connectivity.mcc.sip.impl.MccConnectionManagerImpl [sip.0] Incom-
ing connection from '<sip:Wolfgang%20Schiffer@10.1.32.129>' to '"807"
<sip:807@10.1.32.129>', offered to 1 listeners

13:49:21,335 INFO       af.ivr.ms.applet.CannedApplication [] Loading application proper-
ties [file:/D:/starter-kit/application_host/work/SimpleIVR-
1.1.1/conf/symvia.properties.xml]

13:49:21,511 INFO       cycos.media.host.container.DeployedComponentImpl [] Compo-
nent:[SimpleIVR] initialized in [179] ms!

13:49:21,524 INFO       media.host.binders.terminalbinder.MCCListener [sip.0] Call
'sip.0' will be handled by application 'application:/SimpleIVR', session 'session:sid.1',
terminal 'SimpleIVR'

13:49:21,642 INFO       cycos.media.framework.native.stdout [] 2010-10-06 13:49:21,642
INFO  mfw.event.EventThread - notifiying event-consumer(8) took very long -> '52' ms. It
seems, that the system is extremly under load; stuttering is possible! (notified '1' con-
sumer; in list '1'  overloadCounter '5'  threadOverloaded '0'  logCounter '0')

13:49:21,757 INFO       connectivity.mcc.sip.impl.MccConnectionManagerImpl [sip.0] Incom-
ing connection established. CallId='Y2M2MTJlYzRmN2Y2MmQ3ZjkyOTJhZmJmYTQwZGNkMzI.'

13:49:22,093 INFO       cycos.mps.af.api.AbstractRuntime [sid.1 CannedAp-
plet.CannedDialog.0] WF Session created for application
[com.cycos.mps.af.symvia.ivr.application.SymviaSpeechApplication]: [sid1347004f-7346-
4b69-aa78-bf8806195f72]
```

```
13:49:22,097 INFO       api.control.general.impl.AbstractSymviaControl [sid1347004f-7346-
4b69-aa78-bf8806195f72] Application [Simple IVR] called

13:49:28,634 INFO       cycos.mps.af.api.AbstractSession [sid1347004f-7346-4b69-aa78-
bf8806195f72] WF session closed:[DefaultSession(sid1347004f-7346-4b69-aa78-
bf8806195f72;CLOSED;currentControl:null;application:null;isExecuting:true;isJoined:false)
]

13:49:28,754 INFO       connectivity.mcc.sip.impl.MccConnectionManagerImpl [sip.0] Dis-
connected
```

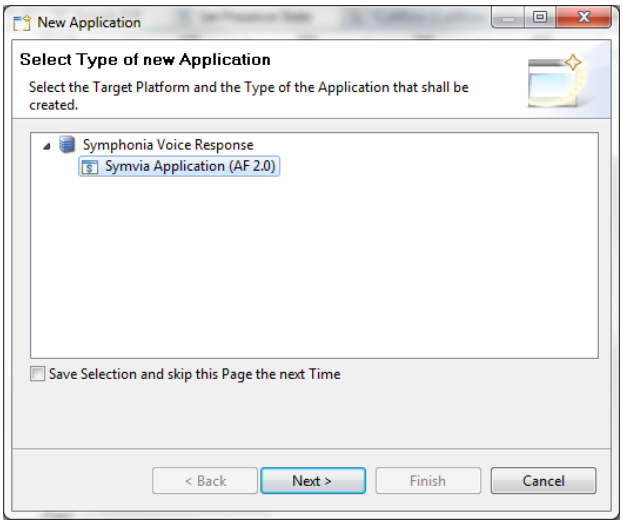
6.3. Creating a new Application

In this section we will create a new application from the scratch. In order to keep the tutorial reasonable short the application will be rather simple, but of course it can be easily extended. For more complex applications consider to have a look at the available sample applications or control compositions.

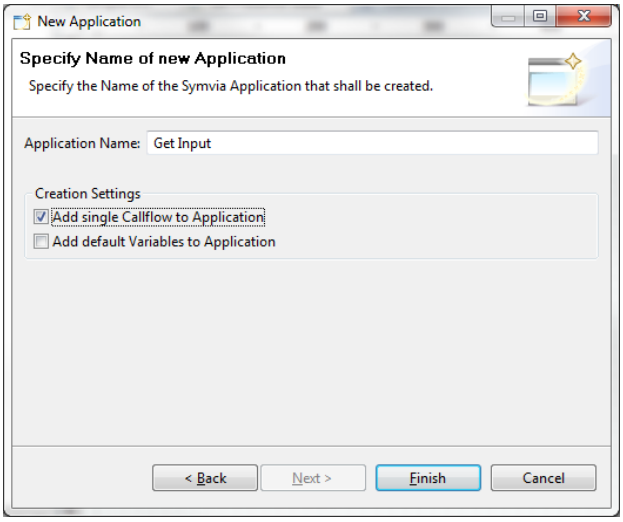
The example application will collect DTMF input from a user and in a second step repeat the input. This application requires a TTS.

6.3.1. Creating the Project

To create a new application, click on File→New→Application. In the dialog select 'Symvia Application (AF 2.0)' as the type of the new application and click on 'Next'.

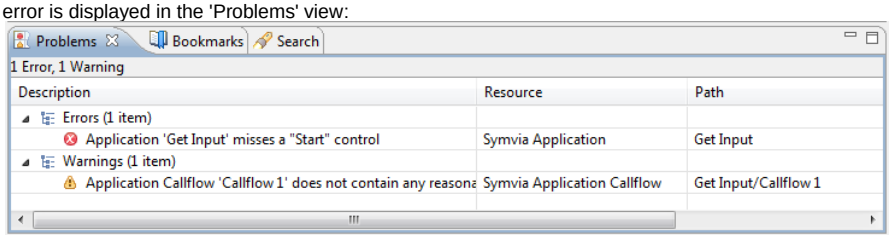


In the next dialog provide a name for the new application (e.g. “Get Input”). Select “Add single Call flow to Application”, since this will save us the effort to create the initial call flow later on. We don't need the default variables, so checking the option “Add default Variables to Application” is not necessary. Both settings can be done later on as well.



Click on button 'Finish'.

Please note that the project is marked with a red cross in the workspace, which means that the project contains errors. The error is displayed in the 'Problems' view:

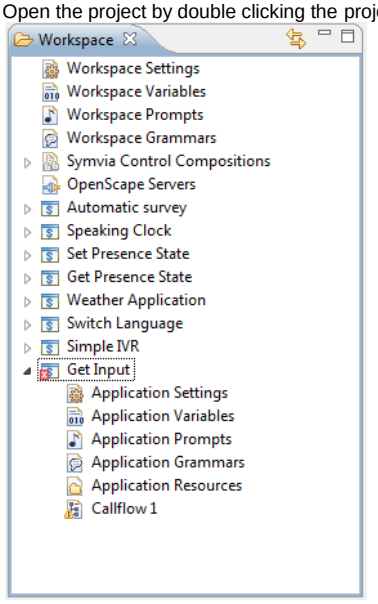


In this case it is reported that the applications call flow does not contain a Start control (reported as error) and that there is no other reasonable content in the call flow (reported as warning).

6.3.2. Creating the Call Flow

The next step is to model the applications call flow by putting the Application Controls we need on the Call flow Editor.

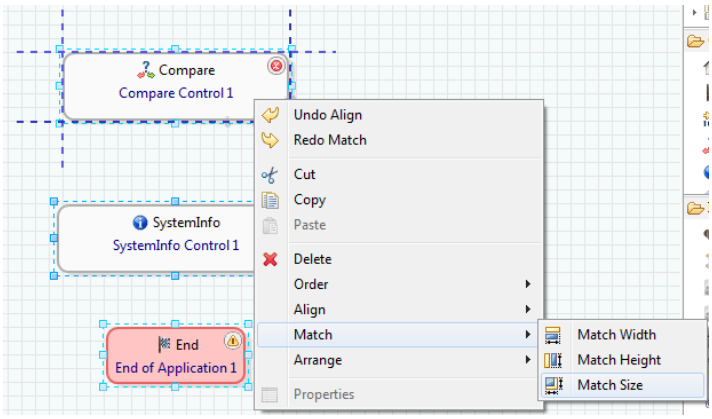
Open the project by double clicking the project in the Workspace and double click the Callflow (→“Callflow1”):



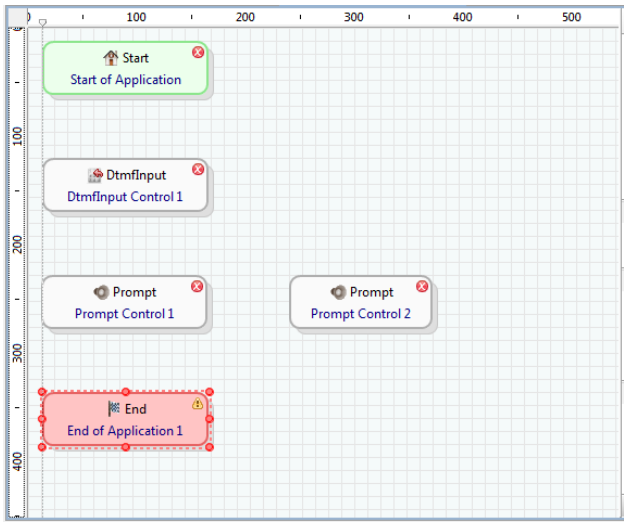
For this application we need the following controls:

- DtmfInput Control
- Prompt Control, to repeat the user input
- Prompt Control, if a problem occurs in the DtmfInput Control
- Start/End Control, which are required for our application

In order to make the Call flow look 'smoother', it is recommended align controls width. Both size alignment and alignment of positions can be done in the menu opening with right mouse click:



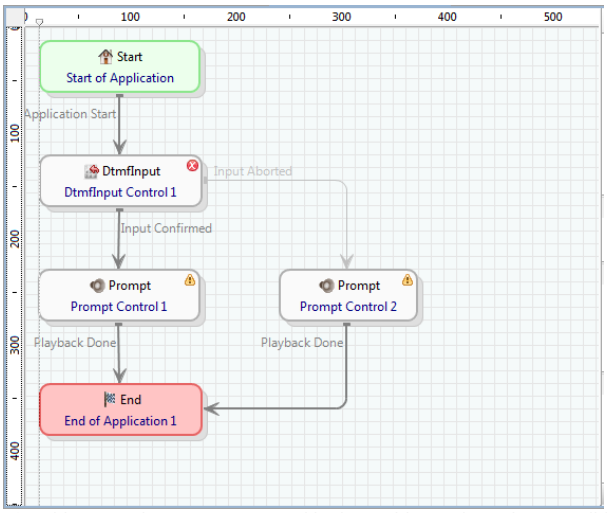
The result could look like this:



Now the wiring of the Application Controls needs to be done, so that we have the core framework of the application finished.

- The DtmfInput control provides two exits which must be connected:
- Input Confirmed: Default exit, the input was successfully received
- Input Aborted: Input operation was aborted by the user

This can be done by using the anchor points of the controls on the canvas or by selecting ‘connection’ in the Palette and clicking on the controls to link afterwards.

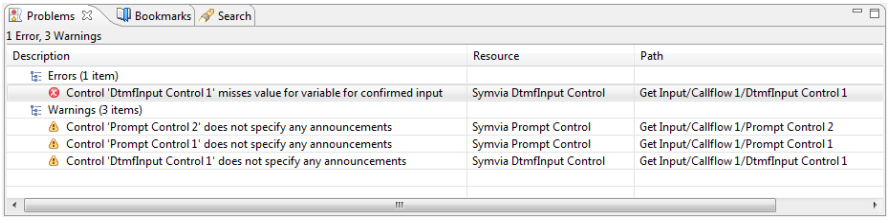


After this, some issues are reported in the problems view, since we did not configure the Application Controls properly. This will be done in the next steps.

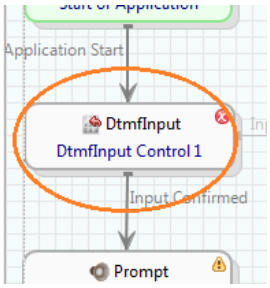
6.3.3. Configuration of the Application Controls

6.3.3.1. Configuration of the DtmfInput Control

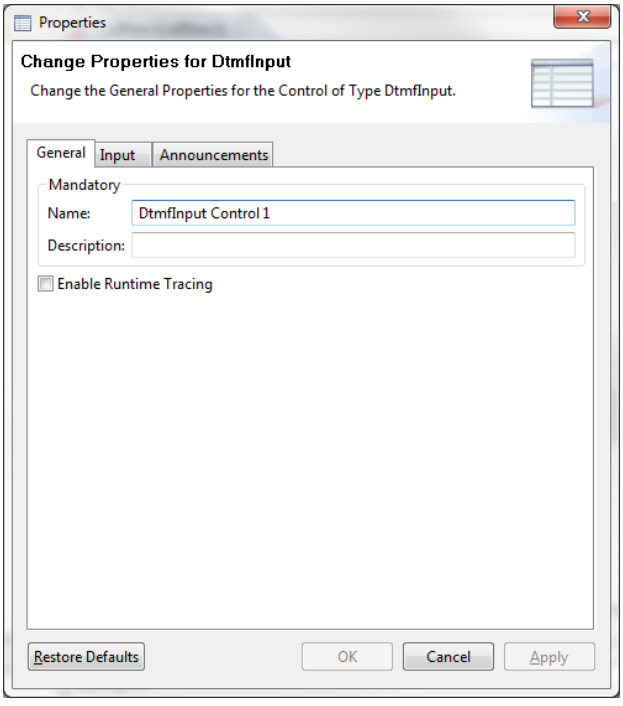
In the problems view this control reports that there is no variable define for confirmed input.



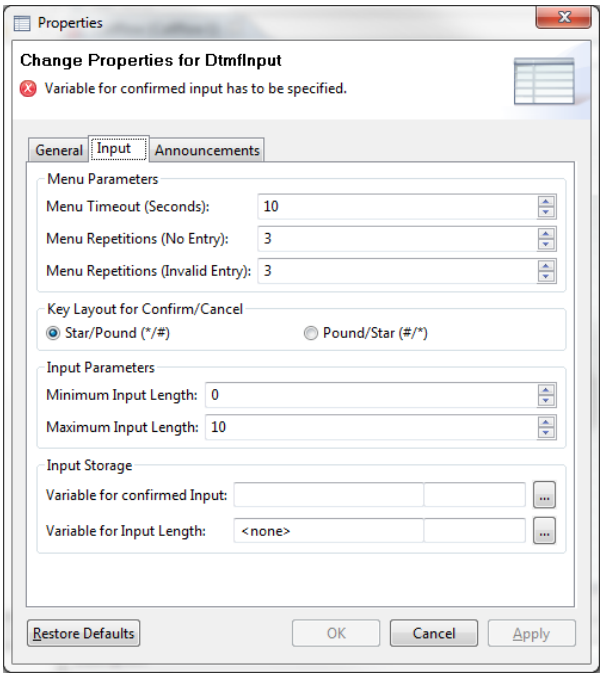
Double click the entry in the problems view or double click the control on the Callflow editor to open the control's configuration dialog:



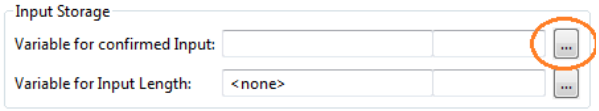
On the 'General' tab you can change the name of the control which appears in the Callflow Editor and provide a description.
The option 'Enable Runtime Tracing' will allow collecting trace data for the control.
These options are available for every control and it is recommended to provide at least a reasonable name.
The description will be used for the tool tip which is displayed in the Callflow Editor.



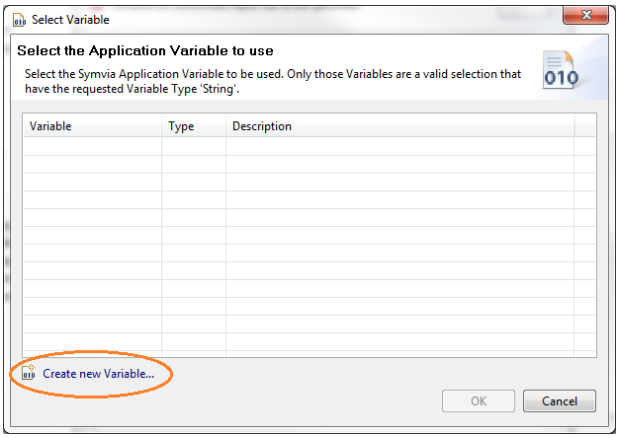
On the 'Input' tab most of the settings don't need modification with the exception of the 'Input Storage' section.
Here we need to specify a variable, which we will use to store the user input:



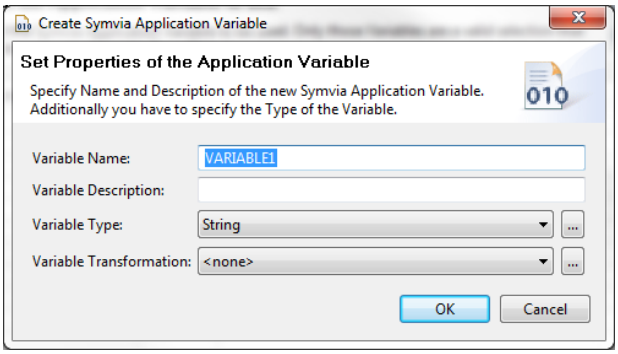
Click on the '...' button to select a variable from the variable list.



Since we did not create a variable so far (can be done in the 'Application Variables' Editor) the variable list is empty. Click on 'Create new Variable...' to create a new variable.

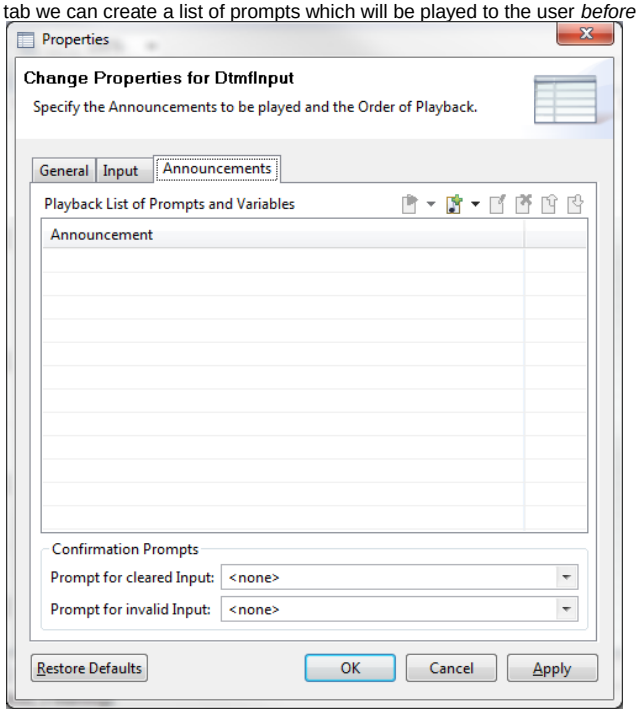


In the dialog provide a suitable name (e.g. “USERINPUT”) and a description. The type “String” is correct. A Variable Transformation is not needed here.



Click on button ‘OK’.

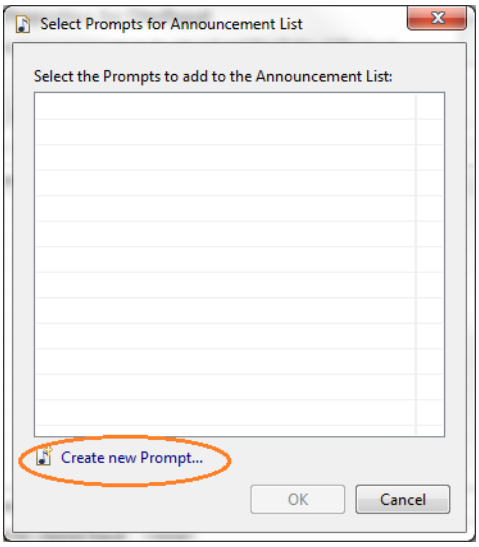
In the “Select Variable” dialog the new variable is automatically selected, so we can click on OK. On the 'Announcement' tab we can create a list of prompts which will be played to the user *before* the Dtmf input is collected.



Click on the 'new prompt' button:

This will open the list of available prompts. Since we did not create a prompt so far (can be done in the 'Application Prompt' Editor) the variable list is empty.

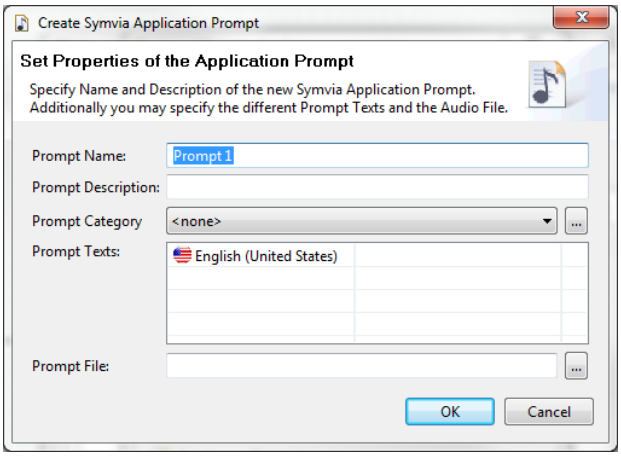
Click on 'Create new Prompt...' to create a new prompt:



In the new dialog ‘Create Symvia Application Prompt’ provide the following values:

- Prompt Name: get user input
- Prompt Description: Prompt the user to provide DTMF input
- Prompt Text: Please enter up to 10 digits with the DTMF keypad and finish your input with the STAR key

Of course other values will work as well.



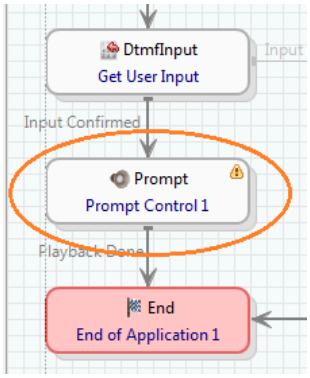
Click on the button ‘OK’.

In the “Select Prompts for Announcement List” dialog the new prompt is automatically selected, so we can click on OK.

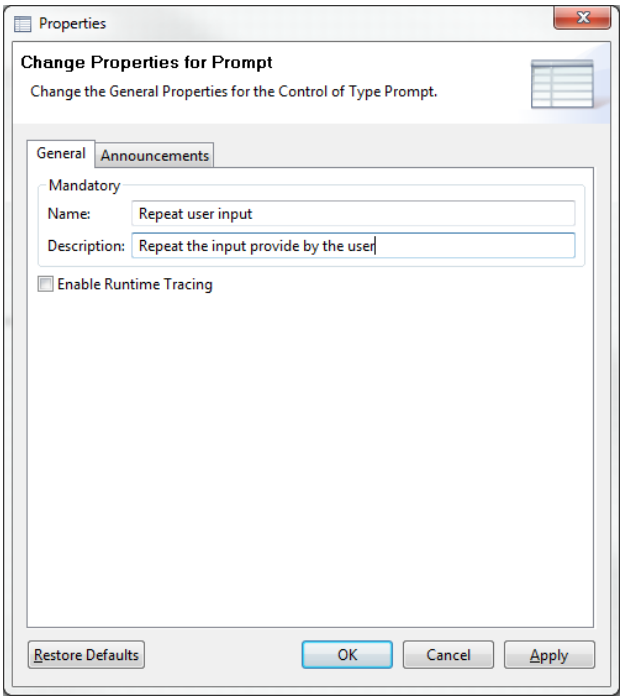
The configuration of the DtmfInput control is now finished. Click on OK and save the callflow (Ctrl-s). The error message should disappear from the Problems view.

6.3.3.2. Configuration of the Prompt Control

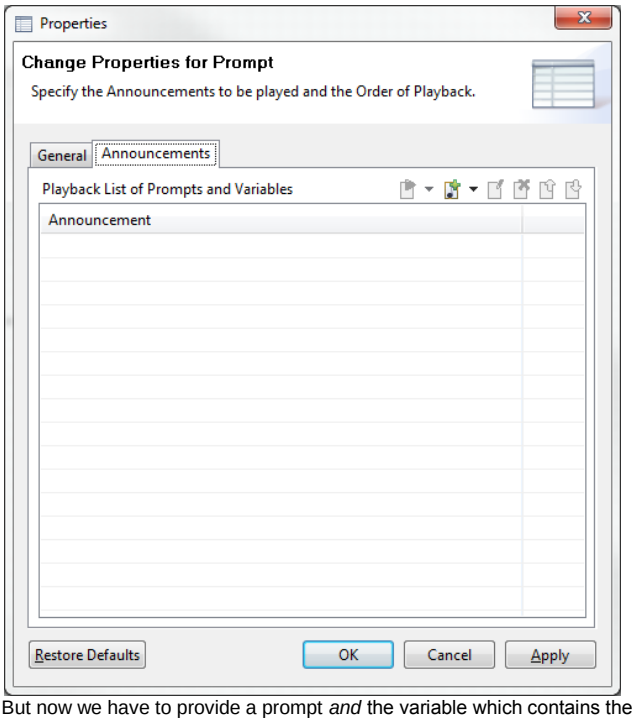
The next prompt control we will configure is the Prompt Control which is connected to the 'Input Confirmed' exit of the DtmfInput Control.



Double Click the entry in the problems view or double click the Control on the Callflow editor to open the controls configuration dialog. On the 'General' tab you can change the name of the control which appears in the Callflow Editor and provide a description:



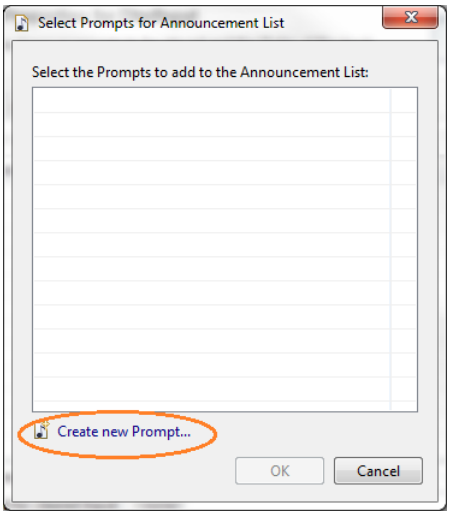
The 'Announcement' tab is the same as in the DtmfInput Control.

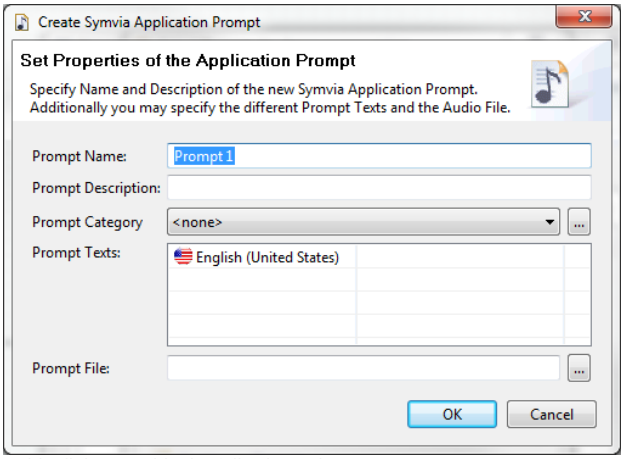


But now we have to provide a prompt *and* the variable which contains the stored user input (“USERINPUT”).

First we create a new prompt – therefore click on the 'new prompt' button:

This will open the new list of available prompts. We need a new prompt so click on 'Create new Prompt...' to create a new prompt:



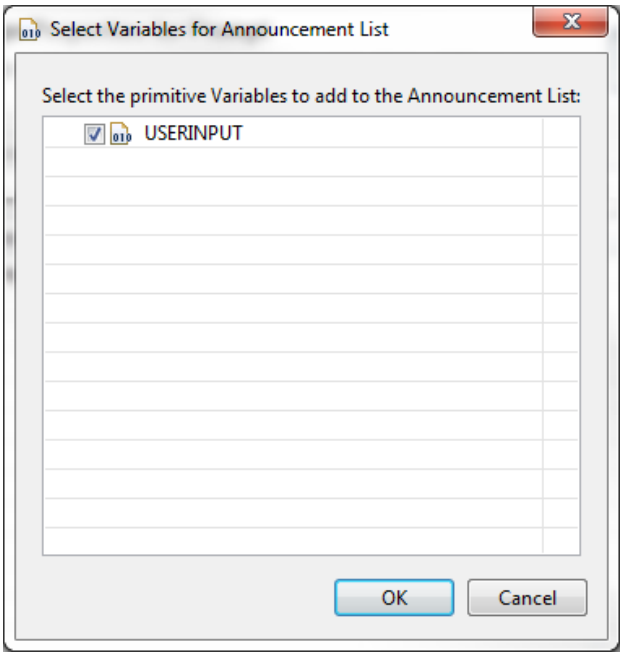


In this dialog provide the following values:

- Prompt Name: 'repeat user input'
- Prompt Description: 'Repeat the input provided by the user'
- Prompt Text: 'You entered:'

Of course other values will work as well. Click on button ‘OK’.

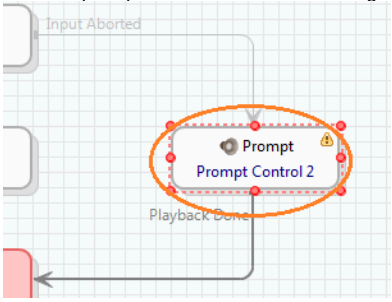
Next we select the variable which contains the user input: Click on the arrow next to the 'new prompt' button and select 'Add Variable…'. Select the 'USERINPUT' variable from the list and click on OK:



Click on OK again to close the prompt configuration and save the call flow (Ctrl-s). One of warnings should disappear from the Problems view.

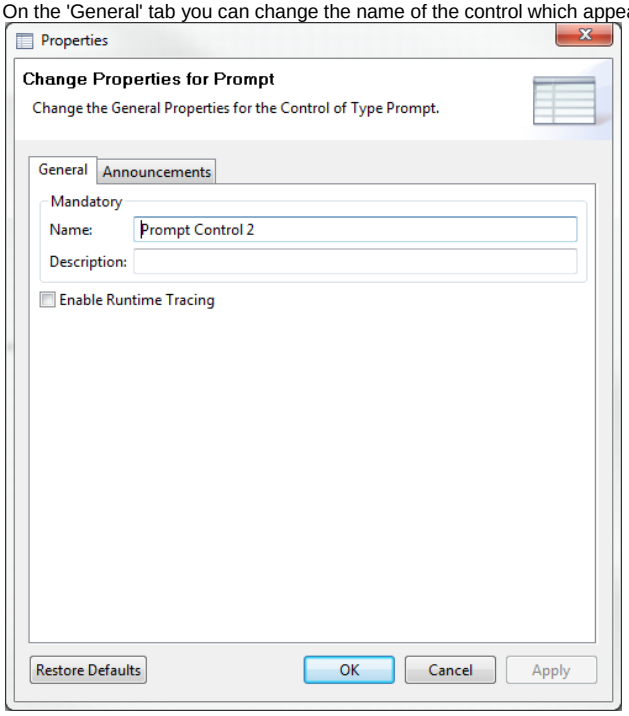
6.3.3.3. Configuration of the second Prompt Control

The next prompt control which needs configuration is connected to the 'Input Aborted' exit of the DtmfInput control.

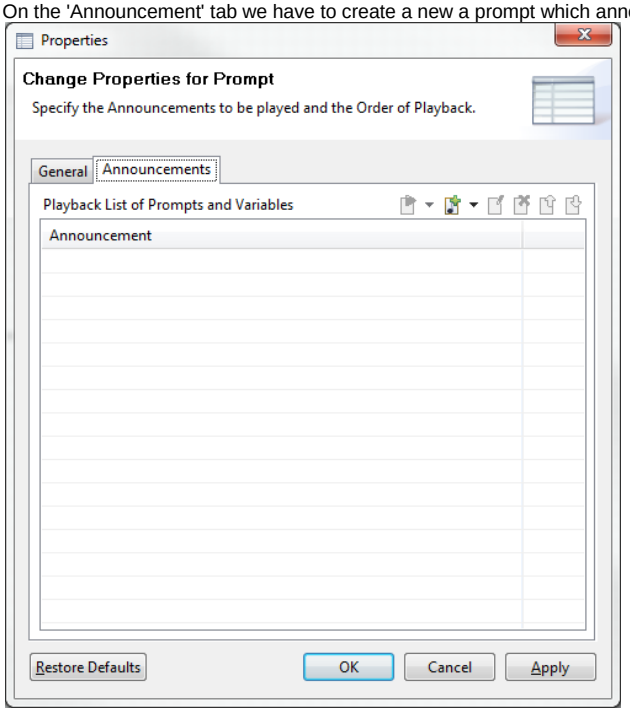


Double Click the entry in the problems view or double click the Control on the Callflow editor to open the controls configuration dialog.

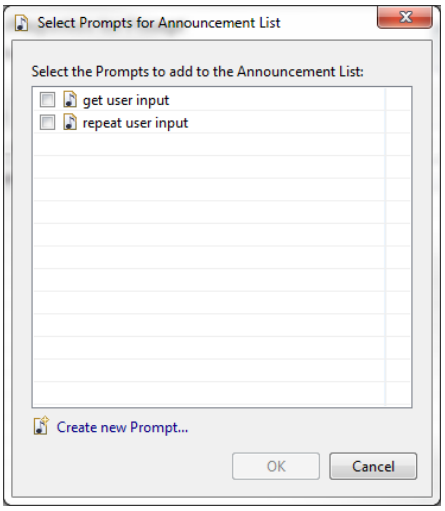
On the 'General' tab you can change the name of the control which appears in the Callflow Editor and provide a description.



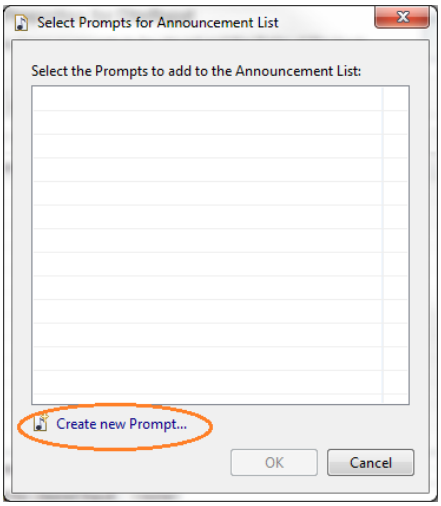
On the 'Announcement' tab we have to create a new a prompt which announces that the user cancelled the operation:



Click the 'new prompt' button. This will open the new list of available prompts:

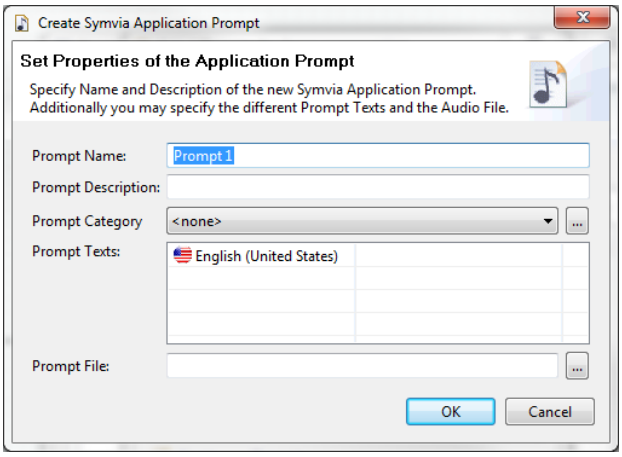


Click on 'Create new Prompt...' to create a new prompt.



In the next dialog provide the following values:

- Prompt Name: 'Operation cancelled'
- Prompt Description: 'The operation was cancelled by the user'
- Prompt Text: 'The operation was cancelled. Thank you for calling. Goodbye.'



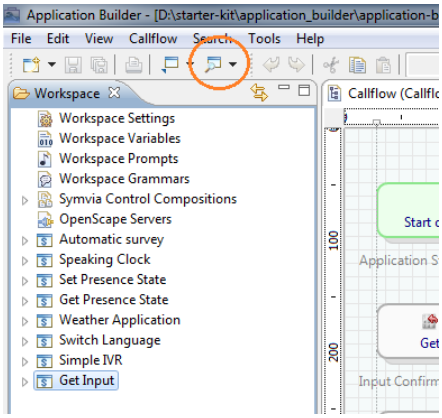
Click button ‘OK’.

Click on ‘OK’ to close the prompt configuration and save the call flow (Ctrl-s).

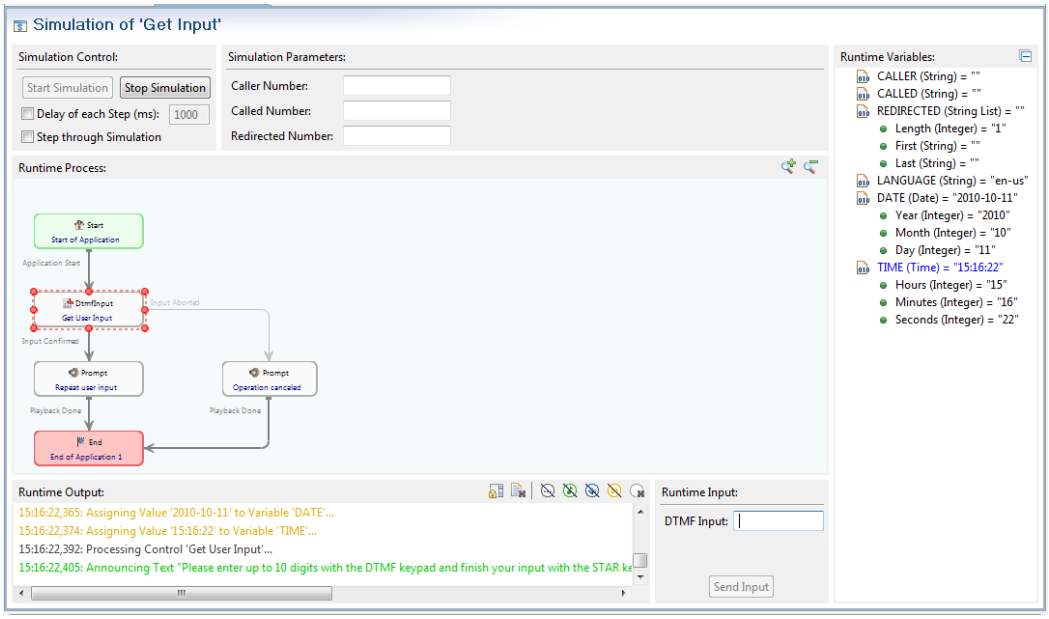
All warnings should now be gone from the Problems view.

6.4. Test the Application via Simulation

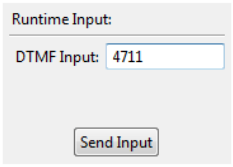
In order to test the new application, select the new application in the workspace and click on the Test Button on the menu bar:



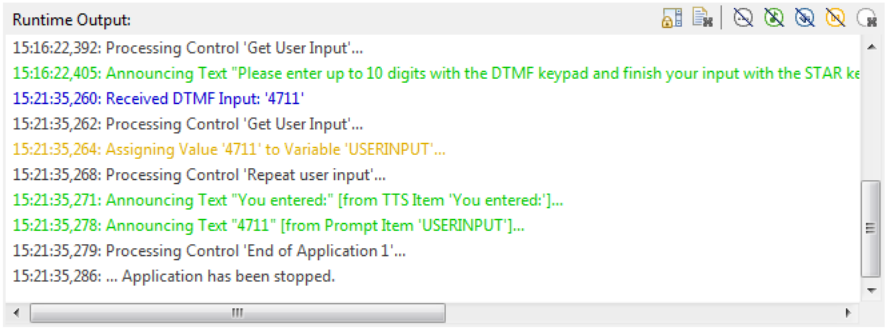
Start the simulation in default mode. The application should run to the DtmfInput Control and request input from the user.



Provide some input in the DTMF Input box in the Runtime Input section, and click on 'Send Input'.



In the Runtime Output section of the Simulator you should see that the application repeats the user input.



For more details about the Simulation capabilities of the Application Builder please refer to chapter ‘

Control Compositions

Control Compositions do consist of other controls. They can be compared with subroutines which can be reused easily. Drawn from the palette to the canvas (work area) they look like other controls.
The Starter Kit comes with 2 examples:

- **Change numeric password:** Change the PIN of UC user
- **Logon:** does allow a UC user to logon – this includes the change of the user’s PIN

If you edit a composition you find specific start and end controls (Composition Entry, Composition Exit) on the palette.

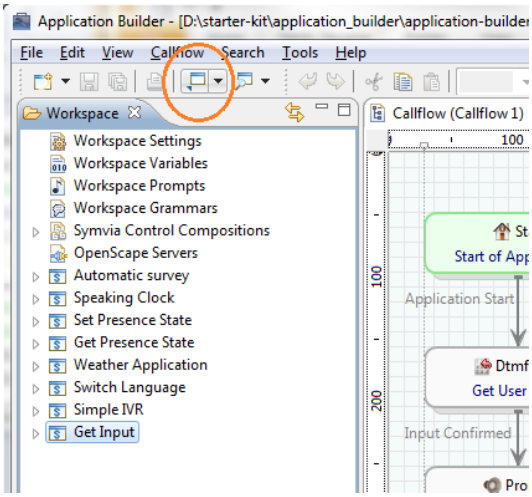
6.4.1. Lists, Operators and Flows

- **List Iterator:** A List Iterator control serves for running through step by step and to execute actions programmed in a subflow for each of the variables.
  The List Iterator is defined in a specifc subflow with the specific controls Subflow Entry, Subflow Exit, Subflow Loop
- **List Sorter:** Sort a variable list with sort order and the sort criterion.
- **List Modifier:** Allows to add or delete elements to or from a variable list.
- **String Operator:** This control allows to modify strings with 15 operations to select from
- **Time Operator:** Allows modification of time variables
- **Date Operator:** Allows modification of date variables
- **Parallel Flow:** Allows to split a callflow in 2 which can be separately modeled. Example: a callee does have to accept a calling parties' transfer to him

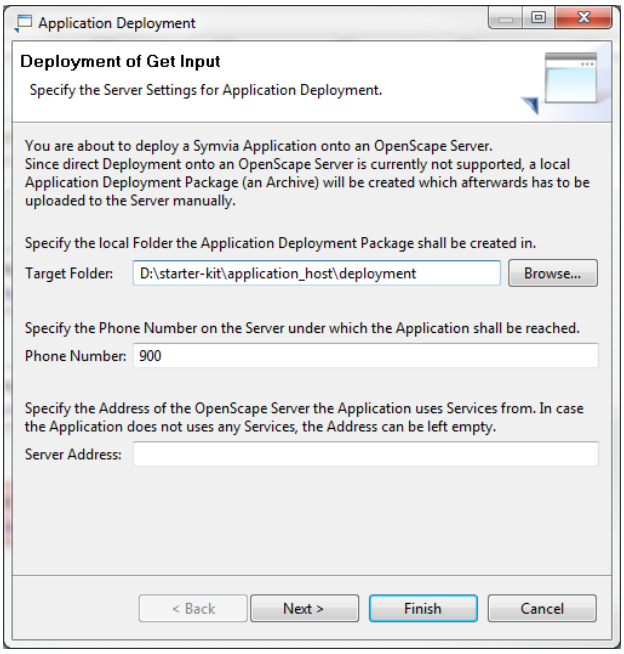
Test and Deployment with Sample Application 'Simple IVR'', sub chapter '*Test'.*

6.5. Deploy the Application with the Starter Kit

In order to deploy the new application, select the new application in the workspace and click on the Deploy Button on the menu bar.



In the first page of the deployment wizard as shown above provide the following information:



Folder in which the Media Server Deployment should be created:

For the Starter Kit scenario it is recommended that the selected folder is the same as the Media Server deployment folder: `starter-kit/application_host/deployment`

`application_host` = Top level `deployment` directory for all Media Server components

`deployment` = The Media Server deployment folder: Contains applications running on the Media Server

Telephone number, which is used to call the application:
In a standard Starter Kit, the number range 900 - 950 is not used. You may use any number from that range. A server address is not required.

Click on 'finish' to the create the deployment package with default settings.

For more details about the Deployment wizard please refer to chapter '

Control Compositions

Control Compositions do consist of other controls. They can be compared with subroutines which can be reused easily. Drawn from the palette to the canvas (work area) they look like other controls.
The Starter Kit comes with 2 examples:

- **Change numeric password:** Change the PIN of UC user
- **Logon:** does allow a UC user to logon – this includes the change of the user's PIN

If you edit a composition you find specific start and end controls (Composition Entry, Composition Exit) on the palette.

6.5.1. Lists, Operators and Flows

- **List Iterator:** A List Iterator control serves for running through step by step and to execute actions programmed in a subflow for each of the variables.
  The List Iterator is defined in a specifc subflow with the specific controls Subflow Entry, Subflow Exit, Subflow Loop
- **List Sorter:** Sort a variable list with sort order and the sort criterion.
- **List Modifier:** Allows to add or delete elements to or from a variable list.
- **String Operator:** This control allows to modify strings with 15 operations to select from
- **Time Operator:** Allows modification of time variables
- **Date Operator:** Allows modification of date variables
- **Parallel Flow:** Allows to split a callflow in 2 which can be separately modeled. Example: a callee does have to accept a calling parties' transfer to him

Test and Deployment with Sample Application 'Simple IVR'', sub chapter Deployment.

Check the Media Server console. Wait until you notice that the application is deployed on the server:

```
15:34:44,185 INFO       cycos.connectivity.terminal.impl.TerminalProviderImpl [] Adding
terminal 'GetInput', applet='application:/GetInput'

15:34:44,197 INFO       cycos.media.host.tomcat.WebContextComponent [] Deploying Tomcat
WebContext:[GetInput] at:[file:/D:/starter-kit/application_host/work/GetInput-
1.1.1/webcontext]

15:34:44,251 INFO       cycos.media.host.container.DeployedComponentImpl [] Compo-
nent:[GetInput-webcontext] initialized in [54] ms!
```

Start the VoIP softphone. Make sure that it is connected to the Media Server. Check the Prerequisites as described above for details in regard to the softphone configuration.

Dial the number **900 to call the application**. Besides hearing the application, the Media Server log should show the call as well:

```
15:55:33,159 INFO connectivity.mcc.sip.impl.MccConnectionManagerImpl [sip.0] Incoming
connection from '<sip:Wolfgang%20Schiffer@10.1.32.129>' to '"900" <sip:900@10.1.32.129>',
offered to 1 listeners

15:55:33,176 INFO af.ivr.ms.applet.CannedApplication [] Loading application properties
[file:/D:/starter-kit/application_host/work/GetInput-1.1.1/conf/symvia.properties.xml]

15:55:33,318 INFO cycos.media.host.container.DeployedComponentImpl [] Compo-
nent:[GetInput] initialized in [146] ms!

15:55:33,332 INFO media.host.binders.terminalbinder.MCCListener [sip.0] Call 'sip.0' will
be handled by application 'application:/GetInput', session 'session:sid.1', terminal
'GetInput'

15:55:33,536 INFO connectivity.mcc.sip.impl.MccConnectionManagerImpl [sip.0] Incoming
connection established. Call Id='YWE1MGE5ODBiMmQ5NTU2MGZkZjk4N2I1ZmI0ZTk1YWU.'

15:55:33,701 INFO cycos.mps.af.api.AbstractRuntime [sid.1 CannedApplet.CannedDialog.0] WF
Session created for application
[com.cycos.mps.af.symvia.ivr.application.SymviaSpeechApplication]: [sid0537e5ff-a9a8-
4a72-84a6-17c50aa94f1e]

15:55:33,705 INFO api.control.general.impl.AbstractSymviaControl [sid0537e5ff-a9a8-4a72-
84a6-17c50aa94f1e] Application [Get Input] called
```

```
15:55:43,195 INFO cycos.mps.af.api.AbstractSession [sid0537e5ff-a9a8-4a72-84a6-
17c50aa94f1e] WF session closed:[DefaultSession(sid0537e5ff-a9a8-4a72-84a6-
17c50aa94f1e;CLOSED;currentControl:null;application:null;isExecuting:true;isJoined:false)
]

15:55:43,317 INFO connectivity.mcc.sip.impl.MccConnectionManagerImpl [sip.0] Disconnected
connection locally
```

7. Grammar Studio

The *Grammar Studio* is designed to create, edit or analyze *Dialog Engine* (DIANE) Grammars, which are used in Speech Applications with Natural Language Understanding (NLU).

This section contains the following content:

The chapter 'general overview' will try to illustrate the features of the *Grammar Studio*, whereas the 'tutorials' will try to guide you through with the help of examples. The 'knowledge base' finally collects chapters about more basic backgrounds.

The FAQ section will collect all interesting questions which can occur and the known issues will allow you to check if there is something that does not work yet or we are planning to improve.

7.1. General Overview: Grammar Studio

The main features of the *Grammar Studio* are creating grammars which can be used in speech-enabled applications, edit existing more complex grammars and test given grammars in different ways.

7.1.1. What is a Grammar File?

A grammar file has two purposes. First of all it specifies which (combination of) utterances will be recognized by an application using this grammar. Nevertheless a grammar also defines a kind of return value according to utterance which will be recognized. This return value will be called the *Semantic Result* of a specific utterance and can be used to allow some application response in regard to a recognized user utterance.
Please review page 120 chapter *Knowledge Base: Grammar Components* to review the classification of some basic terms.

Please regard that the term application will be used in this section as a synonym for IVR Application which means an application with Interactive Voice Response. Furthermore the term grammar will be used as a synonym for a grammar file used by the Dialog Engine (DIANE) in this section.

DIANE grammars are readable text files using the file extension .**grm** . The grammars are created and formatted in a specific internal scheme which is illustrated by the following (simple) example.

```
$ROOT =
  $ACTION:X {: X :}
| $ACTION:X $OBJECT:Y {: X + Y :};

$ACTION =
  listen to {: "LISTEN" :}
| record {: "RECORD" :};

$OBJECT =
  voicemail {: "VMAIL" :}
| a voicemail {: "VMAIL" :}
| email {: "EMAIL" :}
| an email {: "EMAIL" :};
```

Figure: Grammar Code in internal scheme

Because the grammar code shown above is not intuitive for everybody the Grammar Studio was introduced to help users creating and editing grammar files. By using the Grammar Studio you can save the time to study the scheme used by DIANE Grammars files and you do not need to keep track on the correct syntax of your grammar.

7.1.2. What is my Repository Folder?

When you start the *Grammar Studio*, the first thing you will have to do, is to select a *Repository Folder*. The *Repository Folder* is the folder you want to work with. In case you use the *Grammar Studio* to create or edited grammars for applications made with the *Application Builder*, you will have to select the *Application Builders* workspace as your repository folder within the *Grammar Studio*.

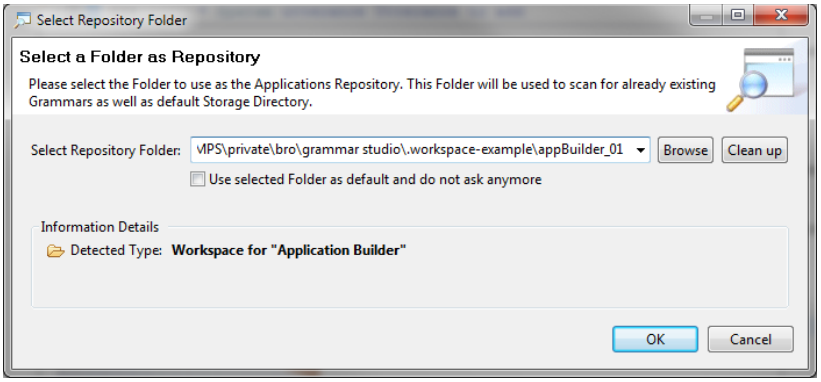


Figure 1: Select your Repository Folder

As the previous screenshot shows, the *Grammar Studio* will check which type of *Repository Folder* is recognized by your selection.

7.1.3. Application Builder Workspace as Repository Folder

When you select an *Application Builder Workspace* as *Repository Folder* for your *Grammar Studio* the content of your selection is presented to you in *Repository Explorer* which uses the left window of the *Grammar Studio Application*.
In the stage of maximum extension you can see four different folder icons named *Workspace Grammars*, *Compositions Grammars*, *Applications Grammars* and *Standard Grammars* in your *Repository Explorer* like it is shown in the following screenshot.

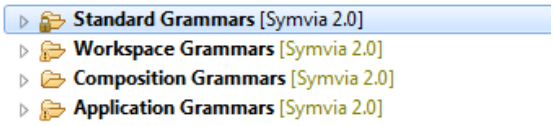


Figure 2: Application Builder Workspace as Repository Folder

Depending on your Application Builder workspace the content can differs, of course. In the most cases you will see at least Applications Grammars and Standard Grammars folders.

The folder containing Standard Grammars is something special, because it offers you those grammars which are officially supported by any DIANE Environment. These Standard Grammars handle the most common topics like recognition of time and date utterances. Please regard that Standard Grammars are read-only files and can therefore not be edited. However Standard Grammars can of course be used in your grammars.

Please read the chapter *Knowledge Base: Grammar Components* for more detailed information on Standard Grammars.

7.1.4. Language-specific Grammars

When you explore your Repository Folder you will find tree nodes representing your grammars (ending on the file extension grm) like it is shown in the following screenshot.

Figure 3: Language-specific Grammar in Repository Explorer

Please regard that these grammar nodes can also be expanded in the *Repository Folder*. That means that grammars will be sub-divided in the context of the *Grammar Studio*. Each grammar can be seen as kind of container for all languages in which the selected grammar has been created. If you expand such a grammar (container) you will see at least one *icon flag* representing the language (and country) for which this Grammar was designed.

If you double-click these icon flags the corresponding language-specific grammar will be loaded into the so-called *Grammar Working Environment*, which represents your working desktop in the *Grammar Studio*. This working environment will make use of the right window part and will allow you to edit or analyze your grammar.
By right-clicking on a *icon flag* an context menu will offer you all possible options for this selected grammar.

7.1.5. Grammar Working Environment

If you have selected your language-specific grammar and want to start editing, analyzing or just reviewing its content the so-called *Grammar Working Environment* is the right choice. It combines four different use-cases into one graphical user interface separated in four different tabs.

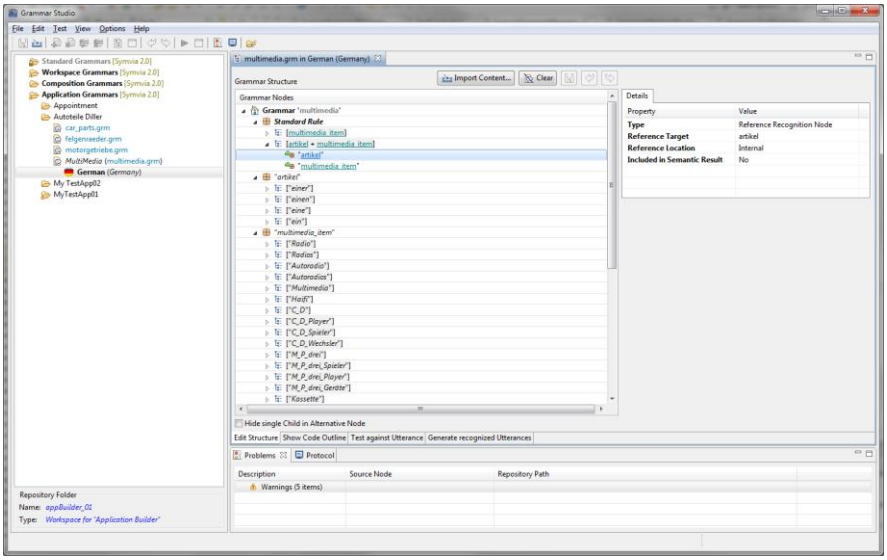


Figure 4: Grammar Working Environment

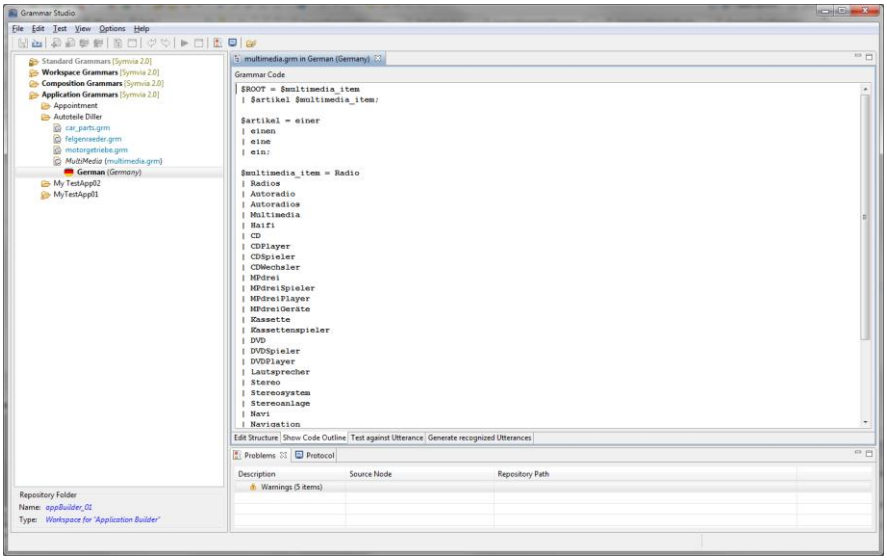


Figure 5: Grammar Working Environment

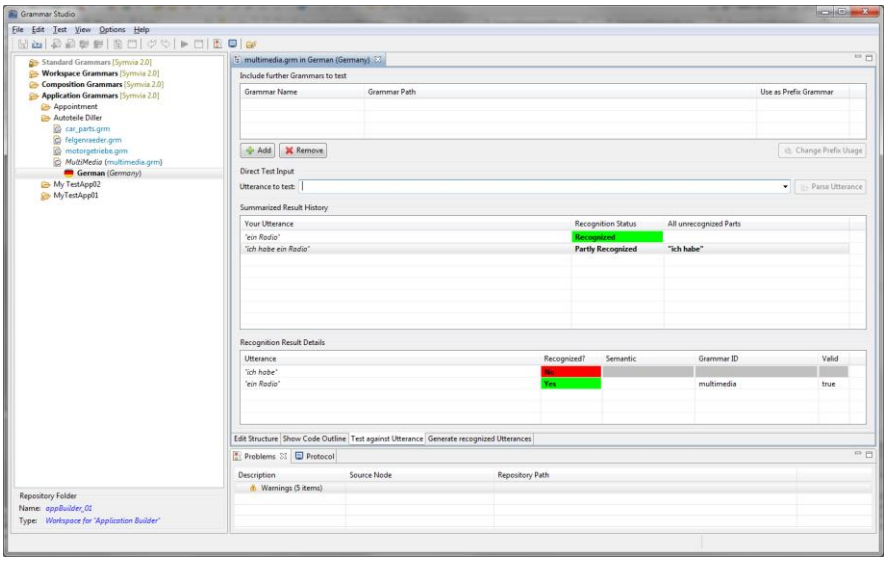


Figure 6: Grammar Working Environment

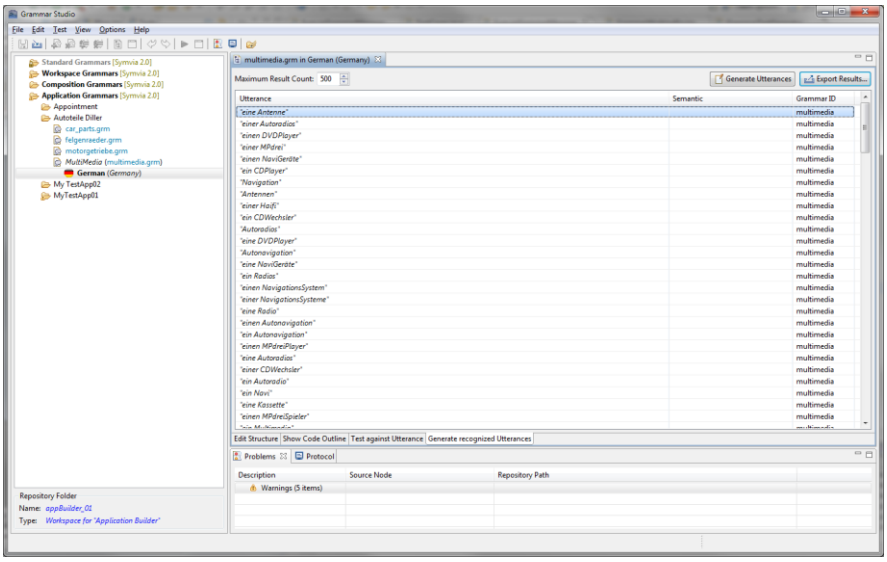


Figure 7: Grammar Working Environment

There is one tab showing you the tree-based structure of the grammar. Here you can edit your language-specific grammar content. The next tab allows you to review the grammar code of the corresponding structure while the third tab supports analyzing the selected Grammar. You can type an utterance and test if your grammar will recognize it. If not, you can directly enhance the recognition of the grammar without editing it manually. Finally the last tab analyzes a grammar by generating possible utterance your grammar will recognize.

7.2. Edit a Grammar

Editing a grammar with the *Grammar Studio* will be done by editing the structure of the grammar in a tree-based layout. If you have double-clicked a language-specific grammar in the *Repository Explorer* this opens the already mentioned *Grammar Working Environment* which supports multiple actions separated in different tabs. As entry point the grammar structure is located on the first tab.

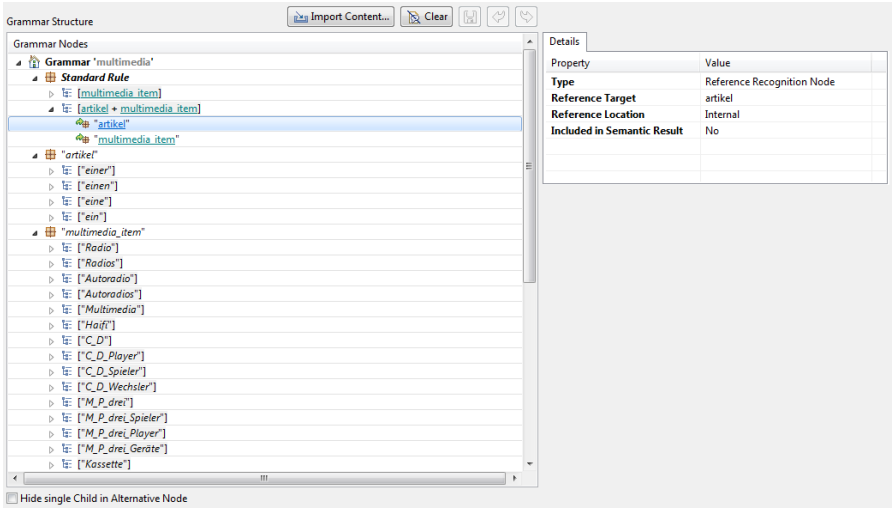


Figure 8: Grammar Structure

The grammar structure itself is shown to you in the left part of the editor as you can see in the following screenshot. Each node represents a specific component of the grammar. A grammar will of course consist of different components and therefore there are also different types of nodes. According to the type of a node, a node can have child nodes or not.

Please read chapter '9.3 Knowledge Base: Grammar Components' for more detailed information on different component types of a grammar.

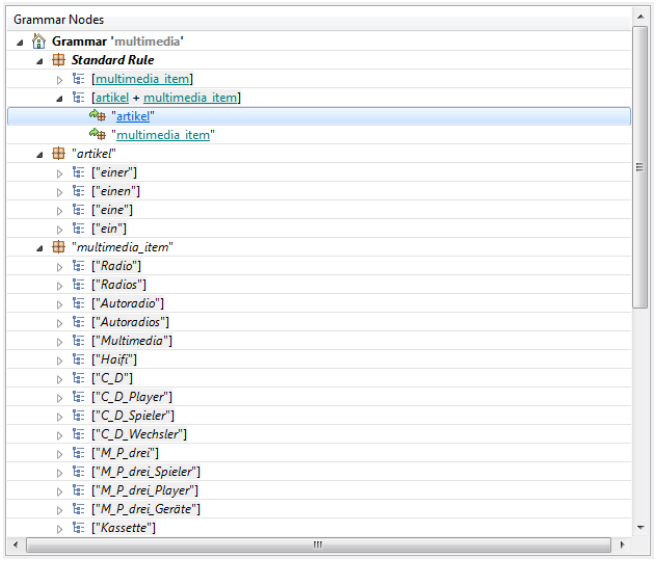


Figure 9: Structure of Grammar in tree-based layout

If you select a tree node the properties of this node are shown in the *Details* pane, which you can see on the right part. The following screenshot shows the properties of a selected node from type *Alternative*.

Please read chapter '*Knowledge Base: Grammar Components*' for more detailed information on *Alternative* nodes.

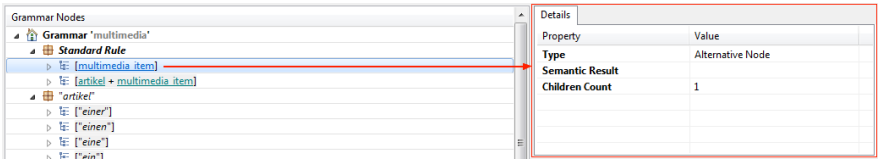


Figure 10: Details of selected node

The properties in the *Details* pane can directly be edited by double-clicking the corresponding value. The following screenshot shows how the property named *Semantic Result* will be edited in the *Details* pane.
Please regard the button '  ' in the screenshot below. It indicates the possibility to open a dedicated editor for the selected property.

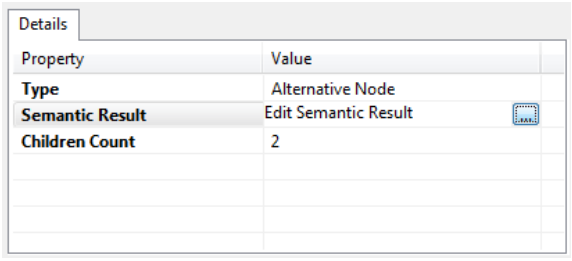


Figure 11: Edit a Property

Additionally each tree node offers possible options in a context menu which can be opened by right-clicking the tree node. Using this context menu is considered as the fastest way to edit a grammar in most cases. Of course the most important options are also offered in the applications menu under *Edit/Grammar Structure/…*.

Editing a Grammar demands some background information about the different components of a grammar. Although the *Grammar Studio* tries to reduce the complexity of editing a grammar, the user should know the meaning and purpose of those grammar components. To keep the clearness of this section the detailed information on grammar components have been sourced out.

For further and more detailed information please review the following *Knowledge Base: Grammar Components* illustrating the different component types of a grammar.

If you need some more practical hints which guide you through the process of editing by examples, please check out the tutorials and review the following Tutorial: Create Grammar.

7.3. View Grammar Code

If you just like to take a look at the textual code representation in the internal grammar scheme, you can use the option to view the grammar code.

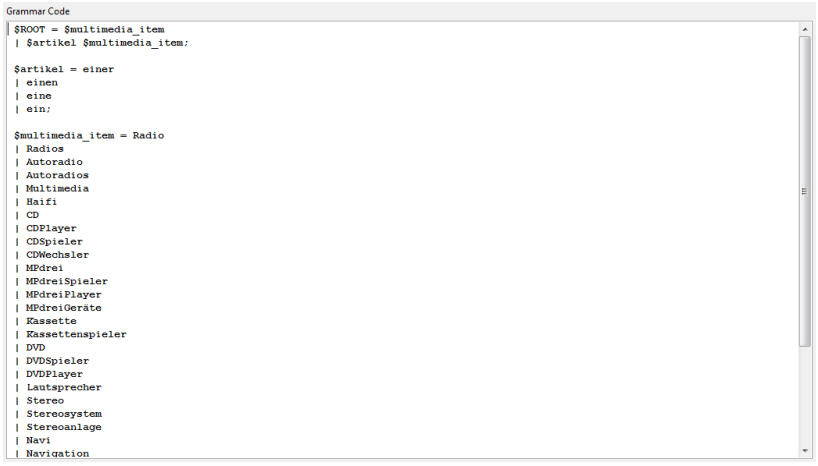


Figure 12: Grammar Code

As you can see in the screenshot above, this option will show you the internal code of your grammar. This is what your grammar would look like in a simple text editor.

Currently this view does not offer you any additional benefit like direct editing. In a further development of the *Grammar Studio* this option will hopefully be enhanced to allow direct (and faster) manipulation of the grammar code for more experienced user.

7.4. Analyze Grammar

After you have created a grammar you probably want to test it. If your grammar has several *Recognitions* or *Recognition References* which are used in different *Rules* it will be quite complicated to predict the complete set of utterances your Grammar will recognize. Therefore you can analyze the grammar and check some user utterances against it. The following screenshot shows how this will look like in the *Grammar Studio*.

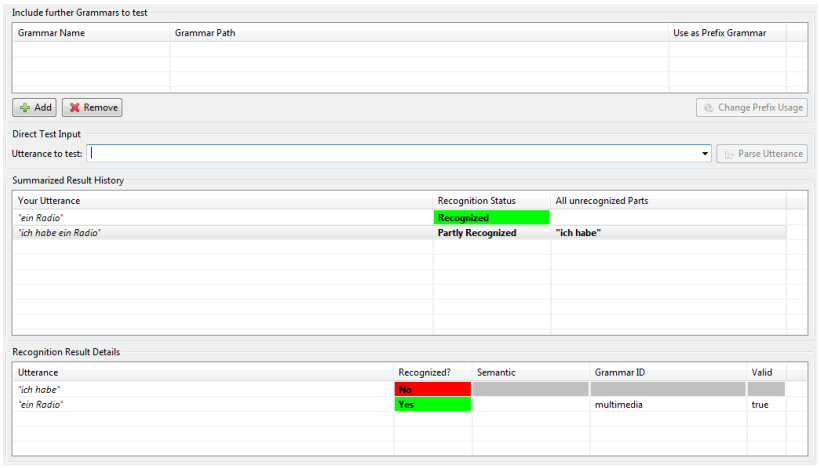


Figure 13: Analyse user utterances

The previous screenshot illustrates this quite easy way of testing in principle. Simply type in what a user would give as audible input and press the «Parse Utterance» button. Please regard that the current version of the *Grammar Studio* is case-sensitive in context of the given user utterance. This means that an user utterance has to match exactly a phrase which is defined as *Recognition* in your grammar. The following screenshot shows where you have to type in your test utterances.

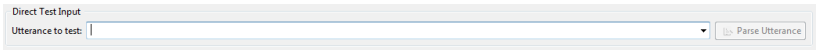


Figure 14: Test input representing user utterance

As a test feedback you will first of all get a summarized three-stepped recognition status. Your grammar can recognize the given test input partly, completely or not all. In the first case the unrecognized parts of the given test input will be shown to you. This information can be very useful if you want to improve the recognition of your grammar. The following screenshot shows up some examples of summarized results.

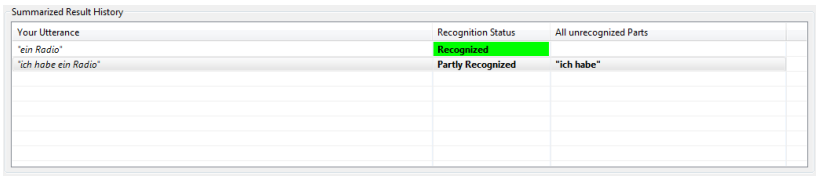


Figure 15: Summarized Results

Additionally you have the option to get some more detailed feedback which will allow a closer look at the un-/recognized parts. Just select your recognition result in the summarized result table. This will display the corresponding details in a further table. For more information about the detailed feedback a Tutorial: 'Analyze Grammar' is planned. Some sample result details can be seen in the following screenshot.

Kommentar [JHK2]: Lacking SDK Article Tutorial: Analyze Grammar

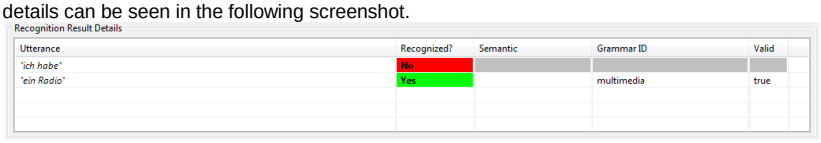


Figure 16: Result in Details

If you want to analyze your grammar in combination with other already existing grammars, you will have to include the desired grammars to your current test process. Of course this will not change anything in the tested grammar itself.

Including grammars for the current testing process can help you out in situation where you know that multiple grammars are active in the *DIANE Runtime* at the same time. In fact this is the case for most applications, because in nearly all applications there will be grammars for different dedicated purposes.
By including other grammars to the test process you can simulate such situations. The following screenshot illustrates where you can include additional grammars for the current testing.

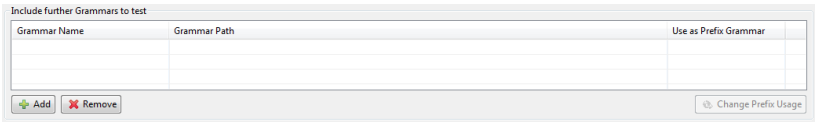


Figure 17: Include additional Grammars to Test

Finally the *Grammar Studio* supports to enhance your grammar directly from within the analyzing process. If you have some unrecognized parts shown in your result details, the *Grammar Studio* supports to automatically improve the recognition of your grammar.

Simply open the context menu by right-clicking on an unrecognized part and select the quick fix option. This will enhance the tested grammar without any further user action. If you have defined additionally grammars, which are included in your test, you can also specify one of those grammars to enhance (if they are not read-only ones).

If you need some more practical hints which guide you through the process of analyzing, please check out availability of the planned Tutorial: Analyze Grammar.

Kommentar [JHK3]: As above

7.5. Generate possible Utterances of a Grammar

After you have already checked that the grammar recognizes your desired (combinations of) utterances, you probably like to know which permutations of utterances it recognizes furthermore. So another way to test a grammar is by generating the possible utterances of a grammar. This test can be started as simple as possible. Just select the desired maximum count of results and press the corresponding button. This process can take some time and differs depending on the complexity of the grammar.

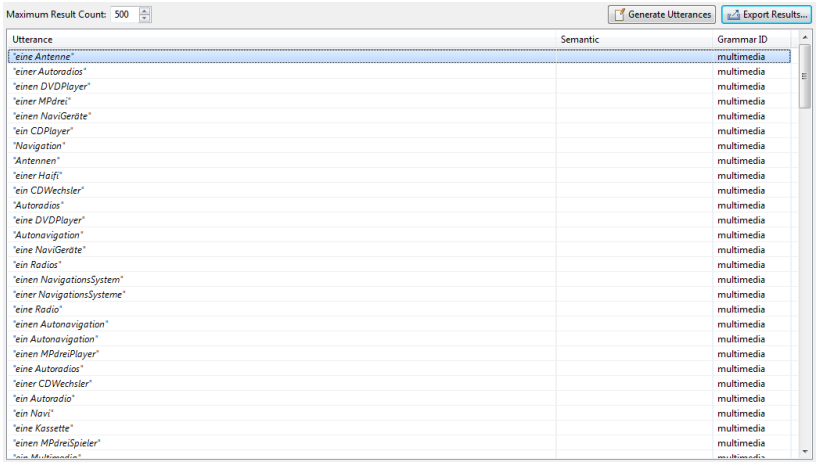


Figure 18: Generate possible Utterances

The generated list can be exported as a comma separated text file to import it e.g. in *Microsoft Excel*® for documentation purposes. Just press the corresponding button and select a target file (name).

8. Tutorial 'Create Grammar'

This tutorial will give you a detailed view on how to create a grammar. In a closer look it will guide the reader step-by-step through the process of creating grammars for an application which was made with the *Application Builder*.

8.1. Introduction

Within this tutorial we will create several application grammars on different ways. This will give you an overview over the different possibilities you will have and it illustrate how the interaction of *Application Builder* and *Grammar Studio* will work.

We will start in the *Application Builder* where we will import a sample application which makes use of *Natural Language Understanding* (NLU) and therefore will need one or more grammars.

Such a grammar will be used to define the allowed user inputs. Furthermore it can (and will) be used to return the meaning of a recognized user utterance which we call *Semantic Result*.

For more information about *Semantic Results* please review the *Knowledge Base: Understanding Semantic Results* on page 126.

The grammars needed by the sample application will be created within this tutorial. On the one hand the tutorial will make use of the *Grammar Studio* which is a standalone *Rich Client Platform* (RCP)-Application. On the other hand it will directly use the *Grammar Wizard* which is directly integrated into the *Application Builder*. This wizard offers you an intuitive way to create and edit simple grammars, whereas the *Grammar Studio* offers you more options to edit and test grammars.

To illustrate the interaction between *Application Builder* on the one side and *Grammar Studio* on the other side, this tutorial will also show up how to switch between these two programs. This will especially show you how to create a grammar in the one application and use or edit it in the other one or vice versa.

8.2. Content

The content of this tutorial was split into several pages to keep a better overview. If you start studying the tutorial for the first time simple read through all the pages in the order of appearance. If you only want to review one special aspect, dive directly into one of the following subchapters.

8.2.1.1. Prepare your personal working environment...

- Step 01: General Preliminaries
- Step 02: Download Sample Application as initial Draft
- Step 03: Import Sample Application into Application Builder
- Step 04: Verify successful import of Sample Application
- Step 05: Configure Sample Application in Application Builder

Kommentar [JHK4]:

Hier hatten wir einen Verweis auf 1 nicht vorhandenes 2. Tutorial

Tutorials
This section was created to offer some tutorials which guide the reader step-by-step through the described process. This section contains the following tutorials:
- Tutorial: Create Grammars for an Application which was made with the Application Builder
- Tutorial: Analyze a Grammar

8.2.1.2. Introduce application...

- Step 06: Introducing the Sample Application

8.2.1.3. Create a Grammar...

- Step 07: Taking a first insight into the Sample Application
- Step 08: Create a Grammar to recognize User Utterances using the Grammar Studio only

8.2.1.4. Create a Customized Garbage Grammar...

- Step 09: A deeper insight into the Sample Application
- Step 10: Create a customized Garbage Grammar using the Application Builder only
- Step 11: Create a customized Garbage Grammar using the Application Builder and the Grammar Studio

8.2.1.5. Check your results...

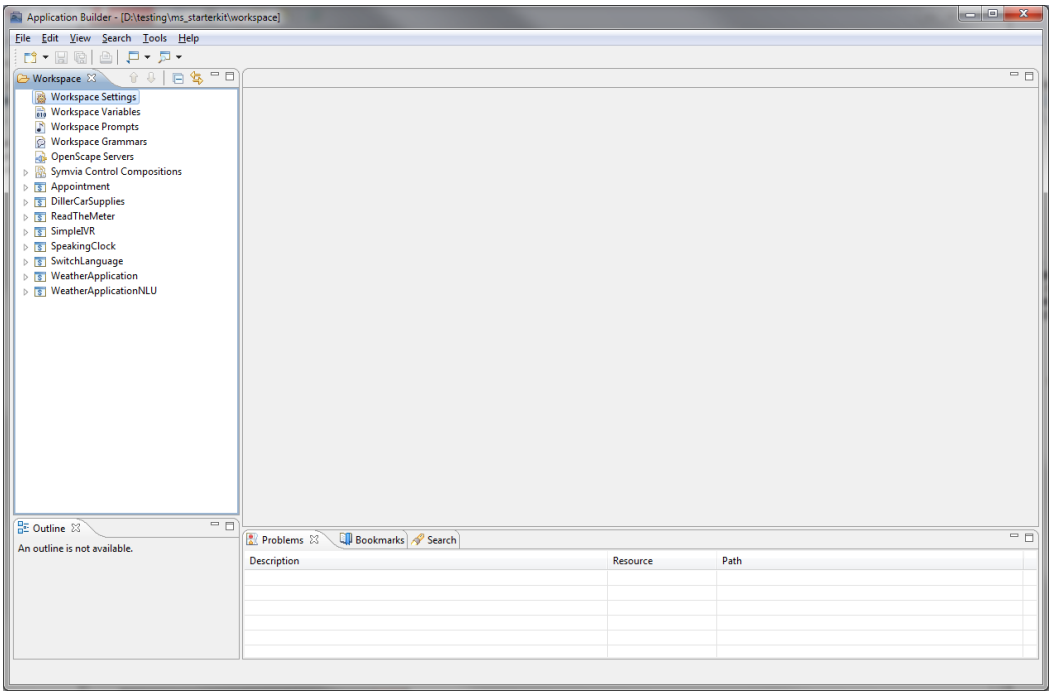
- Step 12: Verify your Work

8.3. Tutorial Step #1: General Preliminaries

First of all we need to install the *Media Server Starter Kit* if this is not already done. By installing the starter kit an *Application Builder* and a *Grammar Studio* will be installed on your system in addition to the OpenScape Media Server and a SIP soft phone. This is described in the chapter *Installation & Startup* which includes a section to learn how to prepare your environment, too.

After your development environment is successfully setup, you will have to **start up the** *Media Server*. Please review these instructions to start the *Media Server*, which will be needed to run your applications which you will create with the help of the *Application Builder*.

Furthermore we need to **start up the** *Application Builder* to begin this tutorial. Therefore please review the corresponding page to learn how to start the Application Builder. After you have successfully started your Application Builder your screen should look similar to this screenshot:



8.4. Tutorial Step #2: Download Sample Application as initial Draft

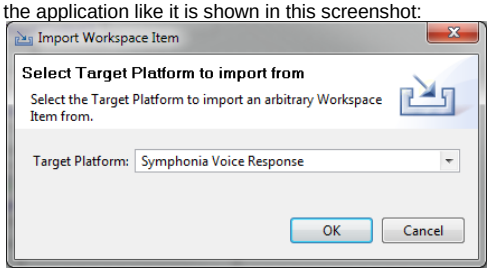
This tutorial will guide you through the process of creating and editing grammars for applications using *Natural Language Understanding* (NLU). Please regard that creating an application will be not content of this tutorial.
For a guided tour helping you to create an application from the scratch with the *Application Builder* please review the instructions on page 36. Instead this tutorial will make use of a sample application which has to be imported into your local *Application Builder* workspace. This application will only be an initial draft representing the fundament of this tutorial. It can

be downloaded from the Unify's Fusion Developer Portal in the same download area where you find the Starter Kit. You need to download a zip file named as ***«tutorial_dillercarsupplies_initialdraft.zip»*** which represents an *Application Builder Archive File*. This archived package contains finally our sample application named «Tutorial - DillerCarSupplies (Initial Draft)».

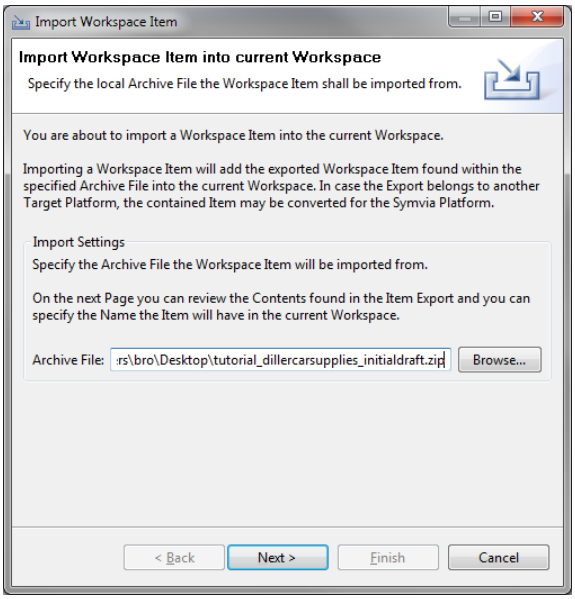
8.5. Tutorial Step #3: Import Sample Application into Application Builder

To import the sample application into the *Application Builder* you do not need to unzip the downloaded zip package. Just start the import process by clicking «File/Import Workspace Item…» within the *Application Builder*. This process will include three steps to follow:

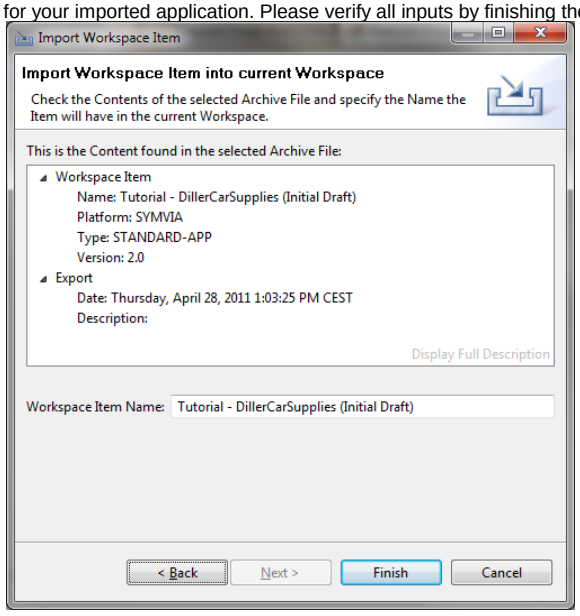
As a first step you will need to select a target platform. Please select «Symphonia Voice Response» as target platform for the application like it is shown in this screenshot:



In the next step you will have to select a zip package as archive file. Please download the *Application Builder Archive File* for the initial draft version of our tutorial/sample application and select this zip file on your local hard disk like it is shown in this screenshot:



As a last step you have to verify your selection and accept the proposed name of the workspace item which will be created for your imported application. Please verify all inputs by finishing the dialog like it is shown in this screenshot:



8.6. Tutorial Step #4: Verify successful import of Sample Application

After you have successfully imported the initial draft of our sample application «DillersCarSupplies» you (hopefully) will see a newly created workspace item named «Tutorial - DillersCarSupplies (Initial Draft)». To verify that everything is imported correctly, your *Application Builder* workspace has to look similar to the following screenshot:

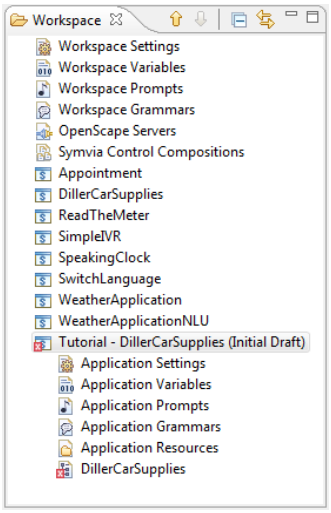


Figure 19: Imported Application

Although the import was successful (at least) an error will be reported for our sample application, because a specific control references an undefined grammar. On the following screenshot you can see how this error will be reported. Please regard that this problem was explicitly created for this tutorial and will be solved when we have finished the tutorial.

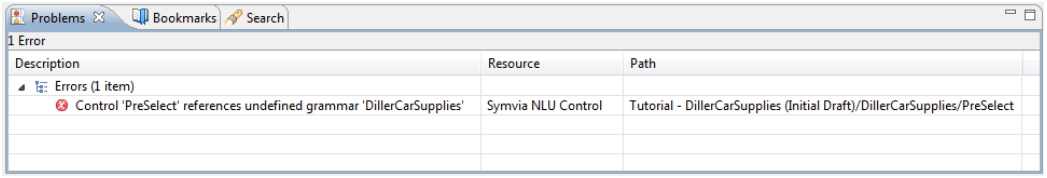


Figure 20: Missing Grammar

8.7. Tutorial Step #5: Configure Sample Application in Application Builder

If your *Application Builder* reports additional problems for our sample application, you probably will have to change the default application language. In this case an error will be reported, that the application settings do not specify a default language. The following screenshot illustrates how this would look like.

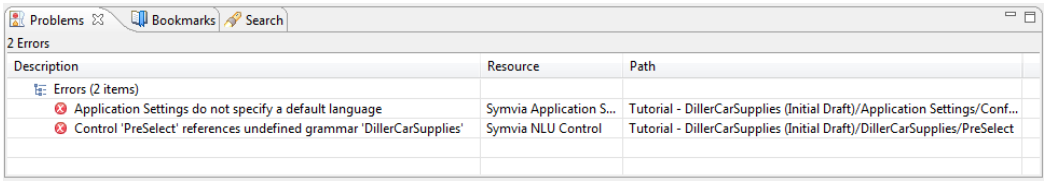


Figure 21: Missing Default Application Language

If your *Application Builder* does not report this problem, you can skip this chapter at this point and we are ready to go. In the other case, this chapter will help you to solve this problem.

To solve the problem with the application default language just double-click on this reported error. The *Application Builder* will open the *Application Settings* where you edit the *Language Settings*. According to the configuration of your *Application Builder* you can only select a language as default application language if it was defined as *enabled default language* within the *Application Builder* itself. Please select «**English (United States)**» as default application language. If this language is not offered, please enable this language in your *Application Builder*. By default only «English (United States)» will be supported by our sample application. The following screenshot shows how to principally select an application default language.

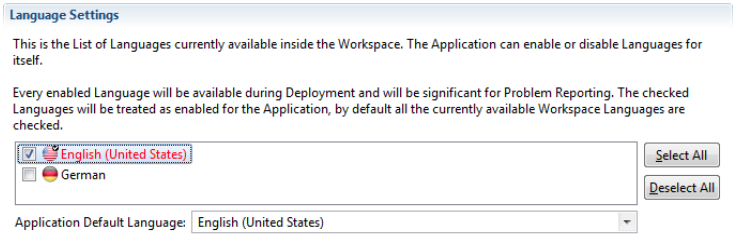


Figure 22: Select Default Application Language

After the application default language has been set and the settings were successfully saved, there should be no more reported error for this application and we are ready to go on.

8.8. Tutorial Step #6: Introducing the Sample Application

At this point you will need some more background information about our sample application. What is the purpose and what is done by this sample? Please study the application call flow which is also named «DillerCarSupplies» in the following screenshot and make a first idea of this application on your own.

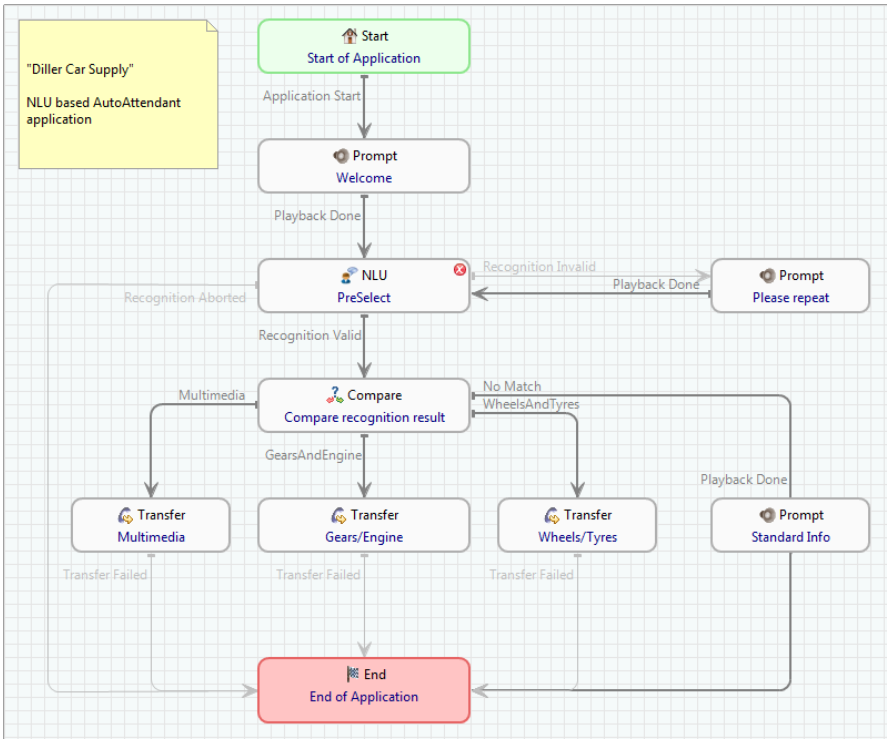


Figure 23: 'Diller Car Supply' Application Call Flow

This little IVR application represents a **speech-enabled information service** for an anticipated retailer of car parts. Customers will be welcomed before they will be asked what products they need more information on. The application will accept the following three different product categories:

- Car Hi-Fi and multimedia
- Gears and engines
- Wheels and tyres

This means that customers who call this application can request information about for example «Car Radios» which would be a possible product for the category «Car hi-fi and multimedia». Please keep in mind that this application is speech-enabled. Thus potential callers will tell the application his or her requests by natural speech.
If those customers request information for a product of one of those categories, the application will try to transfer the customer to an dedicated information desk which is in fact just another phone number in our sample. If there is no device registered for those phone numbers the transfer will not work and the sample application will end the call. Of course this application logic is neither perfect nor realistic, but it is sufficient for this tutorial.

8.9. Tutorial Step #7: Taking a first insight into the Sample Application

By now you can define the purpose of our sample application. But what is our concrete task to do now in this tutorial? Let us take a first deeper insight into the sample application to finally answer this question.

8.9.1. What is a NLU Control?

As you can see the in the application call flow shown in Figure 23 this sample application uses a *NLU Control* which is an *IVR Control* supporting the understanding of natural language. Natural language in this context describes an everyday language which would be used in a dialog between human beings. This control therefore will enable the application to understand user utterances in natural language and it will enable the application to react on these utterances.

However a *NLU Control* is not able to understand any user utterance without a grammar. **Only a grammar defines the user utterances which the application will recognize**. Therefore this grammar has to be created in this tutorial.

8.9.2. For what does a NLU Control need a Grammar?

Do you still remember that this sample application reported an error that was already mentioned earlier in this tutorial? This error (Figure 20) occurred, because a specific control references an undefined grammar. If you inspect the screenshot once again, you will see that this error-reporting control is a *NLU Control*. - This means that the *NLU Control* **needs a grammar to recognize user utterances** and it already references such a grammar to recognize user utterances. The (only) problem is that the referenced grammar itself is not available - in other word: Somebody has removed the grammar file (only for training purposes, of course).

8.9.3. What is the purpose of the Grammar?

As we know by now, **we have to create a grammar to get the** *NLU Control* **working again**. But what shall this grammar do and how should it be named? To get answers to these questions let us look once again at the reported error shown in the screenshot above (Please see Figure 20). Just double-click the reported error to let the *Application Builder* focus on the cause of this error. Right away the configuration dialog of the *NLU Control* will be opened. The following screenshot shows you how this would look like:

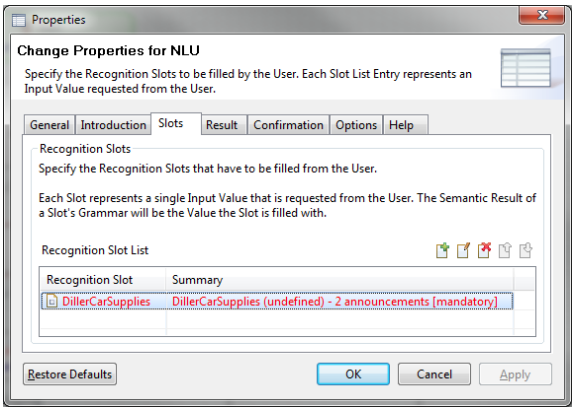


Figure 24: Recognition Slots of NLU Control

As you can see the *Application Builder* will have opened a specific tab named «Slots». This is the place where recognitions of user utterances will be handled. Each of those slots represents a grammar which not only defines the allowed user utterance but also defines the meaning of an utterance. Please regard that such meanings are called *Semantic Results* in the context of grammars.

If you need more detailed background information about *Semantic* Results please review the *Knowledge Base: Understanding Semantic Results* on page 126.

But for what will the *Semantic Result* of this grammar be used now? To answer this question, you have to remember yourself what the reaction of a recognized user utterance will be. As a reaction our sample application will route the call flow (from the *NLU Control*) to specific successor controls. As you remember in this sample these successors were controls transferring to dedicated phone numbers.

Summarized once again: The *NLU-Control* in our sample application already defines a «Recognition Slot» which references a grammar recognizing user utterances. This means our grammar has to make use of *Semantic Results*, because it has to return the meaning of a user utterance. This *Semantic Result* will be used to route the application call flow.

After we know the purpose of the grammar, which we have to create, we finally have to know its name. This is important in this tutorial, because in this sample application the *NLU Control* already references a specific grammar name.

To reveal the used grammar name we only need to double-click the «Recognition Slot» shown in the screenshot above. This will open a further dialog to edit this slot. Of course we do not want to change anything but only want to check the used grammar name. The following screenshot shows how this would look like.

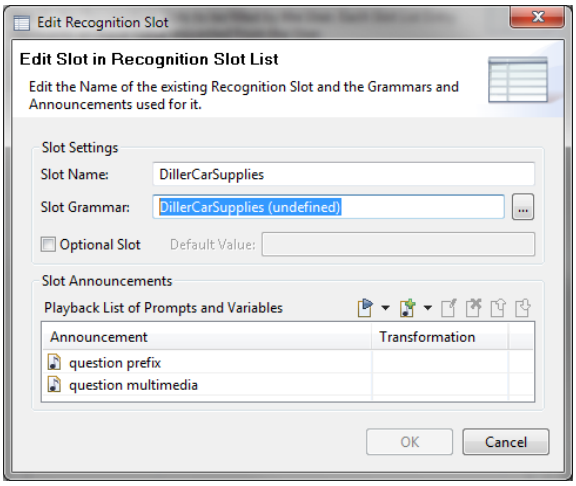


Figure 25: Grammar of Recognition Slot in NLU Control

As we can finally see in the screenshot shown above, the grammar of this «Recognition Slot» (the so-called «Slot Grammar») is named «DillerCarSupplies».

8.9.4. Definition of our First Job

As a final conclusion our first task will be to create an application grammar named «DillerCarSupplies». When creating the grammar we should take the following aspects into regard:

- The grammar has to define all possible utterances of customers of a car parts retailer
- These user utterances can be separated into three different product categories
- Each product category will contain several different products
- The grammar has to return a *Semantic Result*
- The *Semantic Result* shall enable routing the application call flow according to the recognized user utterance

8.10. Tutorial Step #8: Create a Grammar to recognize User Utterances using the Grammar Studio only

Now we are ready to start creating the grammar which is described as our first job in the previous chapter. As mentioned before there are several ways to do so and one of those will be to start right away with the *Grammar Studio*.

8.10.1. Reasons for using the Grammar Studio

Using the *Grammar Studio* is suggested in the following use cases:

- The grammar to create seems to be an **extensive or complex** one
- The grammar shall re-use other non-standard grammars
- The grammar shall be manually **structured** to have a better overview for future developments
- You want to **test** your grammar right away and see what user utterances will be recognized

In this case the grammar to create seems to be a (relative) extensive and complex one. Therefore we should use the *Grammar Studio* to create the grammar for our sample application. To begin the process of creating, please **start the** *Grammar Studio*. For a general overview and basic information please review chapter *General Overview: Grammar Studio* on page 62.

8.10.2. Select Application

When starting up, the *Grammar Studio* which is delivered within the starter kit will use the default workspace of the *Application Builder* as folder to work with. This folder will be called the *Repository Folder* in the *Grammar Studio*. The *Repository Explorer* will list all found Grammars in this *Repository Folder*. The following screenshot will show how this would look like.

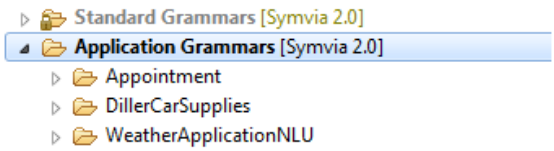
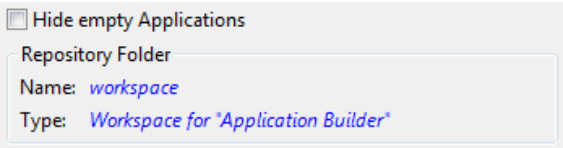


Figure 26: All Grammars in Application Builder Workspace

As you see in screenshot above, our sample application is not listed. The reason for that is simple. Our sample application just does not contain any grammar up to now. Per default application without any grammar are not shown in the *Repository Explorer*, but this can be changed. Simply deactivate the filter to hide empty applications as it is shown in this:



If you deactivate the filter option to hide empty application grammars, the listed content in the Repository Explorer will change and also applications without any grammar like our sample application will be displayed as you can see in the following screenshot:

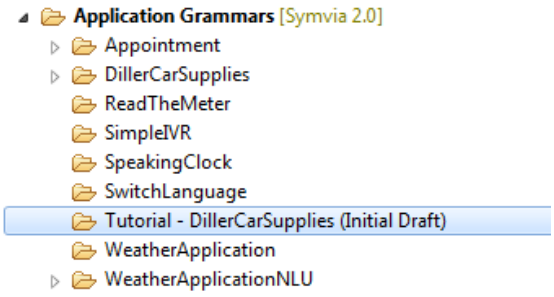


Figure 27: Repository Explorer showing also Applications without any Grammar

8.10.3. Create a Grammar Container

Before we start to create a new grammar for our sample application, please remember yourself that any grammar is language-specific. This means that there would be a grammar for english content as well as a grammar for german content if our application would support those two languages. In the *Grammar Studio* all language-specific contents of a grammar are collected into one so-called *Grammar Container*. This container therefore represents a grammar in all defined languages.

As described in the previous passage, we have to create a *Grammar Container* as a first step. To achieve this you can use the application menu like it is shown in the following screenshot. Alternatively can you also use the context menu by right-clicking on the node representing our sample application.

Figure 28: Create Grammar Container

Selecting this option will open a dialog where you have to define a name for this Grammar Container. In fact this name will be used by all language-specific grammars. We will use the name «DillerCarSupplies.grm» here, as we remember the definition of our first task. The following screenshot shows how this will look like.

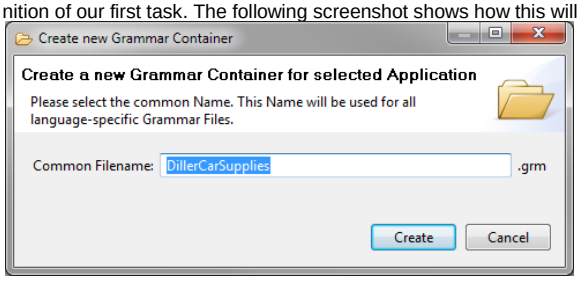


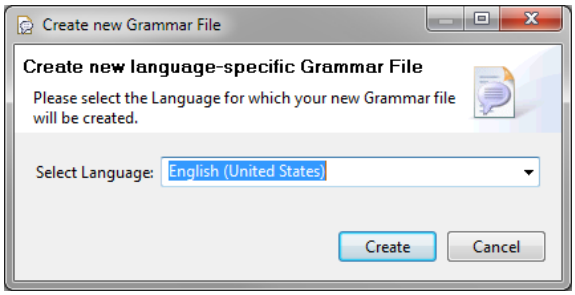
Figure 29: Select Grammar Name

8.10.4. Create a language-specific Grammar

After creating a *Grammar Container* named «DillerCarSupplies.grm» we have to create (at least) one language-specific grammar for this *Grammar Container*. Please proceed analog to the section above. You can use the application menu show like it is shown in the following screenshot. Alternatively can you also use the context menu by right-clicking on our previously created *Grammar Container*.

Figure 30: Create language-specific Grammar

Selecting this option will open a further dialog where we have to define the grammar language. As our sample application is designed in American English we have to use the Language «English (United States)» here. If our sample application would support another language, we would have to create a further language-specific grammar. This screenshot illustrates how the dialog would look like:



The Grammar Studio will now create an empty grammar template which content can now be edited in the needed way. The following screenshot shows how your Repository Explorer should look like.

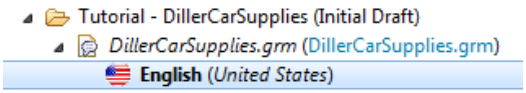
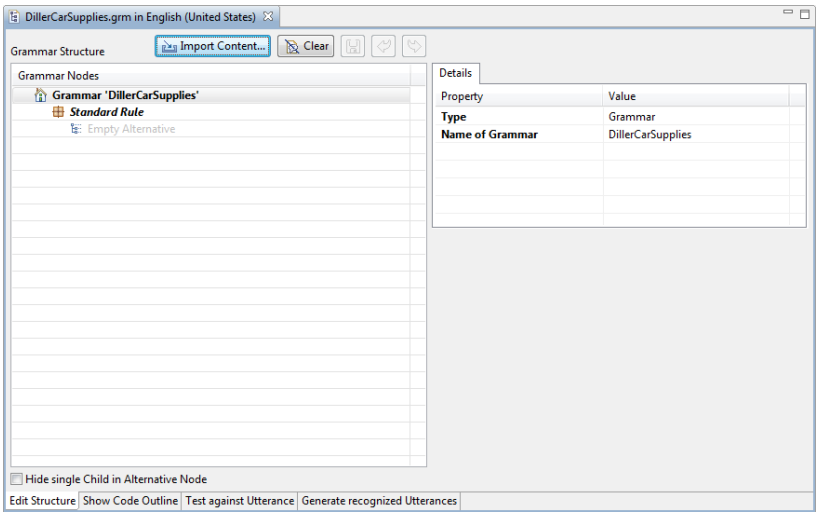


Figure 31: Created language-specific Grammar

8.10.5. Start editing language-specific Grammar Content

To start editing just double-click the previously created grammar for american english in the *Repository Explorer*. Alternatively you can of course use the content menu by right-clicking the language-specific grammar. This will load the grammar content into the editor like it is shown in this screenshot.



Now we are able to edit the content in a tree-based view of the grammar structure. This tree-like overview displays each grammar component as an own tree node. Because a grammar is constructed by different grammar components, this structure view will contain several different node types.

Each of those grammar components has a special purpose and meaning. To give you a better overview we have collected detailed information about all supported grammar components in *Knowledge Base: Grammar Components* on page the 120. Please review this chapter at least once or open it as an additional browser windows/tab to have direct access to this information till the end of this tutorial. In general all supported options of a grammar component will be shown to you if you open the content menu by right-clicking a specific tree node which represents a grammar component.

Divide Grammar Structure logically

As we remember the definition of our first task, this grammar has to define all possible utterances of potential customers. The objects of those user utterances will refer to three different product categories that we have already defined in this previous chapter.

To keep the grammar open-ended, we should structure the grammar according to these product categories. If the application needs to support a fourth product category in future, this could simply be added without refactoring the complete grammar.

In this case creating an open-ended grammar means to map the grammar structure to the application product categories. Therefore we have to divide the structure of the grammar logically.

If you did not spend time to review the detailed information about all supported grammar components in the *Knowledge Base: Grammar Components*, you will need to know that we have to use *Rules* to divide the grammar structure into logical sections.

This means we have to add one *Rule* for every applications product category. To add a *Rule* please open the context menu by right-clicking the parent tree node of possible *Rules*, which is in fact the node representing the grammar itself. The context menu will offer an option to add a new Rule.

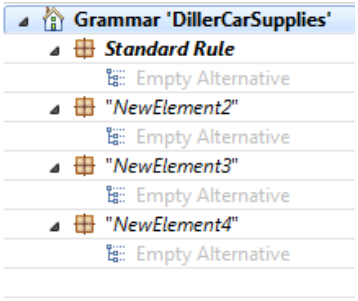


Figure 32: Add Rules as logical Division

These generically named *Rules* should be renamed for a better overview. Because we map these *Rules* to the applications product categories, we rename this *Rules* to «wheelstyres», «gearsengine» and «multimedia». Again please open the context menu by right-clicking a *Rule* and select the option to edit it. As result a dialog will be opened where you can change the name of the selected *Rule*. This screenshot illustrates how this dialog will look like.

After renaming all your newly introduced Rules your grammar structure should look it is shown in the following screenshot.

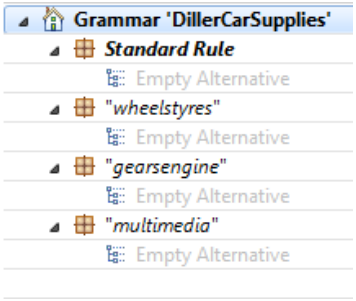


Figure 33: Renamed Rules

8.10.6. Add possible User Utterances

Each of these newly introduced logically divisions needs to be filled now with several user utterances, which a potential customer could use while interacting with the application. A possible start for editing the grammar would be the product category handling wheels and tyres.

If you want to add possible user utterances for a category, we have to regard the context of this grammar in our sample application. In our sample application the grammar will be used to recognize a product of a specific category (e.g. «wheels and tyres») to route the further application call flow.

Therefore we have to think about the question: What are possible products a customer could ask for in this category?
As possible answers to this question could have replied for example the following phrases (in the case of the category «wheels and tyres»):

- «Wheels»,
- «Rims»,
- «Aluminium wheels»,
- «Steel rims»,
- etc…

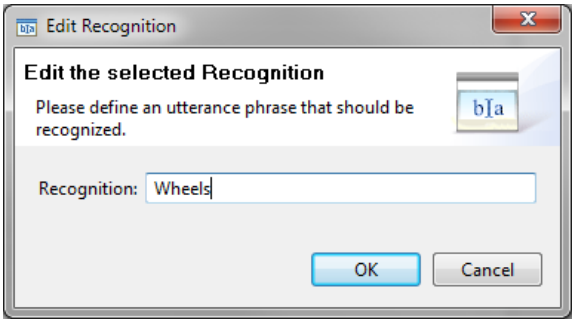
After collecting possible user utterances, we now will try to add those possible user utterances to the grammar. Please do not forget corresponding singular versions of those phrases to support a little more variety.

If you did not spend time to review the detailed information about all supported grammar components in the *Knowledge Base: Grammar Components*, you will need at least a basic summary of the relation between the needed grammar components. For a general proceeding you have to know that the entry point of our grammar will be the so-called *Standard Rule*. Like every (other) Rule it can consists of one or more *Alternatives*. Finally an *Alternative* can contain one or more user utterances.

As described above, we need to do the following for every user utterance:

- **Step A1**: Add an *Alternative* to a *Rule* (or use an empty *Alternative*)
- **Step B1**: Add a *Recognition* of the user utterance to the previously created *Alternative*

Because each newly introduced *Rule* (see Figure 5.2) already offers an empty *Alternative* we can skip *Step A1* for the first user utterance and start right away with adding a *Recognition* to this *Alternative* in *Step B1*.
Just right-click on the tree-node representing the empty *Alternative* to open the context menu and select the option to add an *Recognition*. This will open a dialog where you can define the utterance the grammar will recognize. The following screenshot illustrates how this would look like.



Adding the user utterance «Wheels» as a first Recognition will change the grammar structure. The following screenshot shows how this would look like.

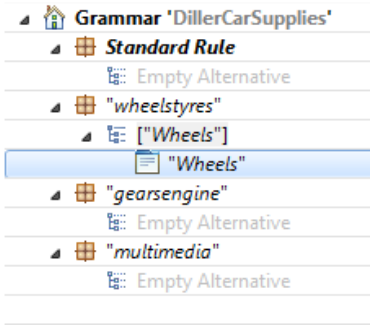


Figure 34: Added Recognition

To add the next user utterance we have to create a new (and empty) *Alternative* first (*Step A1*). Therefore just open the context menu of the current *Rule* by right-clicking on the tree node which represents this *Rule*.
The *Rules* context menu will offer you the option to add a new *Alternative*. Please proceed analog with this empty *Alternative* (*Step B1*) like it was already described above.

After adding *Recognitions* for all collected user utterances (and their singular versions) your grammar will look like the following screenshot.

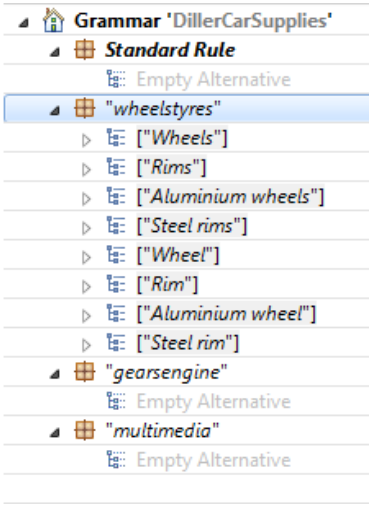


Figure 35: User Utterances of one Product Category as Recognitions

Please proceed analog with the other Rules. Each *Rule* represents a logically devision and includes several products which a potential customer could ask for in the given category. The following list contains the products which have to be recognized in the specific *Rules*:

Products to recognize in Rule «gearsengine»

- «Motors»,
- «Gear boxes»,
- «Injections»

Products to recognize in Rule «multimedia»

- «Car radios»,
- «Amplifiers»

The following screenshot shows how the grammar structure should look like after you have added all potential user utterances.

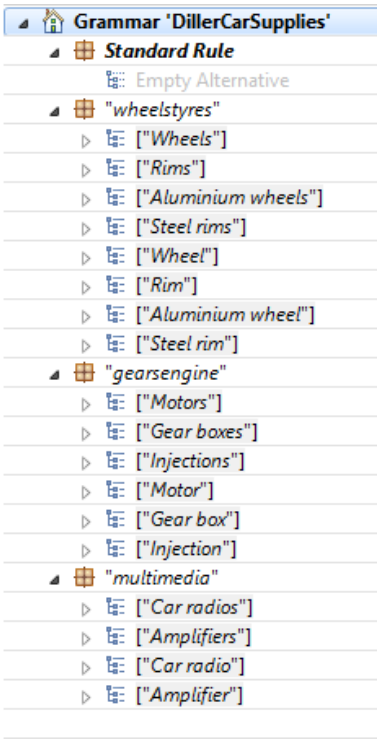


Figure 36: All User Utterances as Recognitions

8.10.7. Include logical Divisions to Grammar Entry point

By now, we have mapped the grammar structure to the applications product categories. The grammar structure is separated into three different logical divisions each represented by one *Rule*. However this grammar will not recognize any of those defined user utterance, because there is still something missing.
If you did not spend time to review the detailed information about all supported grammar components in the *Knowledge Base: Grammar Components*, than you will need the following basics about how to include *Rules*. You have to distinguish between the declaration of a *Rule* and the usage of a *Rule*. The declaration of our *Rules* is already done, but there are not used/included by anybody. To include an already declared *Rule* to a grammar we have to make use of it by defining a reference to this *Rule*.

The entry point of each grammar will be its so-called *Standard Rule*. No other *Rule* will be taken into regard per default if this *Rule* is not used.

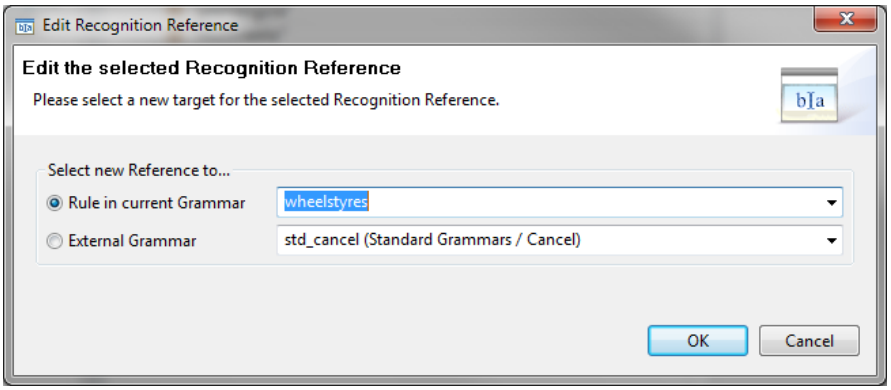
In our grammar this *Standard Rule* contains only an empty *Alternative* so far. This means that this grammar will recognize simply nothing, because the *Standard Rule* does not make use of any of our previously defined *Rules*. **To keep it short: The references to those previously defined** *Rules* **are still missing in the** *Standard Rule***.**

Let us take a look at the *Standard Rule*. Here we have to edit the grammar structure to include the three previously declared *Rules*. Because each of those *Rule* represents an applications product category, each *Rule* has to be seen as one own possible variation in the context of our sample application. A potential customer could just request information about a product of the first category *or* of product of the second category *or* of the third category. This disjunction of different variants is perfectly reproduced by the grammar component *Alternative*.

As described above, we need to do the following for every *Rule*, which we want to include:

- **Step A2**: Add an *Alternative* to the *Standard Rule* (or use an empty *Alternative*)
- **Step B2**: Add a *Recognition Reference* to the previously created *Alternative*. The reference target will be the *Rule*, which we want to include

Because the *Standard Rule* (see Figure 5.2) also offers an empty Alternative we can skip *Step A2* for the first reference and start right away with adding a *Recognition Reference* to this *Alternative* in *Step B2*.
Just right-click on the tree-node representing the empty *Alternative* to open the context menu and select the option to add an *Recognition Reference*. This will open a dialog where you can select one of our *Rules* as a reference target.



The screenshot above shows how to set a *Rule* as target of a reference in principle.

Using the *Rule* «wheelstyres» as reference target of a first *Recognition Reference* will change the grammar structure. The following screenshot shows how this would look like.

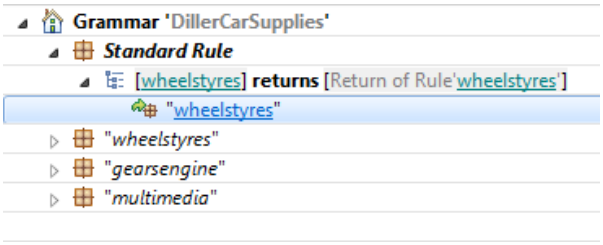


Figure 37: Added Recognition Reference

To add the next reference we have to create a new (and empty) *Alternative* first (*Step A2*). Therefore just open the context menu of the current *Rule* by right-clicking on the tree node which represents this *Rule*.
The *Rules* context menu will offer you the option to add a new *Alternative*. Please proceed analog with this empty *Alternative* (*Step B2*) like it was already described above.

After adding *Recognition References* **for all previously created** *Rules* your grammar will look the following screenshot.

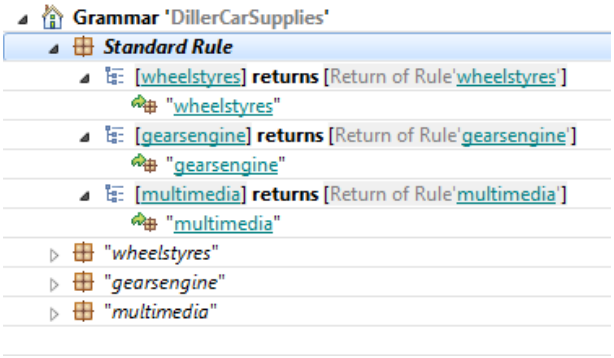


Figure 38: All Recognitions References

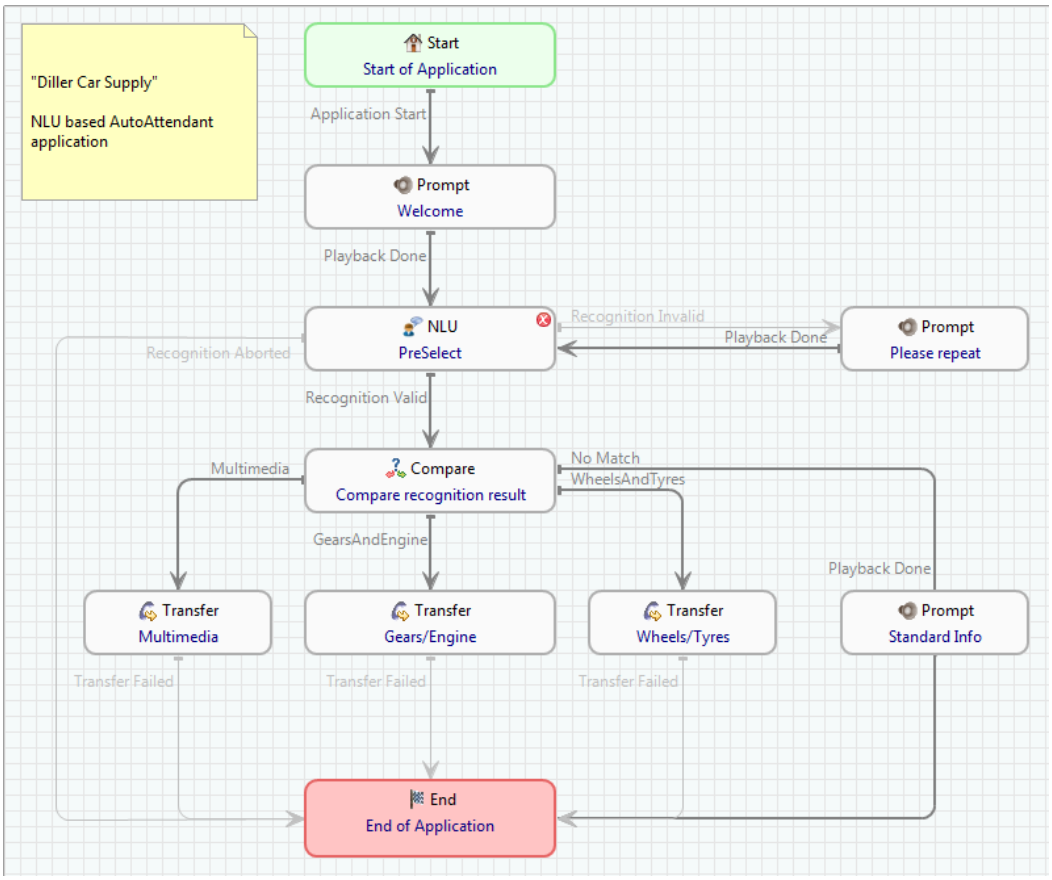
8.10.8. Define Semantic Result as Grammar Return Value

By now our grammar makes use of all previously declared *Rules*. Therefore the grammar will recognize several different kinds of user utterances. It will recognize all user utterances defined in the *Rule* «wheelstyres» as well as all user utterances in the *Rules* «gearsengine» and «multimedia».

Although this sounds quite well, the grammar still does not completely fulfill the requirements we previously had defined. As we remember the definition of our first task once again, this grammar will be used to route the application call flow. This

means, that the application has to react to the user utterance which was recognized by the grammar. To achieve this, we have to define a kind of return value for this grammar which is the so-called *Semantic Result*.

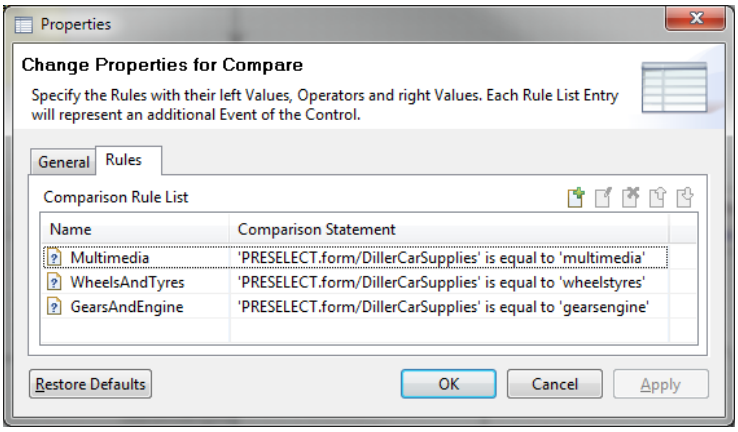
Before we start to update our grammar, we have to think about how this *Semantic Result* should look like. Please remember the application call flow which was already referenced in a previous chapter.



As you can see in this call flow, the sample application uses a NLU Control before a comparison of internal variables takes place to finally route the application call flow to one of the different Transfer Controls. In more details this means, that the application asks the user about a product out of three different categories first. If the user response was recognized, the application has to check what product category the user response was about and then react to this user response.

As a conclusion, we can define, that our grammar has to return a value that indicates the category of the product, which the user gave as response.

Because our sample application already defines what this value has to look like, you should not select a value on your own. The following screenshot shows the properties of the *Compare Control* used by our sample application to route the call flow. This screenshot will display the values we have to use:



As you can see in the screenshot above, the Semantic Result of our grammar is expected to look like the following:

- «wheelstyres» for all products in the category *Wheels and tyres*
- «gearsengine» for all products in the category *Gears and engines*
- «multimedia» for all products in the category *Car hi-fi and multimedia*

If we want to add those values as results of our grammar, we will have to add several *Semantic Results* as grammar components.
If you did not spend time to review the detailed information about all supported grammar components in the Knowledge Base: Grammar Components, you will need to know that we have to edit an *Alternative* to define/create a *Semantic Result*. Each *Semantic Result* is only valid for the *Alternative* it is defined for.

As described above we have to edit the *Alternatives* of our previously introduced logically divisions. As you remember these division were represented by different *Rules*. Therefore we will edit all *Alternatives* in those *Rules* like the following:

- Add «wheelstyres» as Semantic Result of all Alternatives in the Rule named «wheelstyres»
- Add «gearsengine» as Semantic Result of all Alternatives in the Rule named «gearsengine»
- Add «multimedia» as Semantic Result of all Alternatives in the Rule named «multimedia»

Let us start with the first *Alternative* of the *Rule* named «wheelstyres». Please open the context menu by right-clicking the *Alternative* and select the option to edit the *Semantic Result* of this *Alternative*. This will open an further dialog. The following screenshot shows how this dialog will look like:

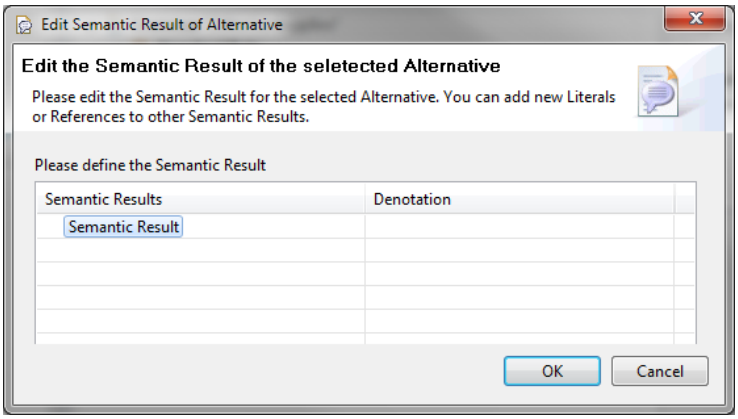
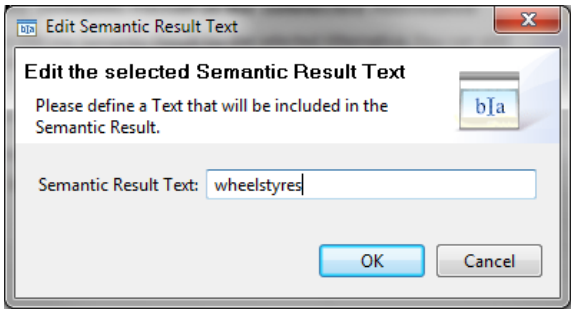


Figure 39: Dialog to edit the Semantic Result of an Alternative

Editing a *Semantic Result* will be done analog to the editing of the other grammar components. Please open the context menu of the node representing the *Semantic Result* by right-clicking this node. Within this context menu please select the option to add a text. This will open a further dialog allowing you to define a textual result like «wheelstyres». This screenshot will show you this dialog:



Adding this textual content to your Semantic Result will cause that this text content will be returned if this Alternative matches the recognition of a user utterance. The Semantic Result will then look like the following screenshot:

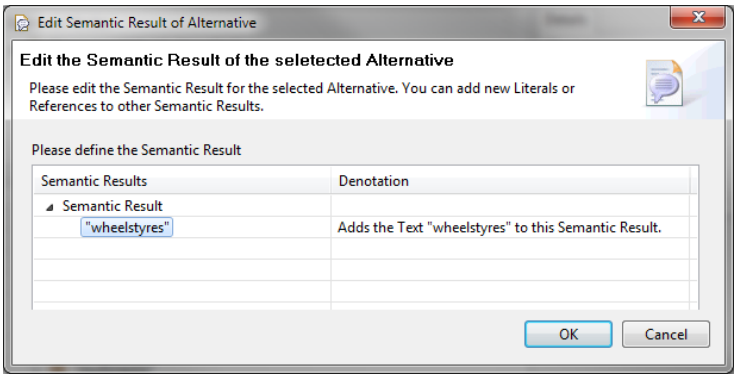


Figure 40: Semantic Result of an Alternative

If you apply this dialog, your grammar structure will be changed and the previously created *Semantic Result* will be added to the current *Alternative*. As you can see in the following screenshot the current *Alternative* indicates its *Semantic Result* with the keyword *…returns….*

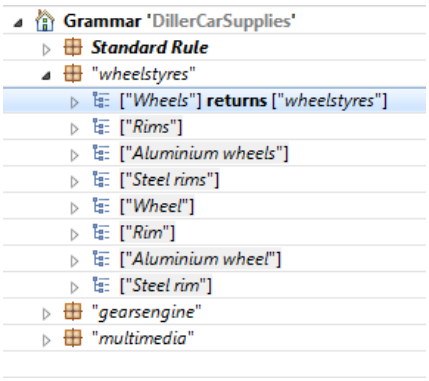
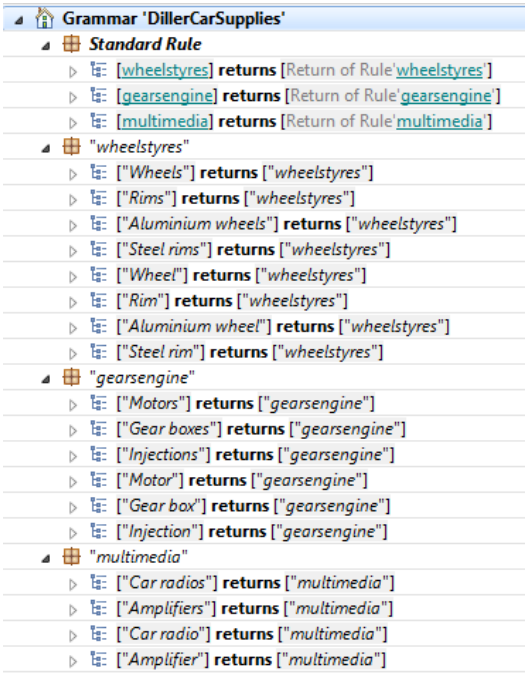


Figure 41: Alternative with Semantic Result

Please proceed analogue with all other *Alternatives* in the previously created *Rules*.



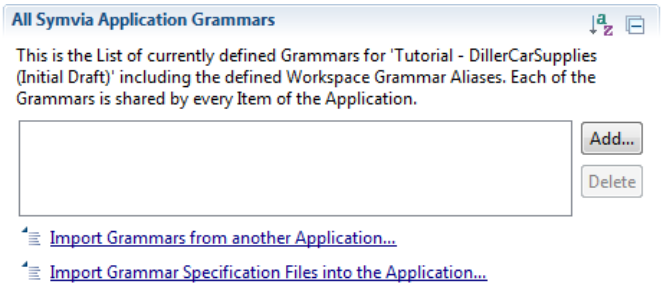
This screenshot illustrates how the grammar structure will look like after adding *Semantic Results* to all corresponding *Alternatives*.

…Congratulations! We have successfully done our first job! The final grammar «DillerCarSupplies» fulfills all the requirements we defined in an previous chapter.

8.10.9. Import Grammar in Application Builder

Although we have finished our grammar we still have to execute a final operation. Because we created this grammar with the *Grammar Studio*, we have to tell the *Application Builder* how to access it. Therefore **we need to switch to the** *Application Builder* **and import our previously created grammar as an** *Application Grammar* **to our sample application**.

Please open the *Application Grammars* within the *Application Builder* and select the option to add a new one. This screenshot illustrates the corresponding dialog.



This will open a further dialog requesting some properties for the new Application Grammar which are needed in the context of the Application Builder. These properties include a name, a description and the reference to a grammar file. This grammar file reference represents the connection to our grammar which we created in this tutorial. Therefore the file reference needs to be set to our previously created grammar. The following screenshot shows how this dialog will look like.

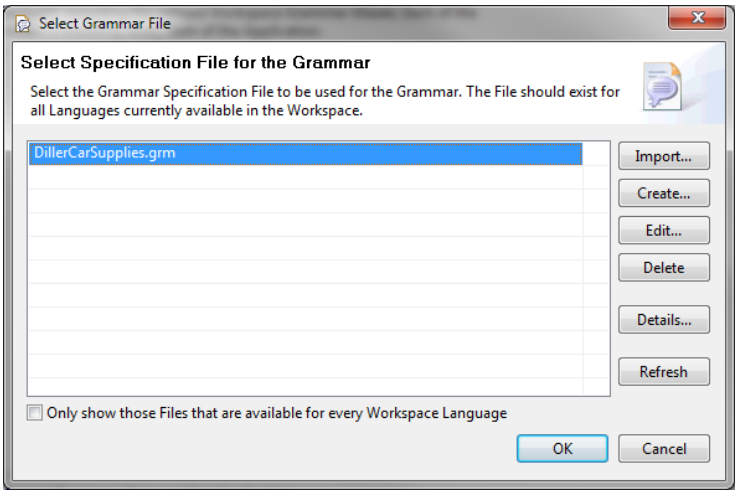


Figure 42: Select Grammar File

After we selected our grammar named «DillerCarSupplies.grm» as file reference for the *Application Grammar* the *Application Builder* is able to access our grammar. The following screenshot displays all set properties of the *Application Grammar* «DillerCarSupplies» which represents our grammar «DillerCarSupplies.grm» in the context of the *Application Builder*.

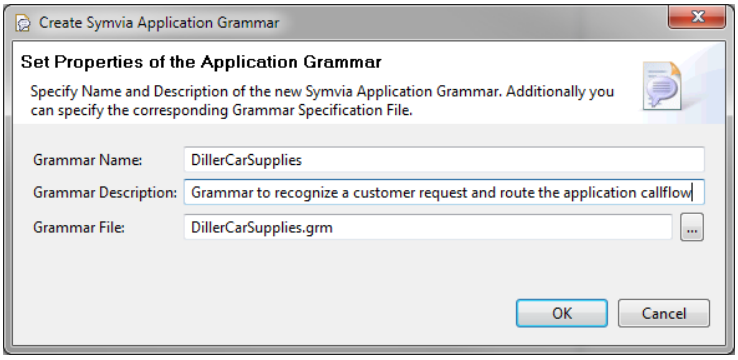
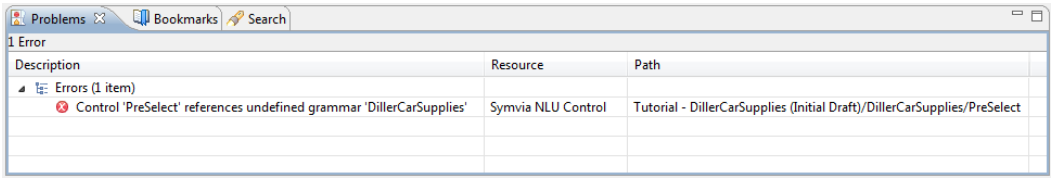


Figure 43: Add Grammar to Application Builder

Finally the *Application Builder* is now able to access our grammar via this added kind of facade. This will trigger the *Application Builder* to update its reported problems. As you remember there was a reported error because an *Application Grammar* named «DillerCarSupplies» was missing. By importing our grammar into the *Application Builder* we solve this problem:



8.11. Tutorial Step #9: A deeper insight into the Sample Application

After we successfully created a grammar to recognize the utterances of potential customers with the *Grammar Studio* and finally imported this grammar into the *Application Builder* Workspace what is next on our list of tasks?

8.11.1. What is missing?

Well, the created grammar does actually recognize only some single key phrases that a customer could use to communicate with this sample application. And in fact the recognition of those key phrases or words is sufficient enough to allow the desired application call flow routing according to a specific user utterance. **But there is still something missing**. Do you have an idea what this could be?

One very important aspect is not supported yet by our sample application. This aspect is the understanding of *natural* language. Natural language means in this case that our sample application should understand a customer who speaks like he does in everyday life.

Would you suggest customers only speaking in single (key)words? If you think of everyday life you probably would not only reply a single key phrase like «car radios!» on a question asking on which topic you would like more information about. In

normal conversations you perhaps would answer something like «I would like to have more information about car radios, please».

8.11.2. What is a customized Garbage Grammar?

The grammar which we created in the last chapter only recognizes single key phrases. If we want to support the understanding of (more) natural language, we have to enhance the application in relation to the recognition of user utterances in more natural expressions.

Supporting the understanding of natural language can again be achieved on several different ways. One possibility would be to simply edit the previously created grammar and enhance it in the requested way.
Another more effective (and a little more cooler) possibility is to introduce some additional so-called **(customized)** *Garbage Grammars*.

Before we go on with the definition of *Garbage Grammars*, let me first give you a structural overview that you will need later on.

To support *Natural Language Understanding* (NLU) the *Application Builder* makes use of the so-called *Dialog Engine* (DIANE) which comes along with a set of so-called *Standard Grammars*. These *Standard Grammars* **define a set of grammars which can be re-used by any other application to recognize a set of basic expressions** like e.g. time data or dates. Furthermore these *Standard Grammars* also include some basic *Garbage Grammars*.
Please review chapter *Knowledge Base: Application Builder Workspace as Repository Folder* on page 129 to get more details about Standard Grammars.

By now you probably will ask yourself, what these already mentioned *Garbage Grammars* are and what is meant by customized ones? Let us start with the differentiation between **customized** and **standard** *Garbage Grammars*. **Customized** *Garbage Grammars* **are just personal refinements or improvements of some standard** *Garbage Grammars*, which are supported by the *Dialog Engine* (DIANE) within its package of *Standard Grammars*. Please regard that we will use the expression «Garbage Grammar» as a synonym for customized or standard ones in the following sections.

Now let us take a look when (standard or customized) *Garbage Grammars* are used and what they are used for. *Garbage Grammars*, are grammars which will be activated in the DIANE environment in addition to the main grammar handling the recognition of key phrases. This means that the recognition of user utterances will not be done by one single grammar for its own, but it will be done by one recognizing grammar and at least one or more customized *Garbage Grammars*.

The recognizing grammar takes care of the key phrases and returns some *Semantic Result* whereas customized *Garbage Grammars* are only responsible to enable the understanding of all other parts of an utterance. These additional parts of an user utterance do not have any key phrases and therefore represent a kind of filling waste (*garbage*).

Summarized once again: *Garbage Grammars* **contain no key phrases in any application context. Their content represent only a kind of filling waste. As a conclusion** *Garbage Grammars* **do only filter their defined filling waste. Therefore** *Garbage Grammars* **do not return anything useful that could be used to route an application call flow.** Finally this means that (customized) *Garbage Grammars* do not have any *Semantic Result*.

The following figure illustrates the distinction between a key phrase and some additional (garbage) parts of an utterance in the context of our sample application. The figure also shows how to get logical fragments out of a phrase in principle and how these fragments can be categorized.

Original Phrase	«I would like to have details about car radios, please!»		
LOGICAL FRAGMENTS	«I would like to have details about»	«car radios»	«please»
Leading part[2]	«I would like to have details about»		
Key phrase[3]		«car radios»	
Inner part[4]			«please»

Figure 44: Distinction between key phrase and additional parts of an user utterances

As you can see in figure 1.1 (customized) *Garbage Grammars* are used to support the understanding of leading or inner parts of an user utterance which do no contain any key phrases.

As a best practice grammars should be designed in a way that they can easily be enhanced in future. Transfered to the example shown in figure 1.1 this means that there would be one customized *Garbage Grammars* defining several synonymous leading parts and another customized *Garbage Grammars* defining several synonyms usable as inner parts.

8.11.3. Definition of your Second Job

As a final conclusion our second job will be to **create customized** *Garbage Grammars*. These grammars will filter out filling waste contained in user utterances. This filtering will support the understanding of a more natural language spoken in complete sentences. When creating the customized *Garbage Grammars* we should take the following aspects into regard:

- There will be **two** *Garbage Grammars*
- There is a need to create a customized *Garbage Grammar* **for leading garbage**
- There is a need to create a customized *Garbage Grammar* **for inner garbage**
- Each *Garbage Grammar* should contain **several synonyms**

8.12. Tutorial Step #10: Create a customized Garbage Grammar using the Application Builder only

Now we are ready to start creating the customized *Garbage Grammars* which were described as our second job in the previous chapter. As mentioned before there are several ways to do so. Our first job was done by using only the *Grammar Studio*. This time we will do different and use the *Application Builder* as our only editor.

[2] Leading part of an utterance without any key phrase indicating garbage

[3] Key phrase in context of our sample application

[4] Inner part of an utterance without any key phrase indicating garbage

8.12.1. Reasons for using the Application Builder as Grammar Editor

Using the *Application Builder* is suggested in the following use cases:

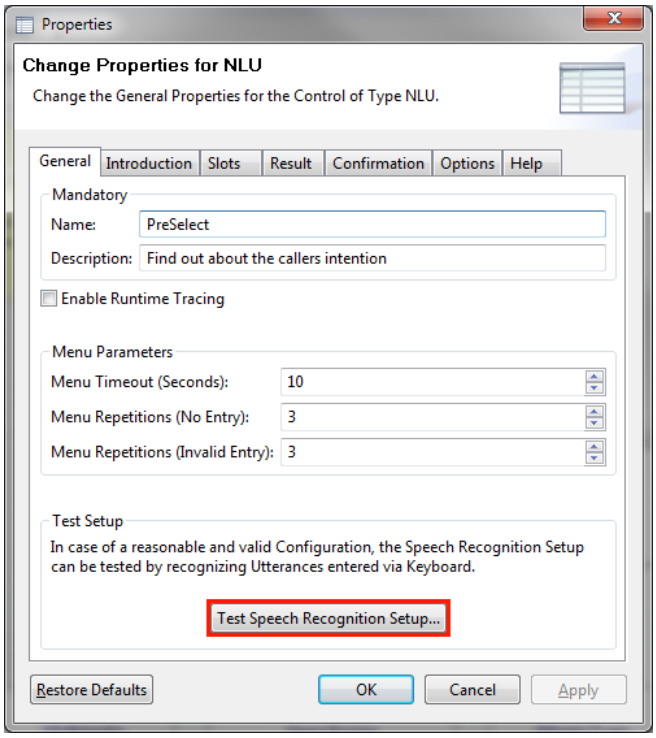
- You want to create a customized Garbage Grammar
- The grammar does not re-use other non-standard grammars
- The grammar structure is not important

In this case the grammar to create will be a customized *Garbage Grammar*. Therefore we should use the *Application Builder*, because this is the most efficient way to do it.

As you remember our second job includes the creation of two customized *Garbage Grammars*. One grammar will be used for leading garbage and one grammar will be used for inner garbage. In this chapter we want to create only one of this. Let us begin with the customized *Garbage Grammar* for leading garbage.

8.12.2. Use NLU Control to create a customized Garbage Grammar

The *Application Builder* offers an implicit way to create such a customized *Garbage Grammar*. This implicit way is hidden in the *NLU Control*. Please check out the *NLU Control* in our sample Application.
If you double-click this *NLU Control* the control properties will be opened as an own dialog. As you can see on the first tab of this dialog the *NLU Control* offers an option to test the speech recognition setup. This following screenshot shows how to find this option.



Please remember yourself that the *NLU Control* supports the activation/usage of two optional *Garbage Grammars* in addition to one mandatory grammar handling the recognition of key phrases. If the *Garbage Grammars* for leading and inner garbage are defined within the *NLU Control*, all three grammars will be activated and used to recognize user utterances.

By default each *NLU Control* will already make usage of standard *Garbage Grammars* for leading and inner garbage. These *Standard Grammars* are a very generic definition of possible parts of an user utterance containing no key phrases. In general an application designer should create own customizations of those standard *Garbage Grammars*. These so-called customized *Garbage Grammars* should define some more application-specific garbage.

Please review this previous chapter once again for more detailed information about the definition of *garbage* in general, of *Garbage Grammars* at all and about the difference between customized and standard ones.

8.12.3. Test user utterances to create a customized Garbage Grammar

Now, if you select the option to test the speech recognition setup on the first tab of the *NLU Control*, a new dialog will be opened. This dialog can be used for testing purposes, because it allows you to simulate if possible utterances would be recognized by the set of grammars defined by the *NLU Control*.

Fortunately we already spent a lot of time in the design of our (main) grammar (named «DillerCarSupplies.grm») handling the recognition of key phrases, which we realized as our first job in this tutorial. Therefore we will use this testing dialog only to check what application-specific garbage content is not recognized by the standard *Garbage Grammars*, which are activated by the *NLU Control* by default. The following screenshot displays the successful recognition results when we test one of our key phrases like «Wheels».
Please regard that all test inputs will be case-sensitive. This means that your test input should exactly match the inputs shown in this tutorial.

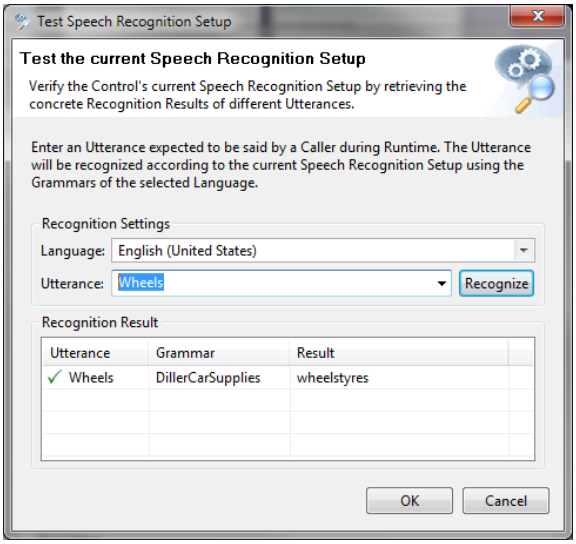


Figure 45: Successful Recognition of a Key Phrase

8.12.4. Add expression as leading Garbage Content

As we can see in the screenshot above key phrases like «Wheels» will be already recognized. In fact that means that our main grammar (named «DillerCarSupplies.grm») works fine. However this is not sufficient at all and does not support the understanding of natural language. If someone asks you on which topic you would like to have more information about, you probably will not answer «Wheels» but for example «I am looking for Wheels».
If you try this possible user utterance in the testing dialog, you will get the result that the leading part «I am looking for» is not recognized by any active grammar. This would be a good first candidate for our customized *Garbage Grammar* handling leading contents, don't you think? The following screenshot shows the unsuccessful recognition of non-defined leading garbage.

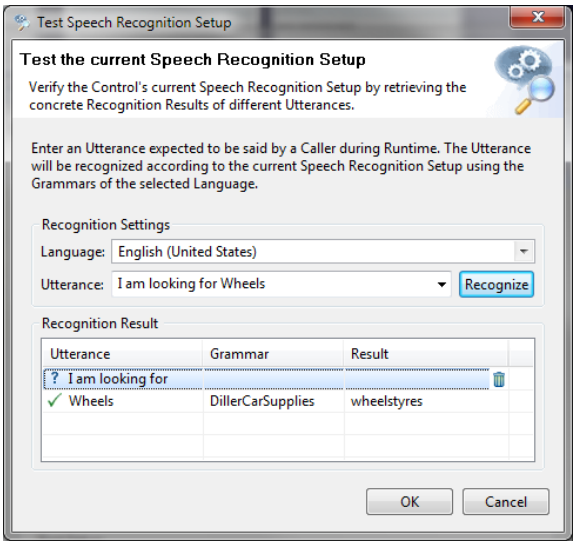


Figure 46: Unsuccessful Recognition of non-defined leading Garbage

So, let us add the expression «I am looking for» as an application-specific leading garbage. Simply use the garbage can symbol to define the selected expression as garbage. This will open a further dialog requesting you to select if this expression should be regarded as leading or inner garbage. The following screenshot illustrates how this dialog will look like.

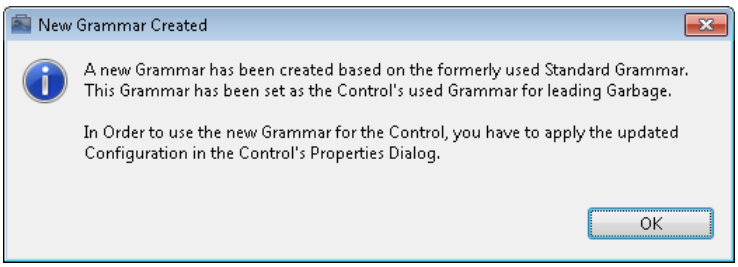
Figure 47: Add expression as leading Garbage

8.12.5. Implicitly create customized Garbage Grammar for leading Garbage

Adding the first application-specific garbage content as leading garbage content, will implicitly create a customized Garbage Grammar for leading garbage. The Application Builder will do this for you after you have been asked for some additional information like grammar name and description. Please remember that this customization is a kind of specialization of the standard Garbage Grammar for leading contents. Your specialization is needed to recognize your application-specific garbage contents. Any further application-specific leading garbage content will now be added to this customized Garbage Grammar. The following screenshot shows how this dialog will look like and which values we have to choose for our first customized Garbage Grammar.

Figure 48: Create customized Garbage Grammar for leading Garbage

After the grammar file was successfully created the *Application Builder* will confirm this and inform you that the properties of the *NLU Control* has been updated and that the *NLU Control* needs to be reloaded. Therefore you can close the dialog of the *NLU Control* properties and reopen it again. This screenshot illustrates how the confirmation dialog will look like.



8.12.6. Add further leading Garbage

Of course we will have to add more application-specific expressions as leading garbage to support *Natural Language Understanding*. **Please add at least the following expressions analog to the one shown above**:

- «I am looking for» *(Already done as example)*
- «I need»
- «I require»
- «I want»
- «I would like to»
- «I have a questions regarding»
- «Do you have»
- «Help regarding»

After you have added the expressions listed above as leading garbage you have successfully created an customized *Garbage Grammar* for leading garbage content.

Let us now continue with the second customized *Garbage Grammar* which will handle inner garbage content. Certainly you could do this analog to the grammar we created in this chapter, but of course there is also another way to do so. This will be illustrated in the next chapter.

8.13. Tutorial Step #11 - Create a customized Garbage Grammar using the Application Builder and the Grammar Studio

After we have successfully created a customized *Garbage Grammar* for leading garbage with the *Application Builder*, we still have to create the second customized *Garbage Grammar* which handles inner garbage.
Please remember yourself that we used the *Grammar Studio* to create our first grammar. In the previous chapter the first customized *Garbage Grammar* was created by using the *Application Builder* as editor and now we finally will use both applications in cooperation.

8.13.1. Reasons for using the Application Builder and Grammar Studio in Cooperation

As described above we will start creating a grammar with the *Application Builder* and then edit/finalize it in the *Grammar Studio*. This way is suggested if a grammar was estimated to be a very simple one while the complexity increases steadily when editing it. Therefore it is a good idea to switch the editor and use the *Grammar Studio*, if the requirements of a grammar have increased after the creation process was started.

8.13.2. Test user utterances to create a customized Garbage Grammar

First of all we use the *Application Builder* to implicitly create an application-specific customized *Garbage Grammar* for inner garbage contents. Therefore we again use the *NLU Control* and its feature to test the speech recognition setup like it is described in the previous chapter.

Now let us start to add a first expression as inner application-specific garbage, which will implicitly create a customized *Garbage Grammar* and update the *NLU Control* to use this newly introduced grammar.

What about the expression «I want new Wheels» when thinking of phrases containing inner garbage? Just perform a test with this expression like it was described in the previous chapter. As a result you will see that only the part «new» will not be recognized. All other parts are already recognized by the main grammar (named «DillerCarSupplies.grm») and our first customized *Garbage Grammar* for leading garbage content. The following screenshot shows how the test result should look like:

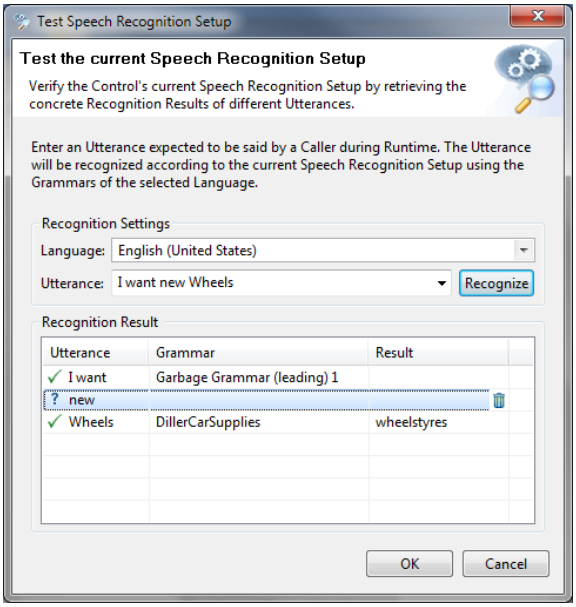


Figure 49: Unsuccessful Recognition of non-defined inner Garbage

8.13.3. Add expression as inner Garbage Content

So, let us add the expression «new» as an application-specific inner garbage. As already mentioned in the previous chapter please do simply use the garbage can icon to define the selected expression as garbage. This will open a further dialog requesting you to select if this expression should be regarded as leading or inner garbage. Please regard to add the expression as inner garbage this time like it is shown in the following screenshot.

Figure 50: Add expression as inner Garbage

8.13.4. Implicitly create customized Garbage Grammar for inner Garbage

Adding this first application-specific garbage content as inner garbage content, will implicitly create a customized *Garbage Grammar* for inner garbage. As described in the previous chapter the *Application Builder* will do this for you after you have been asked for some additional information like grammar name and description. The following screenshot shows how this dialog will look like and which values we have to choose for this second customized *Garbage Grammar*.

Figure 51: Create customized Garbage Grammar for inner Garbage

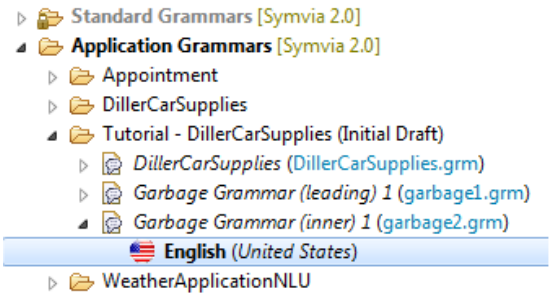
After the grammar file was successfully created the *Application Builder* will confirm this and inform you that the properties of the *NLU Control* has been updated and that the *NLU Control* needs to be update. Therefore you have to close the dialog of the *NLU Control* properties and reopen it again.

8.13.5. Add further inner Garbage by using the Grammar Studio

By now we add the first application-specific inner garbage and thus created implicitly a customized *Garbage Grammar* for inner garbage contents. In the previous chapter we used the *NLU Control* and its feature to test the speech recognition set-up to add all other possible expressions. This was a very easy way.

As an alternative we will try to switch to the *Grammar Studio* now to edit the previously created customized *Garbage Grammar* for inner garbage. This will illustrate once more how these two applications work together.

Please start the *Grammar Studio* which will load the *Application Builder* workspace as *Repository Folder*. Within your repository please open your *Application Grammars* then choose our sample Application «Tutorial - DillerCarSupplies (Initial Draft)» and finally select the current customized *Garbage Grammar* for inner garbage («garbage2.grm») in American English. The following screenshot will help you to find your way through the repository.



If you load this language-specific grammar into the editor by double-clicking the selected node, you will see the grammar structure of our customized *Garbage Grammar*. The following screenshot displays how this grammar structure should look like.

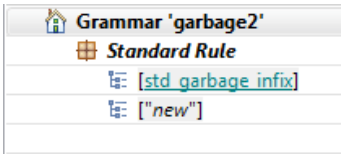


Figure 52: Structure of customized Garbage Grammar for inner Garbage Content

As you can see in the previous screenshot this grammar structure already contains one *Alternative* which will recognize the utterance «new». This was implicitly done by the *Application Builder* when we add this expression as garbage content.

We will now continue to add further application-specific inner garbage contents manually by using the *Grammar Studio*. The following list contains some further possible application-specific inner garbage contents:

- «new» (Already done as example)
- «a»
- «an»
- «some»
- «to buy»
- «to get»
- «to repair»

As you probably remember from this chapter («Step 08») we have to edit the grammar structure for every application-specific inner garbage content:

- **Step A1**: Add an *Alternative* to a *Rule*
- **Step B1**: Add a *Recognition* of the garbage content to the previously created *Alternative*

Please do this for all application-specific inner garbage contents analog to the proceeding described within this chapter («Step 08»). If you have successfully edited the grammar structure in this way, it will look like the following screenshot.

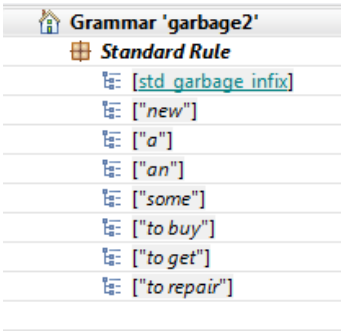


Figure 53: Final Structure of customized Garbage Grammar for inner Garbage Content

…Congratulations! We have successfully done our second (and last) job! Both customized *Garbage Grammars* were successfully created in different ways and both *Garbage Grammars* are activated in the *NLU Control* of our sample application. If those *Garbage Grammars* are activated in addition to our main grammar (named «DillerCarSupplies.grm»), the application call flow can be routed even if potential customers make use of natural language.

Of course our *Garbage Grammars* currently describe only a small set of leading/inner parts. To support a good understanding of natural language, some more expression should be added. It would be your turn now to enhance these grammars and drive them to perfection.

However you can now deploy and test our sample application with the help of the *Application Builder*. Therefore the *Media Server Starter Kit* will come along with the OpenScape Media Server which represents the application host in this scenario. After the application is successfully deployed on this application host you will be able to call your sample application via a SIP softphone.

8.14. Tutorial Step #12: Verify your Work

If you want to verify your created grammars and compare your final sample application against a sample solution, you can download our solutions in the Fusion Developer Portal:

Download a zip file named as «***tutorial_dillercarsupplies_finalversion.zip***» which represents an *Application Builder Archive File*. This archived package contains finally our sample application named «Tutorial - DillerCarSupplies (Final Version)».

- Download the Grammars as sample solution:
- Download a zip file named as *«tutorial_dillercarsupplies_grammars.zip»* which contains all the grammars which were created within this tutorial.

9. The Knowledge Base

This section was created to offer detailed background information on specific topics. The knowledge base contains the following chapters:

- General Concepts
- Grammar Logic
- Grammar Components
- Improve Grammar
- Understanding Semantic Results
- Application Builder Workspace as Repository Folder

9.1. Knowledge Base: General Concepts / Glossary

This knowledge base chapter will try to collect all general concepts which are used through out this tutorial. Every definition needs a kind of subsumption in the context of the Grammar Studio. The following terms are classified in alphabetically order:

- **Alternative**: The term *Alternative* describes a grammar component
- **Application**: In this context the term *application* will be used as a synonym for *IVR Application* which means an application with *Interactive Voice Response*.
- **Customized Garbage Grammar**: See «Garbage Grammar»
- **Dialog Engine (DIANE)**: The term *dialog engine* represents a platform for understand natural language.
- **DIANE Environment**: See «Dialog Engine (DIANE)»
- **DIANE Runtime**: See «Dialog Engine (DIANE)»
- **External Reference**: See «Recognition Reference»
- **Garbage Grammar**: The term *Garbage Grammar* represents a grammar which only recognizes user utterances without returning any *Semantic Result*
- **Grammar**: The term *grammar* will be used as a synonym for a Grammar file used by the *Dialog Engine (DIANE)*.
- **Internal Reference**: See «Recognition Reference»
- **Natural Language Understanding (NLU)**: The expression of *Natural Language Understanding* means the understand of language which is similar to the everyday language used in dialogs between human beings
- **Recognition**: The term *Recognition* describes a grammar component
- **Recognition Reference**: The term *Recognition Reference* describes a grammar component
- **Rule**: The term *Rule* describes a grammar component
- **Semantic Result**: The term *Semantic Result* describes a grammar component

9.2. Knowledge Base: Grammar Logic

This knowledge base chapter will try to explain what is meant by a grammar in this context and how a grammar will work logically. This understanding can be helpful for novices which just have started with creating/editing grammars.

9.2.1. What does Recognition mean?

First of all a grammar is designed to define what user utterances will (and can) be recognized (by an application using this grammar). In most cases there will be different possibilities of user utterances that have to be recognized by a grammar. This means that a grammar will contain several alternative user utterances, which can also be nested into each other.

The process of recognition in the context of a grammar can therefore be seen as a matching of a previously defined text value representing a potential user utterance to a practically given utterance which is also given as a text value.

If a grammar recognizes an user utterance one of these nested alternatives matches. Please image that a grammar does the process of recognition by offering different possible *flows of recognition*. If one of these flows matches, the grammar has recognized an user utterance.

9.2.2. Is this a recognizable user utterance?

If you create a grammar which should recognize three names (e.g Adam, John and Susan), each name is one possible *flow of recognition* through this grammar.

If the grammar has to recognize one of those three names mentioned above, the grammar will recognize the name, because there is a matching *flow of recognition*. But if the grammar has to recognize another name than those three, the grammar will not recognize it. There is no matching *flow of recognition*.

The first aspect of a grammar could therefore be summarized in this question: *Is this a recognizable user utterance?* A grammar therefore will be used by applications to define possible user dialogs

.

9.2.3. What is the meaning of an utterance?

But a Grammar does not only specify possible user utterances. It (optionally) can also give a kind of return value according the recognized utterance. If there is a recognizable user utterance someone could be interested what this utterance was or what this utterance will mean for the further proceeding. In fact most speech-enabled applications use grammars not only to define a possible user dialog but also want to react to the recognized utterance.
Therefore the grammar needs to define a kind of result value that will be called a *Semantic Result* from now on. Such *Semantic Results* can be seen as a return values representing the recognized user utterance (or anything else, depending the logic of a grammar). However these *Semantic Results* has to be specified by the grammar designer as well as the possible *flow of recognitions*.

The second aspect of a grammar can therefore be summarized in the question: *What is the meaning of a utterance (if it was a recognizable one)?* Thus a grammar can be used by application to allow a reaction to a recognizable user utterance.

9.2.4. Conclusion

As a conclusion we can sum up that a speech-enabled application which uses a grammar will always be able to determine if an user utterance is recognizable. Only if the grammar has some kind of *Semantic Result* the application will be allowed to react to the recognized user utterance.

Grammars which do not offer any *Semantic Result* will (sometimes) be called *Garbage Grammars*. *Garbage Grammars* therefore defined as grammars that only determine if an user utterance is recognizable. Because this does not allow any reaction to a potential user utterance, the recognized utterance can be seen as a kind of garbage. These utterances will be lost for any application, because no reaction is possible.

9.3. Knowledge Base: Grammar Components

This knowledge base chapter will give you some background information about the different components of a grammar which you will need to start editing a grammar. Although the *Grammar Studio* tries to reduce the complexity of editing a grammar, the user should know the meaning and purpose of these grammar components.

9.3.1. Components Overview

There are different components of grammar each having its own meaning and purpose. Grammar components can be differentiated in explicit and implicit ones.

On the one hand, *explicit components* are directly shown in the grammar structure and can directly be edited. On the other hand, *implicit components* are not directly accessible. In most of the cases there will be a dedicated editor dialog to change those components.

The following *implicit components* are available:

- Rules
- Standard Rule
- Alternatives
- Recognitions
- Recognition References
- *Internal* References
- *External* References

The following *explicit components* are available:

- Comments
- Semantic Results

9.3.2. Rules

Rules are basic components, because **each Grammar will have at least one** *Rule*. This *Rule* will be called *Standard Rule* (something also referred to as *Root Rule*).

Rules **are used to organize Grammars**. Instead of having a grammar with one *(Standard) Rule* containing dozens of child nodes, the user can create multiple *Rules* each having a dedicated meaning. This will increase the clearness of the grammar if it will be edited by another person later on.

Please regard that **rules are not allowed to be nested into each other** (although theoretically it would be a possible way). Allowing the nesting of *Rules* would it make more complicated to support a re-use of *Rules* within a grammar. In fact the internal reutilization of *Rules* is realized by a differentiation of the declaration and the usage/assignment. The declaration is strictly separated from the usage/assignment of a *Rule*. As you can see in the following screenshot all *Rules* are declared on one identically logical level, which does not describe the usage of a *Rule* (except the *Standard Rule* which can be seen as the entry point of a grammar).

Figure 54: Multiple Rules in a Grammar

If *Rules* are declared in a grammar they can be used by all other internal *Rules*. This finally supports the important internal reutilization of *Rules* without allowing to nest *Rules* into each other.

But how can already declared *Rules* now be used in another *Rule* of the same grammar? If you want to use an additionally declared *'Rule'* within your grammar, you will have to create an *Internal Reference* with your *Rule* as reference target. This topic of internal references will be discussed in more details in the section about *'Recognition References'*.

Please regard to differentiate between the declaration of a *Rule* and the usage/assignment of a *Rule* as an important aspect.

Furthermore *Rules* have at least two interesting properties. **Besides its name (which cannot be changed for the** *Standard Rule***), this is a comment**.

This comment will allow you to describe what the purpose of the associated *Rule* will be. Comments which refer to the Grammar as a whole should be created as a comment of the *Standard Rule*.

Rules **can also have child nodes. These child nodes are called** *Alternatives*. They will be discussed in the following section.

9.3.3. Alternatives

Each *Alternative* is a child node of a parent *Rule*, but there can be multiple *Alternatives* in one *Rule*. This can be tracked in the following screenshot

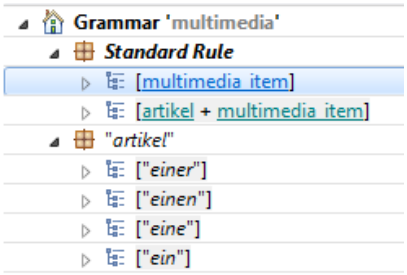


Figure 55: Multiple Alternative Nodes

The main purpose of an *Alternative* **is to define one possible** *flow of recognition*. Therefore the content of an *Alternative* (in other words: its possible child nodes) will define this *flow of recognition*.
Please see page 119, *Knowledge Base: Grammar Logic, f*or more background information on this topic.

Alternatives **can hold two different kinds of component types as child nodes**. Besides *Recognitions* they can include *Recognition References* as content. Both components are discussed in the following sections.
An *Alternative* **itself does not need a describing name**. They are identified by their content. Please regard that an *Alternative* is defined by the sequence of all its child nodes. Therefore the order of all its child nodes has to be regarded.
The importance of the order is discussed in more details in context of *Recognitions.*

Nevertheless *Alternatives* **contain another very important property**. **This property is the so-called** *Semantic Result*. In a nutshell, the *Semantic Result* of a *Alternative* defines what the return value will be, if there is a matching *flow of recognition*. The component *Semantic Result* will also be discussed in a later section.

9.3.4. Recognitions

'Recognitions' are relative simple grammar components, which can be added as child nodes to an 'Alternative'.

The purpose of a *'Recognition'* **is to describe an user utterance which the grammar will recognize**.
Theoretically this user utterance can be any text content. A single character is supported as well as a complete sentence for example. However you have to keep in mind, that the recognition has to match a potential user utterance. Therefore it is not recommended to define complete and complex sentences as one *Recognition*.
Please review chapter *Knowledge Base: Improve Grammar Recognitions (Best Practices)* on page 125 to get more information how to optimize the recognition of more difficult phrases.

Please regard that the position of a *Recognition* in the list of child nodes of a parent *Alternative* is an important factor. **If you change the position of a** *Recognition* **in its parent** *Alternative***, this will have effect on the** *flow of recognition*. The following example will illustrates this fact.

```
['Alternative']
```

```
|
+--['Recognition' of «this»] // 1. Child
+--['Recognition' of «is»]   // 2. Child
+--['Recognition' of «cool»] // 3. Child
```

Figure 56: Theoretical example of an Alternative with multiple Recognitions in given

If you swap the position of the first two *Recognitions* in the example above, you change the *flow of recognition* in the *Alternative* from *«this is cool»* to *«is this cool»*.

Please remember that an *Alternative* is defined by the sequence of all its child nodes. The following screenshot shows how the sample above would like in the *Grammar Studio*.

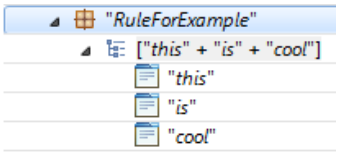


Figure 57: Alternative with multiple Recognitions in given order

In the context of the example shown above, an observing reader meanwhile probably could have asked himself, why there are three single-word-*Recognitions* instead of one *Recognition* defining the complete phrase. Of course, the example above should be optimized in practice to keep the clearness of the grammar structure. However there will be no difference in the recognition capacity. The more optimized version of the sample is shown in the following screenshot.

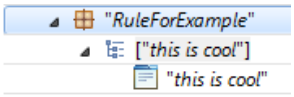


Figure 58: 'Alternative' with an united Recognition

Hint: The *Grammar Studio* explicitly offers you an option to automatically unite *Recognitions* in a sequence. Please see the corresponding option in the context menu of a *Recognition*.

9.3.5. Recognition References

'Recognition References' are the last grammar component to describe within this knowledge base chapter. They represent a reference to a recognition and they can be added as child nodes (only) to an *'Alternative'*.

The purpose of a *Recognition Reference* is to make usage of already existing *parts*. The phrase *«parts»* is intentionally kept unspecific here, because *«parts»* can mean different things. However a *Recognition Reference* specifies always the name of such parts as its reference target.

At this point we have to differentiate between *Internal References* and *External References*.

Internal References **describe a reference that refers to another** *Rule* **within the same grammar**. This *Internal Reference* therewith represents the usage/assignment of a *Rule* as it was already mentioned in the section about *Rules*.
In this case you will select the name of your desired *Rule* as a reference target. This will cause that the referenced *Rule* will be included into the *flow of recognition*.

External References mean something different. Please image the following situation: You already have another existing grammar handling some basics, which you could re-use in your current grammar. **When you want to reference this already existing Grammar, you will have to make use of an** *External Reference*.
In this case you will select the name of the already *existing grammar* as reference of your target. The will cause that the referenced grammar will be included into the *flow of recognition*, which will use the referenced Grammars *Standard Rule* as its entry point.

The following screenshot shows the usage of *Recognition References*. As you can see the *Rule* named *«multimedia_item»* is the target of two different *Recognition References*.

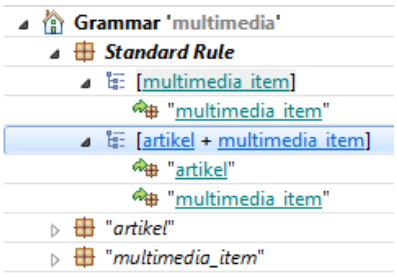


Figure 59: Internal References as Recognition References

Please regard that the position of a *Recognition Reference* in the list of child nodes of its parent *Alternative* is also an important factor. **If you can change the position of a** *Recognition Reference* **in the parent** *Alternative***, this will have effect on the** *flow of recognition*. This works analog to *Recognitions* (See previous section).

9.3.6. Comments

Comments **are optional** grammar components which can only **be added to a grammar by editing** *Rules*. Each *Rule* can have an specific *Comment*. If a *Comment* is valid for the complete grammar it should be added to the *Standard Rule*.

The purpose of a *Comment* **is to describe the purpose** of a Grammar or give an example of an utterance which will be recognized by the grammar. This information can be very useful, if there are more than one person working with the grammar.

Each *Comment* itself consists of different lines. Each line of a *Comment* has to be defined separately when using the *Grammar Studio*.

9.3.7. Semantic Results

Semantic Results **are optional** grammar components. **They can only be defined by edit the corresponding property of an** *Alternative*. Each *Alternative* can have an specific *Semantic Result*.

A *Semantic Result* describes the denotation of an *Alternative*. It represents a kind of return value which can (and in most cases will) be used by an application to react to a recognized utterance, if the associated *Alternative* describes the *flow of recognition*.

In other words: A *Semantic Result* specifies the return value if an user utterance matches the *Alternative*.

A *Semantic Result* can consists of two different types of content. The first possible content type is simple text (containing single characters, words, complete sentences or what so ever). The other content type represents references to other *Semantic Results* within the same *Alternative*.

But what can practically be done now with these different contents in a *Semantic Result*? If you need an answer to this question right now, please review the *Knowledge Base: Understanding Semantic Results* on page 126. This chapter will give a closer look into this grammar component and illustrates the handling with two (theoretical) examples.

9.4. Knowledge Base: Improve Grammar Recognitions (Best Practices)

This chapter gives you some hints how to improve a grammar recognition. This information can be seen as best practices. To increase the recognition of your grammar please regard the following best practices:

- Do not use any punctuation in your *Recognition*
- Announce numbers, time data or dates as they are spoken (e.g to recognize *«3»*, use *«three»*)

- Announce foreign expressions as they are spoken in the language of your grammar (e.g. use *«HaiFi»* to recognize the english word *«HiFi»* in a german grammar)
- If you want to spell phrases character by character, add an underline «_» after each single character (e.g. to recognize the phrase *«CD player»*, use *«C_D_player»*)

9.5. Knowledge Base: Understanding Semantic Results

This chapter tries to give some more details which should help the reader to understand the basic principals of *Semantic Results*.

9.5.1. Content

A *Semantic Result* represents a kind of return value which can (and in most cases will) be used by an application to react to the recognized utterance, if the associated *Alternative* describes the *flow of recognition*.

A *Semantic Result* can consists of two different types of contents. The first possible content type is simple text (containing single characters, words, complete sentences or what so ever). The other content type represents references to other *Semantic Results* within the same *Alternative*.

But what can practically be done now with these different contents in a *Semantic Result*? Let's use some examples to answer this question…

9.5.2. Example: Grammar «G1» without Semantic Results

Please image the following situation:

You have an grammar named *«G1»* which recognizes two different user actions (e.g. *«listen to»* and *«record»*) and two different objects for these actions (e.g. *«voicemail»* and *«email»*). Therefore the grammar will have one *Rule* handling the actions and one *Rule* which handles the objects.

Furthermore the grammar will be designed to recognize the reasonable permutations of these actions and objects (e.g. *«listen to voicemail»* and *«listen to email»* as well as *«record voicemail»* and *«record email»*). This will be done in the *Standard Rule*. Therefore there will be an *Alternative* which makes usage of the previously declared *Rules* in a reasonable permutation.

If someone had spend some time in writing down the structure of this grammar by hand (without using the *Grammar Studio*), this probably would look like the following figure.

```
[Grammar «G1»]
|
+--['Rule' for actions]
|   |
|   +--['Alternative']
|   |   |
|   |   +--['Recognition' of «listen to»]
|   |
|   +--['Alternative']
|       |
|       +--['Recognition' of «record»]
```

```
|
+--['Rule' for objects]
|   |
|   +--['Alternative']
|   |   |
|   |   +--['Recognition' of «voicemail»]
|   |
|   +--['Alternative']
|       |
|       +--['Recognition' of «email»]
|
+--['Standard Rule']
     |
     +--['Alternative']
        |
        +--['Recognition Reference' to «'Rule' for action»]
        +--['Recognition Reference' to «'Rule' for objects»]
```

Figure 60: Grammar Structure of Grammar «G1»

If this grammar would be used by an application, the possible user utterance «listen to email» would be recognized, because there would be a matching flow of recognition.
But how should the application react to this recognized user utterance? How should the application decide what action the user requested for which object? There is no way the application could this, because the grammar makes no use of any Semantic Result!

9.5.3. Example: Grammar «G2» with Semantic Results

Please image the following situation:

We have added Semantic Results to the grammar «G1» and saved this new grammar under the name «G2».

Therefore each Alternative in the Rules handling actions and objects has to be edited. Each Alternative will get an own Semantic Result which defines a text phrase as return value. The Alternatives containing the actions will return a string starting with «ACT_…», whereas the Alternatives containing the objects will return a string like «OBJ_…».

Finally the Alternative in the Standard Rule gets also a Semantic Result. Here the Semantic Result won't be a text phrase. In this case we make use of two references to other Semantic Results. The target of the first reference will be the Semantic Result of the Recognition Reference handling the actions and the second one references the Semantic Result of the Recognition Reference handling the objects.

Let's take a look at the structure of the grammar «G2». Please try to comprehend the changes compared to the structure of the grammar «G1».

```
[Grammar «G2»]
|
+--['Rule' for actions]
|   |
|   +--['Alternative' returns «ACT_Listen» as 'Semantic Result']
|   |   |
|   |   +--['Recognition' of «listen to»]
|   |
```

```
|   +--['Alternative' returns «ACT_Record» as 'Semantic Result']
|       |
|       +--['Recognition' of «record»]
|
+--['Rule' for objects]
|   |
|   +--['Alternative' returns «OBJ_VMail» as 'Semantic Result']
|   |   |
|   |   +--['Recognition' of «voicemail»]
|   |
|   +--['Alternative' returns «OBJ_EMail» as 'Semantic Result']
|       |
|       +--['Recognition' of «email»]
|
+--['Standard Rule']
     |
     +--['Alternative' returns the 'Semantic Results' of both child nodes as own 'Semantic Result']
          |
          +--['Recognition Reference' to «'Rule' for action»]     // 1. Child
          +--['Recognition Reference' to «'Rule' for objects»]    // 2. Child
```

Figure 61: Grammar Structure of Grammar «G2»

If this grammar would be used by an application now, the possible user utterance «listen to email» would be recognized, because there would be a matching flow of recognition.

Furthermore the added Semantic Results will enable each application which use this grammar to react to the recognized user utterance. In this example the return value of the flow of recognition through this grammar «G2» would look like «ACT_Listen OBJ_EMail».

Please regard that the exact return value depends on how the Semantic Results of both child nodes are concatenated to each other.

The application which makes use of this Semantic Result could now parse this return value and finally get the information what action the user requested with which object. So the grammar does not only specify a possible user utterance, but also does enables the application to react to the recognized user utterance.

9.6. Knowledge Base: Application Builder Workspace as Repository Folder

The chapter will give you some more details about the structure of the *Repository Folder* when using an *Application Builder* workspace.

9.6.1. Different Grammar Scopes

When you use an *Application Builder* workspace as *Repository Folder* for the *Grammar Studio*, you will have access to all your grammars you are working with.

If a grammar is part of an application the grammar will be called an ***Application Grammar***. If your grammar however is part of a composition it will be called a ***Composition Grammar***. In contrast ***Workspace Grammars*** define grammars which have been created to be used within the complete *Application Builder* workspace.

Finally ***Standard Grammars*** specify a basic set of grammars which is supported by the *Dialog Engine (DIANE)* environment. These *Standard Grammars* can be referenced by any other grammar without any restrictions and offer handle basic use cases like recognizing time data, dates, etc..

To sum this up, the *Application Builder* (and therefore also the *Grammar Studio*) differs between the following scope of grammars:

- Application Grammars
- Composition Grammars
- Workspace Grammars
- Standard Grammars

Each of these scopes will be represented in an own folder within the *Repository Explorer* of the *Grammar Studio*.

9.6.2. Influences of Grammar Scopes

These different grammar scopes will have influence on different aspects when working with the Grammar Studio. This section will show up those factors which will be influenced.

			External Reference…	
SCOPES	Listed	Editable	…Target	…Validation
Standard Grammars	yes	no	yes	yes
Workspace Grammars	yes	yes	no	no
Composition Grammars	yes	yes/no[5]	no/yes[6]	no/yes[7]

[5] no: Can be defined as read-only (in future developments)

[6] yes: Only for Grammars in the same Composition

[7] yes: Only for Grammars in the same Composition

Application Grammars	yes	yes	no/yes[8]	no/yes[9]

Figure 62: Scope Influences

All grammars in all different scopes will be listed in the Repository Explorer and can be opened to review or analyze those grammars.

With the exception of Standard Grammars all other grammars are editable. Standard Grammars are read-only, because they represent a basic set of grammars which is supported by the DIANE environment. In future developments Composition Grammars will be allowed to be read-only too. The column entitled «External Reference Target» shows if grammars of a specific scope can be used by other grammars as a target of an External Reference. Standard Grammars can be used as reference targets of course, but Workspace Grammars will not be allowed as possible targets. Composition Grammars can only be used as reference targets by grammars from the same composition but not by other grammars.

Application Grammars are similar to Composition Grammars in this case. They can also only be used as reference targets by grammars from the same application but not by any other grammar.

Finally the different scopes also influence the validation. The Grammar Studio will perform multiple checks if the opened grammar is valid or not. This check also includes the verification of External Reference targets.
In this context Standard Grammars will be included in the validation process. Workspace Grammars won't be included in the validation process of verifying external references, because they are no allowed reference targets.
Composition Grammars will only be included in the validation process if a Composition Grammar is referenced by another grammar from the same composition. In all other cases it won't be included in the validation process, because Composition Grammars are no allowed reference targets.

Application Grammars are similar to Composition Grammars again. They will also only be included in the validation process if an Application Grammar is used by another grammar from the same application.

[8] yes: Only for Grammars in the same Application

[9] yes: Only for Grammars in the same Application

10. Appendix: Frequently Asked Questions (FAQ)

Q: How to load a Repository Folder by command-line argument?

A: Please use the following command-line argument:
-repositoryFolder <path-of-repository-folder>

11. Appendix: Known Issues

- Due to the lack of an admin UI you have to delete a deployed application from the custom deployment folder ..\Unify\ms_starterkit\application_host\deployment-custom and redeploy it if you'd like to change the language at a given extension

12. Appendix: Application Samples of the Starter Kit

The following *applications* are installed with the Media Server Starter Kit:

Application	Description	Default Number	NLU application	Default Workspace
Simple IVR	A very simple IVR application	807	no	yes
Speaking Clock	Reads out the current system time to the caller	804	no	yes
Switch Language	Shows how to change the language of an application	805	no	yes
Weather Application[10]	Provides the current weather for some cities	806	no	yes
Weather Application[11] NLU	Same as Weather Application, but with a Speech interface	815	yes	yes
Appointment	Schedules a meeting	812	yes	yes
Diller Car Supplies	Auto Attendant based on NLU	816	yes	yes
Read The Meter	Collect data for an electricity supplier	814	yes	yes
Automatic survey	Application showing how to read/write into databases		no	no

The following *compositions* are installed with the Media Server Starter Kit:

Application	Description
Change Numeric Password	Changes the numeric password of a user
Login	Performs a login based on the users extension and numeric password

[10] Contained only in old versions of the StarterKit

[11] Contained only in old versions of the StarterKit

Formerly
Siemens Enterprise Communications

Harmonize your enterprise

unify.com