EventStudio System Designer 8

USER'S MANUAL

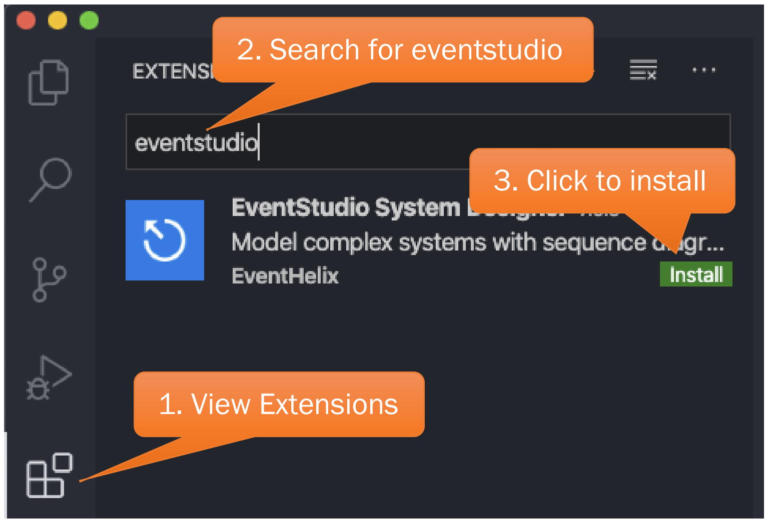
CONTENTS

1 GETTING STARTED

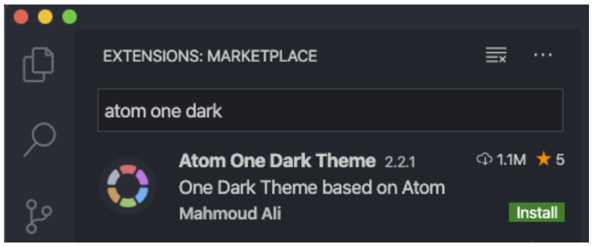
1.1 Download Visual Studio Code and EventStudio extension

EventStudio is implemented as an extension to the popular Visual Studio Code editor. To get started:

1. Download and install Visual Studio Code from: https://code.visualstudio.com/
2. Install the EventStudio extension from the extension menu.


2. Search for eventstudio
EXTENSI
eventstudio
3. Click to install
EventStudio System Designer
Model complex systems with sequence diagr...
EventHelix
Install
1. View Extensions

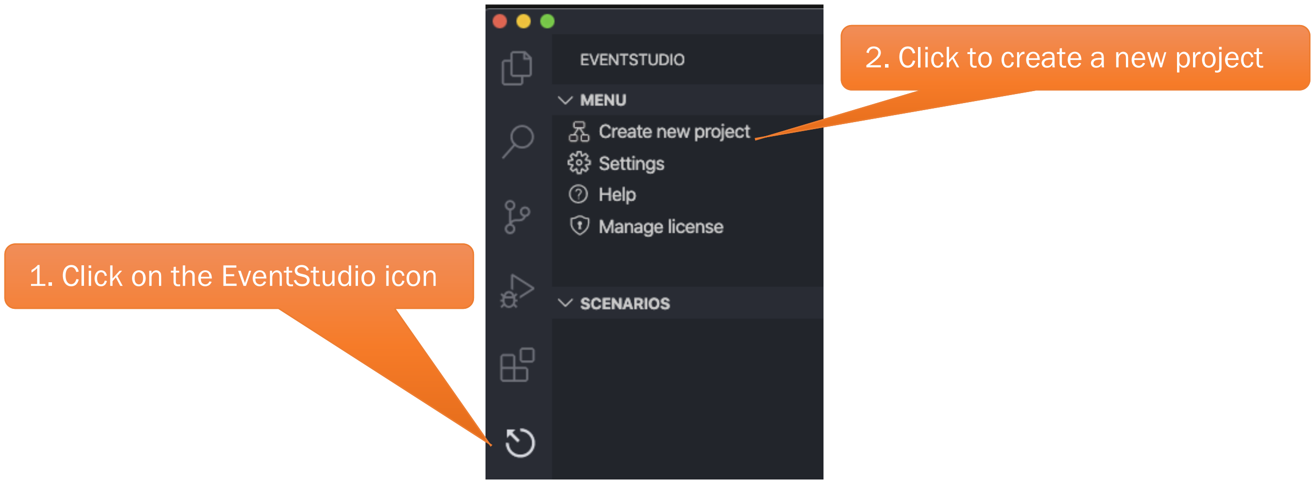
3. We also recommend:
   a. Using the Atom One Dark Theme.



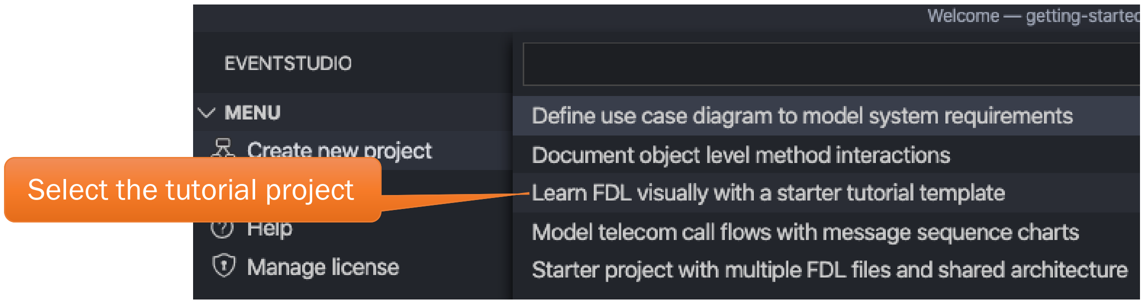
   b. Enabling Auto Save (`File → Auto Save`)
4. If you are new to Visual Studio Code, the following link should get you started: https://code.visualstudio.com/docs/getstarted/tips-and-tricks

1.2 Your First Project

1. Create a new folder named `getting-started` using Windows Explorer/Finder/Console.
2. Invoke Visual Studio Code and use the `File → Open Folder`[1] / `File → Open`[2] menu to open the `getting-started` folder.
3. Click on the EventStudio icon and then create a new project.



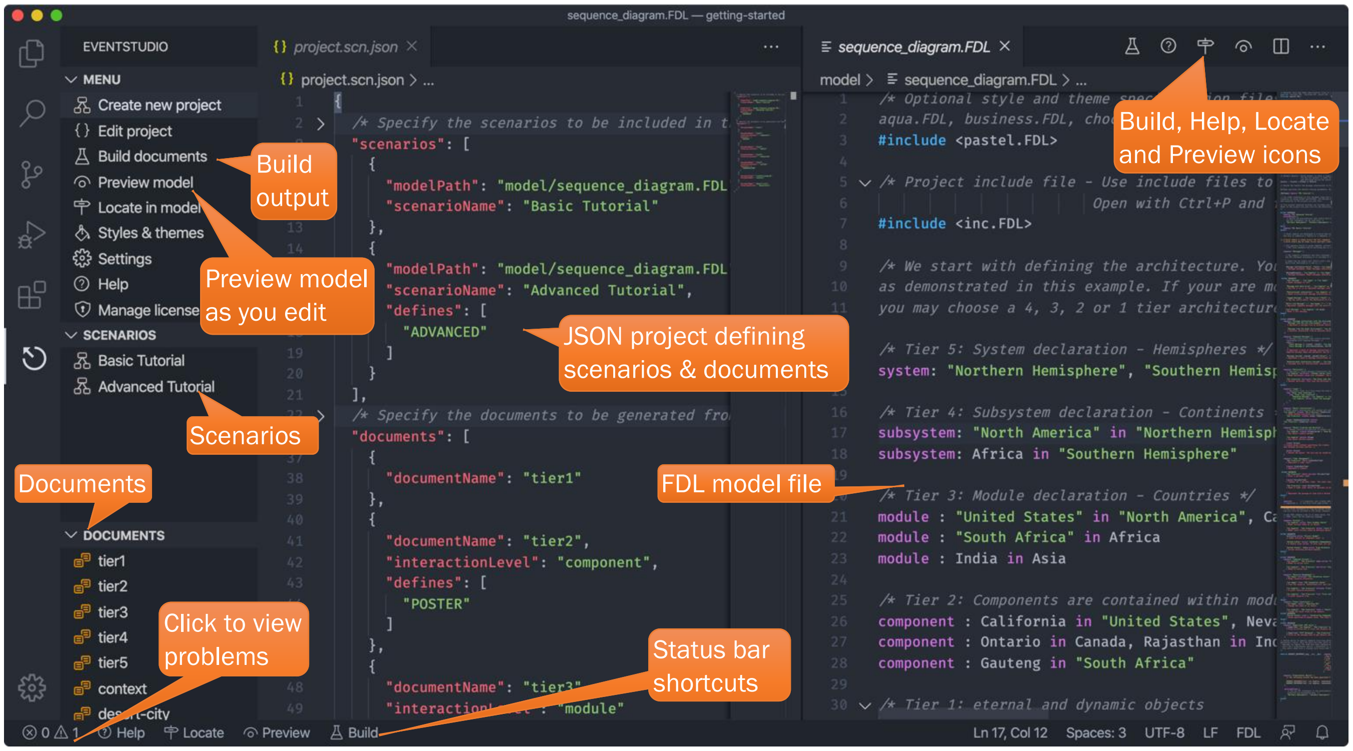
4. Select "`Learn FDL visually with a starter tutorial template`".



Define use case diagram to model system requirements
Document object level method interactions
Learn FDL visually with a starter tutorial template
Model telecom call flows with message sequence charts
Starter project with multiple FDL files and shared architecture

5. EventStudio sets up the project with the following files:
   a. `project.scn.json` defining the project scenarios and documents.
   b. `sequence_diagram.FDL` contains the FDL starter tutorial.

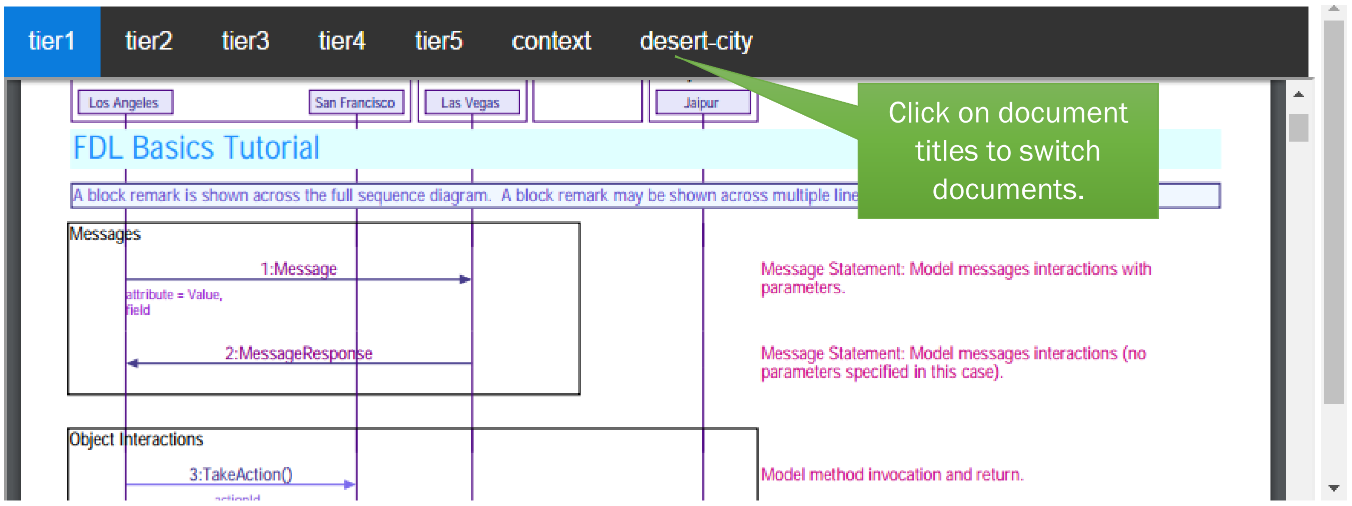
[1] Windows
[2] macOS



```
{
/* Specify the scenarios to be included in t
"scenarios": [
  {
    "modelPath": "model/sequence_diagram.FDL
    "scenarioName": "Basic Tutorial"
  },
  {
    "modelPath": "model/sequence_diagram.FDL
    "scenarioName": "Advanced Tutorial",
    "defines": [
      "ADVANCED"
    ]
  }
],
/* Specify the documents to be generated fro
"documents": [
  {
    "documentName": "tier1"
  },
  {
    "documentName": "tier2",
    "interactionLevel": "component",
    "defines": [
      "POSTER"
    ]
  },
  {
    "documentName": "tier3"
    "interaction         : "module"
```

```
/* Optional style and theme spec
aqua.FDL, business.FDL, cho
#include <pastel.FDL>

/* Project include file - Use include files to
                              Open with Ctrl+P and
#include <inc.FDL>

/* We start with defining the architecture. Yo
as demonstrated in this example. If your are m
you may choose a 4, 3, 2 or 1 tier architectur

/* Tier 5: System declaration - Hemispheres */
system: "Northern Hemisphere", "Southern Hemis

/* Tier 4: Subsystem declaration - Continents
subsystem: "North America" in "Northern Hemisp
subsystem: Africa in "Southern Hemisphere"

/* Tier 3: Module declaration - Countries */
module : "United States" in "North America", C
module : "South Africa" in Africa
module : India in Asia

/* Tier 2: Components are contained within mod
component : California in "United States", Nev
component : Ontario in Canada, Rajasthan in In
component : Gauteng in "South Africa"

/* Tier 1: eternal and dynamic objects
```

6. EventStudio automatically builds the project and displays the output in your default browser.



A block remark is shown across the full sequence diagram. A block remark may be shown across multiple line

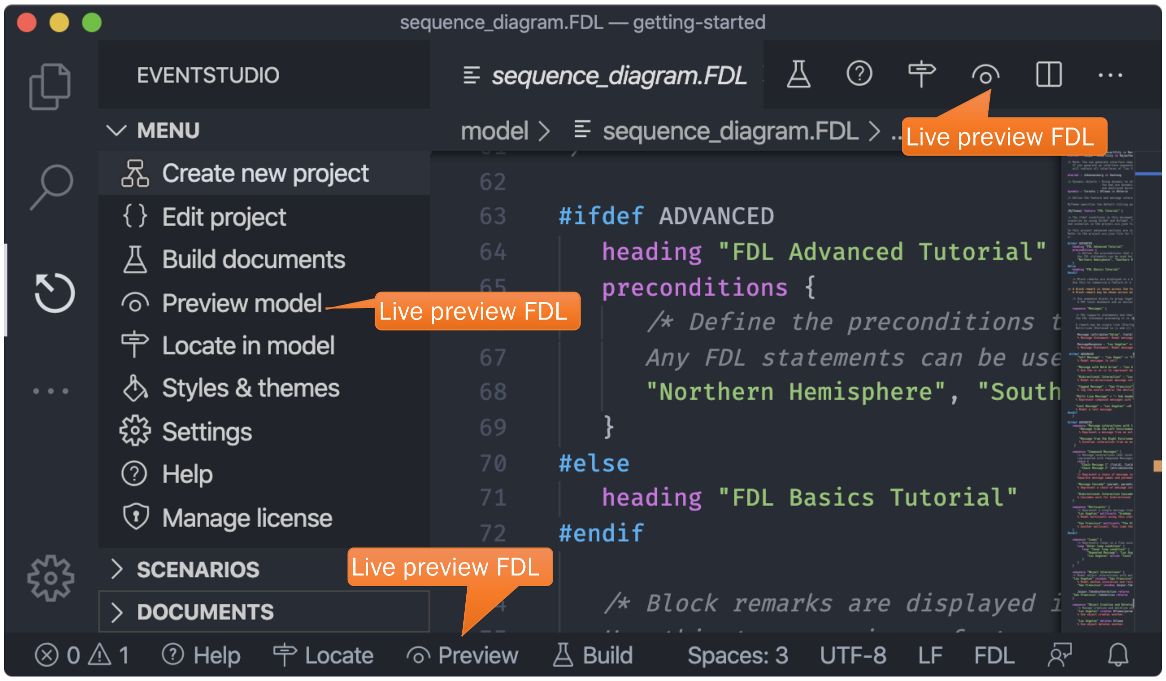
Message Statement: Model messages interactions with parameters.

Message Statement: Model messages interactions (no parameters specified in this case).

Model method invocation and return.

1.3 Learn Feature Description Language (FDL)

1. Quickest way to learn FDL is to match the content of the `sequence_diagram.FDL` file with the live preview.
    a. Make sure that `sequence_diagram.FDL` window is active in Visual Studio Code.
    b. Click the `Preview` button in the status bar (keyboard shortcut: `Alt+P`).

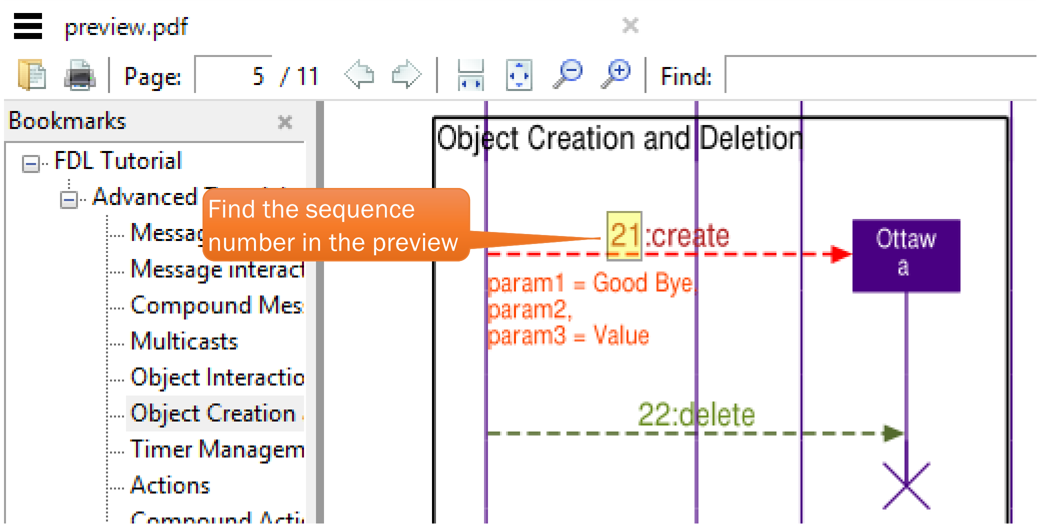


```
#ifdef ADVANCED
   heading "FDL Advanced Tutorial"
   preconditions {
      /* Define the preconditions t
      Any FDL statements can be use
      "Northern Hemisphere", "South
   }
#else
   heading "FDL Basics Tutorial"
#endif

   /* Block remarks are displayed i
```

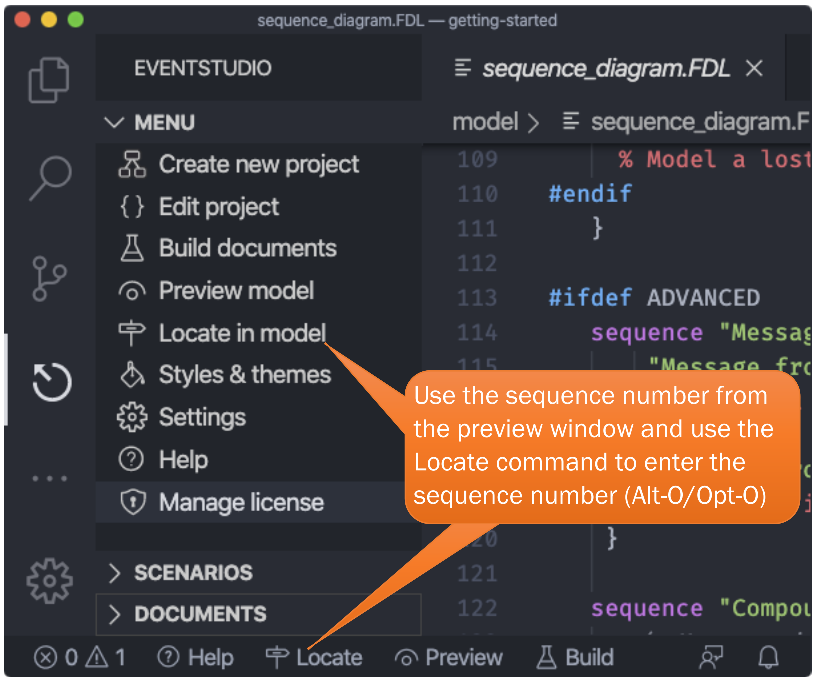
c. EventStudio opens a live PDF preview for the file. When running on Windows, the preview is synchronized to the cursor location in the FDL file[3].

d. We recommend enabling auto save (`File → Auto Save`) so that FDL changes show up in the preview as you type.

2. Use the sequence number in the preview to navigate back to the corresponding FDL file location with Alt+O (Option+O on a Mac).



[3] Cursor sync works on Windows only.



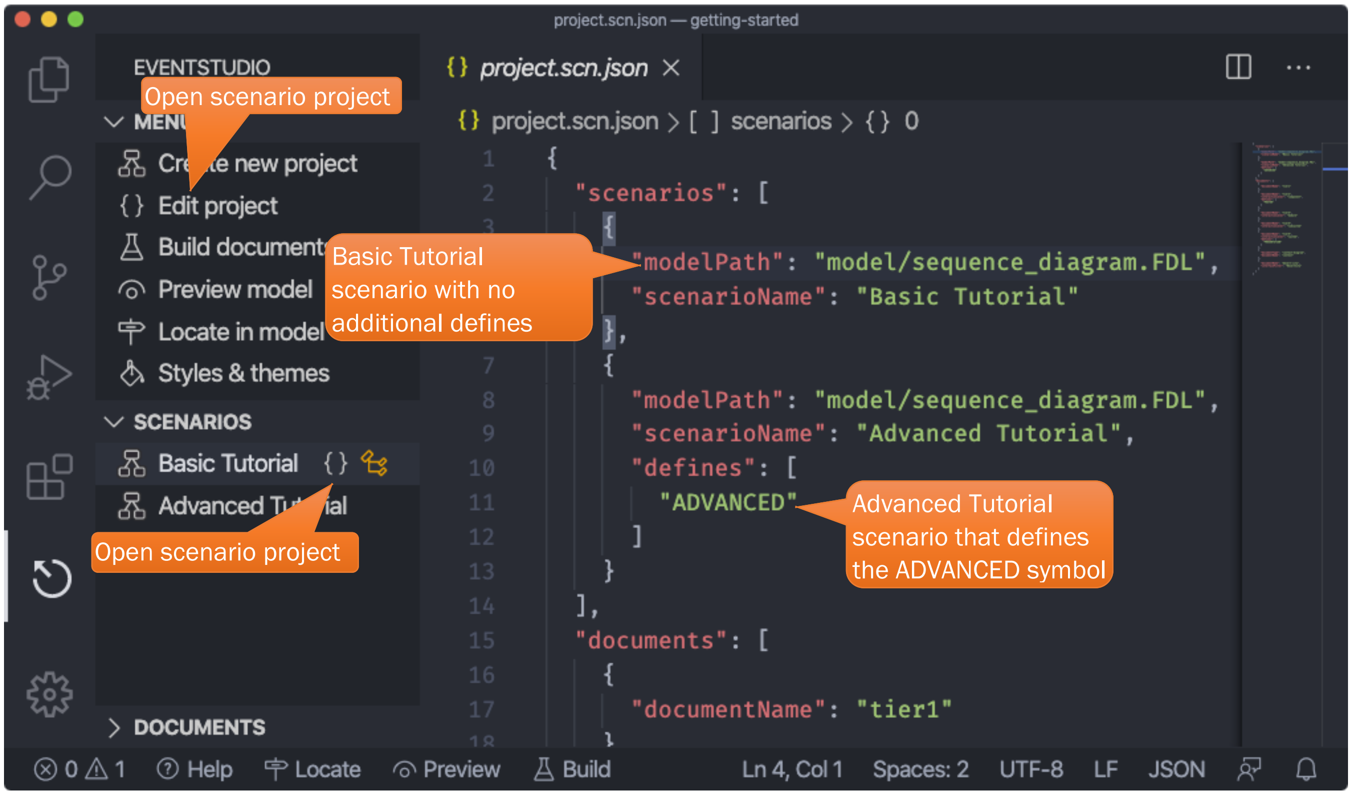
```
        % Model a lost
#endif
    }

#ifdef ADVANCED
    sequence "Messag
        "Message fro
    }

    sequence "Compou
```

Use the sequence number from the preview window and use the Locate command to enter the sequence number (Alt-O/Opt-O)

3. You can also use snippets to quickly add FDL statements (Refer to Insert FDL Snippets).

1.4 Learn About Multiple Scenarios

1. Click on the “Edit project” menu to open the project.scn.json file. This file defines scenarios and documents.



```
{
  "scenarios": [
    {
      "modelPath": "model/sequence_diagram.FDL",
      "scenarioName": "Basic Tutorial"
    },
    {
      "modelPath": "model/sequence_diagram.FDL",
      "scenarioName": "Advanced Tutorial",
      "defines": [
        "ADVANCED"
      ]
    }
  ],
  "documents": [
    {
      "documentName": "tier1"
    }
```

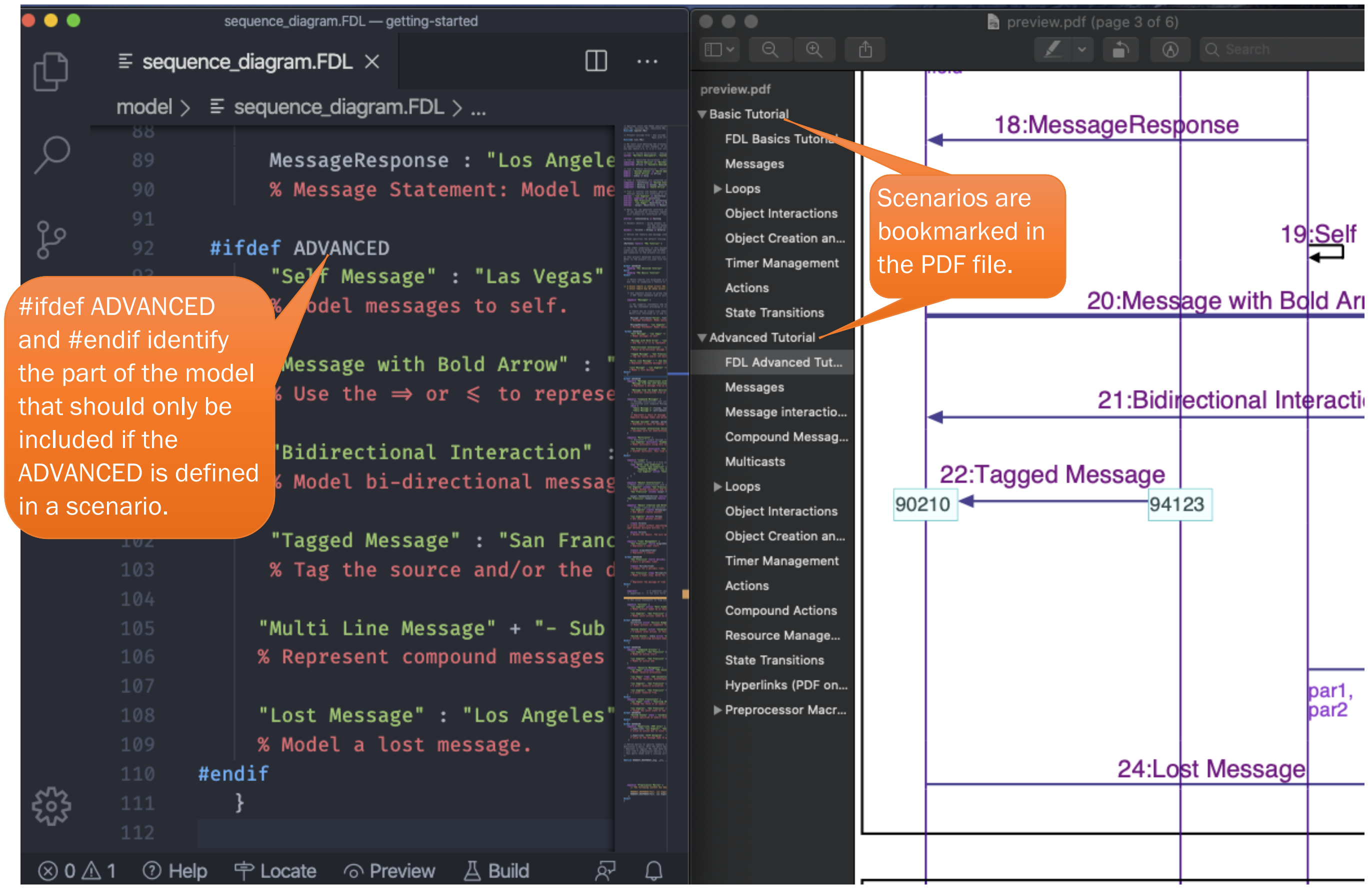
Basic Tutorial scenario with no additional defines

Advanced Tutorial scenario that defines the ADVANCED symbol

What are scenarios?

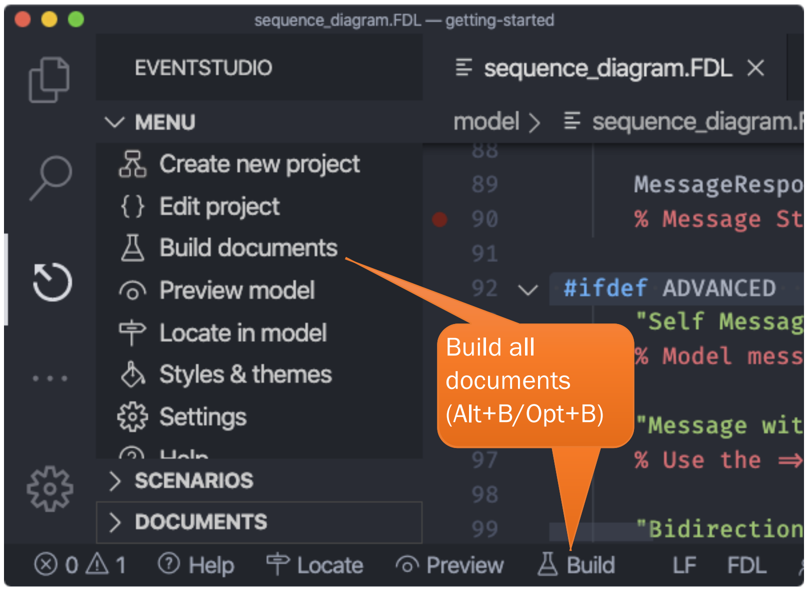
Scenarios identify the success and failure legs of a message flow. Scenarios are identified by specifying a list of preprocessors defines. The FDL model uses #ifdef to mark out the differences between flows.

2. In the FDL file you will see regions marked in the `#ifdef ADVANCED … #endif` blocks. These declarations are used generate `"FDL Basics Tutorial"` and `"FDL Advanced Tutorial"` using two scenarios defined in the `project.scn.json` file.
    a. The default project file defines two scenarios:
        i. Basic Tutorial without the any defines. This scenario excludes the content from the ADVANCED region.
        ii. Advanced Tutorial with the ADVANCED define. This scenario includes the ADVANCED region as well.
    b. Note that the two scenarios are included in the generated preview PDF file with bookmarks for each scenario.



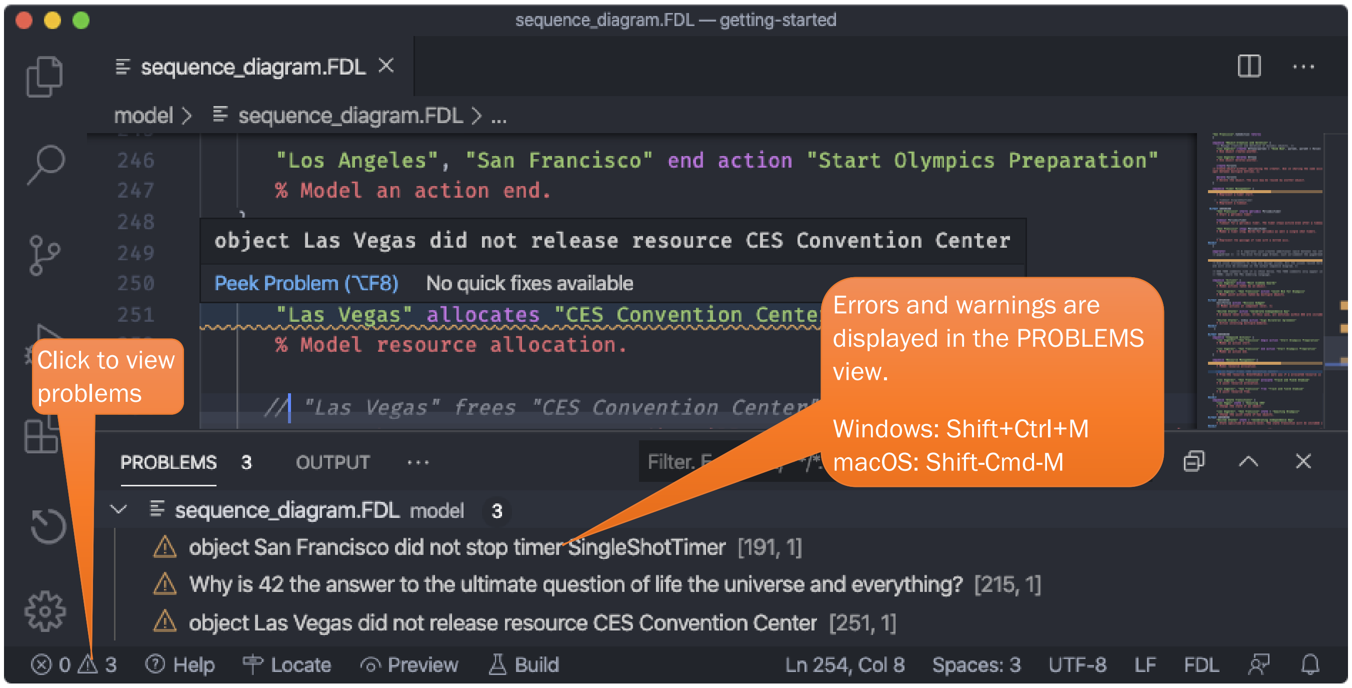
```
    MessageResponse : "Los Angele
    % Message Statement: Model me

#ifdef ADVANCED
    "Self Message" : "Las Vegas"
    % Model messages to self.

    "Message with Bold Arrow" : "
    % Use the ⇒ or ⇐ to represe

    "Bidirectional Interaction" :
    % Model bi-directional messag

    "Tagged Message" : "San Franc
    % Tag the source and/or the d

   "Multi Line Message" + "- Sub
   % Represent compound messages

   "Lost Message" : "Los Angeles"
   % Model a lost message.
#endif
  }
```

3. When you complete the design, you can generate the documents the “Build documents” command.



```
MessageRespo
% Message St

#ifdef ADVANCED
"Self Messag
% Model mess

"Message wit
% Use the ⇒

"Bidirection
```

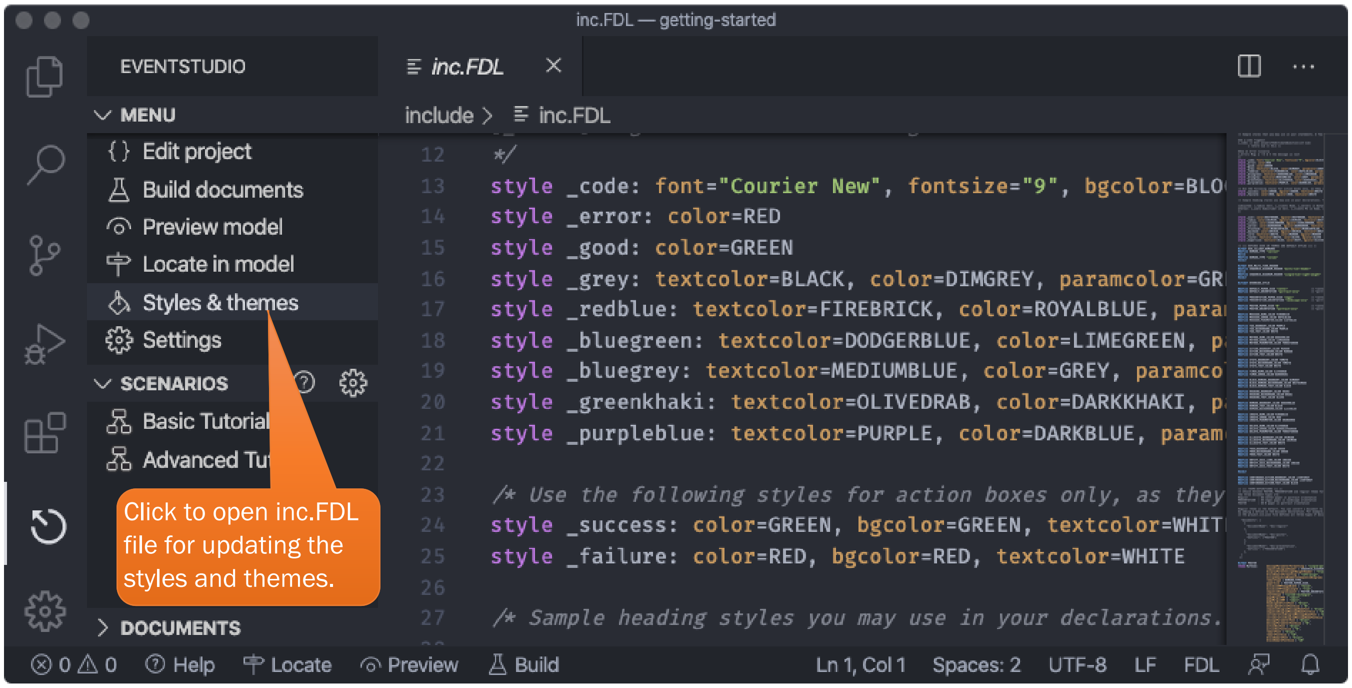
Errors and warnings are displayed in the PROBLEMS window.


sequence_diagram.FDL — getting-started
sequence_diagram.FDL
model > sequence_diagram.FDL > ...
Peek Problem (⌥F8) No quick fixes available
Click to view problems
Errors and warnings are displayed in the PROBLEMS view.
Windows: Shift+Ctrl+M
macOS: Shift-Cmd-M
PROBLEMS 3 OUTPUT
0 3 Help Locate Preview Build Ln 254, Col 8 Spaces: 3 UTF-8 LF FDL

```
   "Los Angeles", "San Francisco" end action "Start Olympics Preparation"
   % Model an action end.
object Las Vegas did not release resource CES Convention Center
   "Las Vegas" allocates "CES Convention Cente
   % Model resource allocation.

// "Las Vegas" frees "CES Convention Center
```

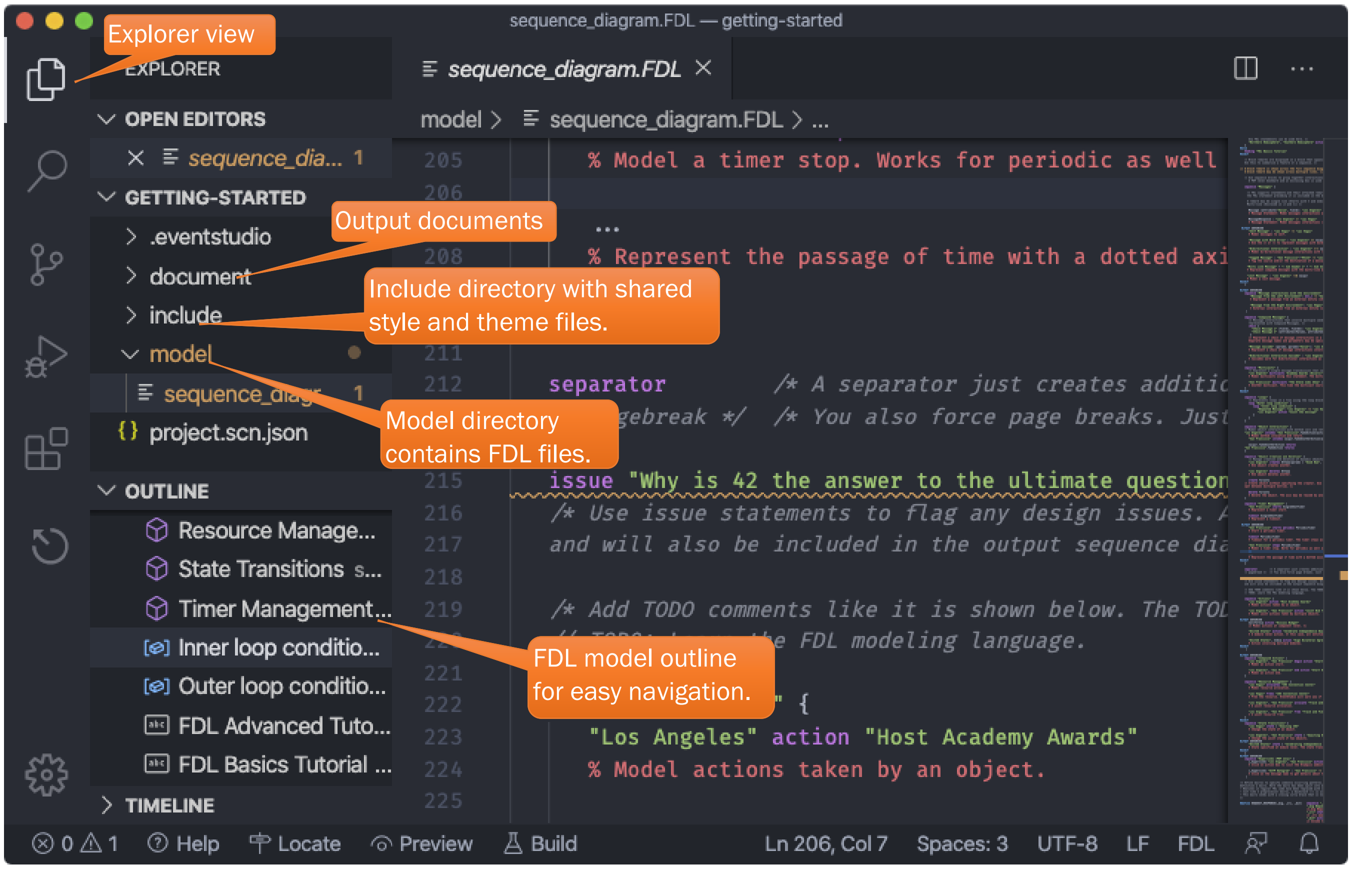
```
sequence_diagram.FDL model 3
  object San Francisco did not stop timer SingleShotTimer [191, 1]
  Why is 42 the answer to the ultimate question of life the universe and everything? [215, 1]
  object Las Vegas did not release resource CES Convention Center [251, 1]
```

1.5 Editing Tips



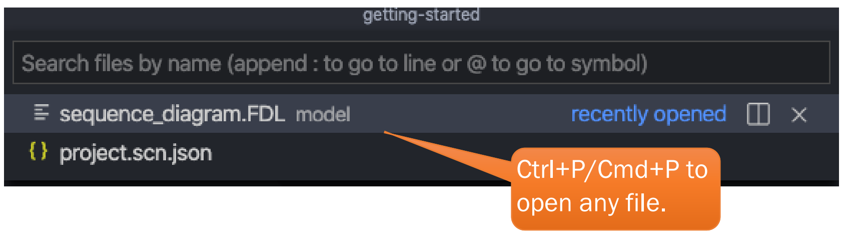
```
*/
style _code: font="Courier New", fontsize="9", bgcolor=BLO
style _error: color=RED
style _good: color=GREEN
style _grey: textcolor=BLACK, color=DIMGREY, paramcolor=GR
style _redblue: textcolor=FIREBRICK, color=ROYALBLUE, para
style _bluegreen: textcolor=DODGERBLUE, color=LIMEGREEN, p
style _bluegrey: textcolor=MEDIUMBLUE, color=GREY, paramco
style _greenkhaki: textcolor=OLIVEDRAB, color=DARKKHAKI, p
style _purpleblue: textcolor=PURPLE, color=DARKBLUE, param

/* Use the following styles for action boxes only, as they
style _success: color=GREEN, bgcolor=GREEN, textcolor=WHIT
style _failure: color=RED, bgcolor=RED, textcolor=WHITE

/* Sample heading styles you may use in your declarations.
```

We have explored the EventStudio view, now let's look at the Explorer view. This view shows the project directory structure and the outline of the active FDL model file.



```
      % Model a timer stop. Works for periodic as well

      ...
      % Represent the passage of time with a dotted axi


   separator          /* A separator just creates additic
        gebreak */    /* You also force page breaks. Just

   issue "Why is 42 the answer to the ultimate question
   /* Use issue statements to flag any design issues. A
   and will also be included in the output sequence dia

   /* Add TODO comments like it is shown below. The TOD
                       e FDL modeling language.

                                    " {
      "Los Angeles" action "Host Academy Awards"
      % Model actions taken by an object.
```

A few handy commands during editing are:

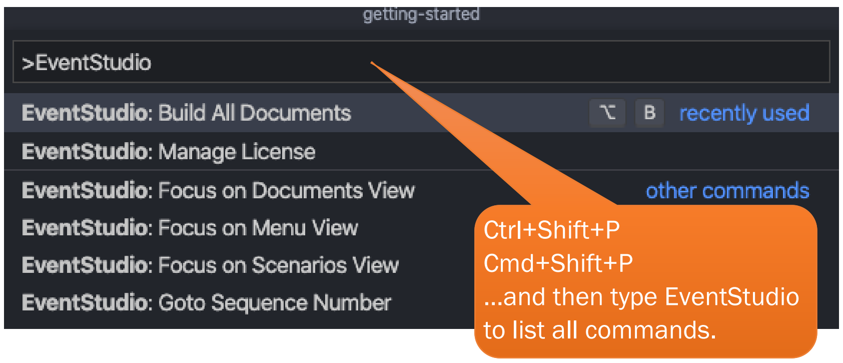

getting-started

Search files by name (append : to go to line or @ to go to symbol)

sequence_diagram.FDL model — recently opened

project.scn.json

Ctrl+P/Cmd+P to open any file.


getting-started

>EventStudio

EventStudio: Build All Documents — ⌥ B recently used

EventStudio: Manage License

EventStudio: Focus on Documents View — other commands

EventStudio: Focus on Menu View

EventStudio: Focus on Scenarios View

EventStudio: Goto Sequence Number

Ctrl+Shift+P
Cmd+Shift+P
...and then type EventStudio to list all commands.

1.6 Community Edition vs Professional Edition

EventStudio gets installed as the Community Edition. You can request a free 45 day free trial to the Professional Edition.

Feature	Supported in the Community Edition?	Supported in the Professional Edition?
Full FDL modeling support • Visual Guide to FDL • FDL Reference	Yes	Yes
Styles, Themes and Document Layout	Yes	Yes
Define multiple Scenarios	Yes	Yes
View the design through multiple Documents	Yes	Yes
Preprocessor	Yes	Yes
Entities Modeled	Limited to 10	50+
Document Page Limit	Limited to 10	500+
Export to Microsoft Word	Not supported	Supported in the Windows version
Paper Size	Letter and A4	Letter, Legal, B, C, A4, A3, B4 and custom size
Document Footer	Not customizable	Customize from Settings

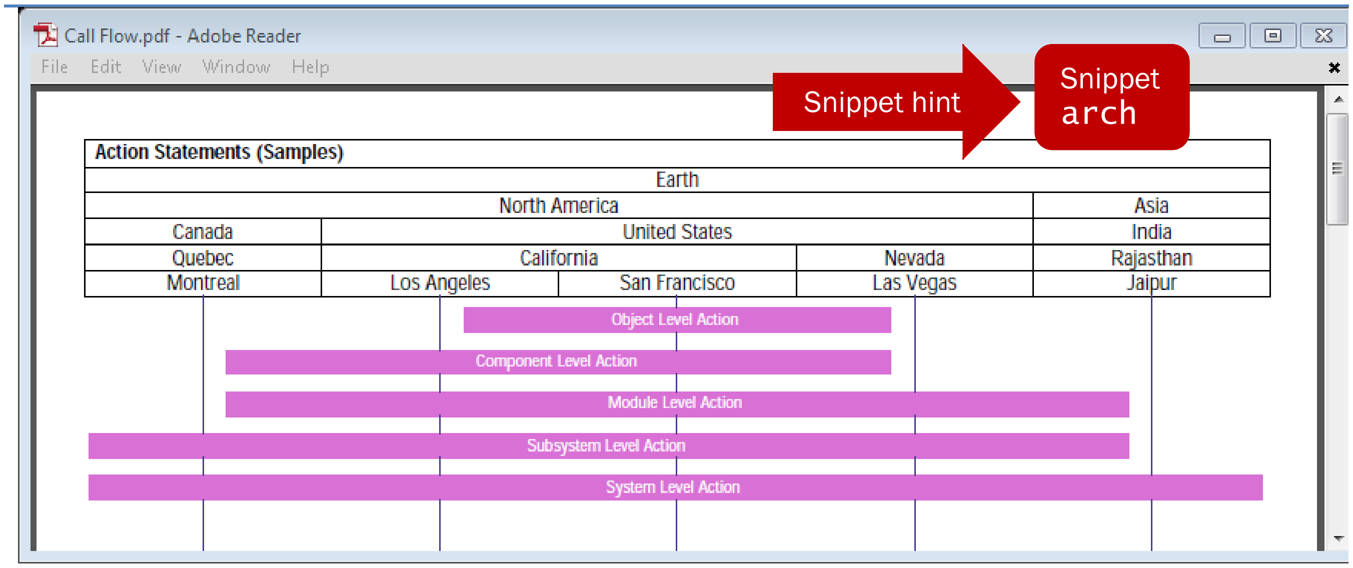
1.7 Explore More

- Visual Guide to FDL
- Scenarios and Documents
- Add New Scenarios and Documents
- Insert FDL Snippets
- Visual Studio Code Tips and Tricks

2 VISUAL GUIDE TO FDL

This section provides a visual introduction to FDL. Sequence diagrams and the associated FDL are shown. The snippet hints are shown on the right side of each diagram (Refer to section 4.5).

2.1 Architecture Definition

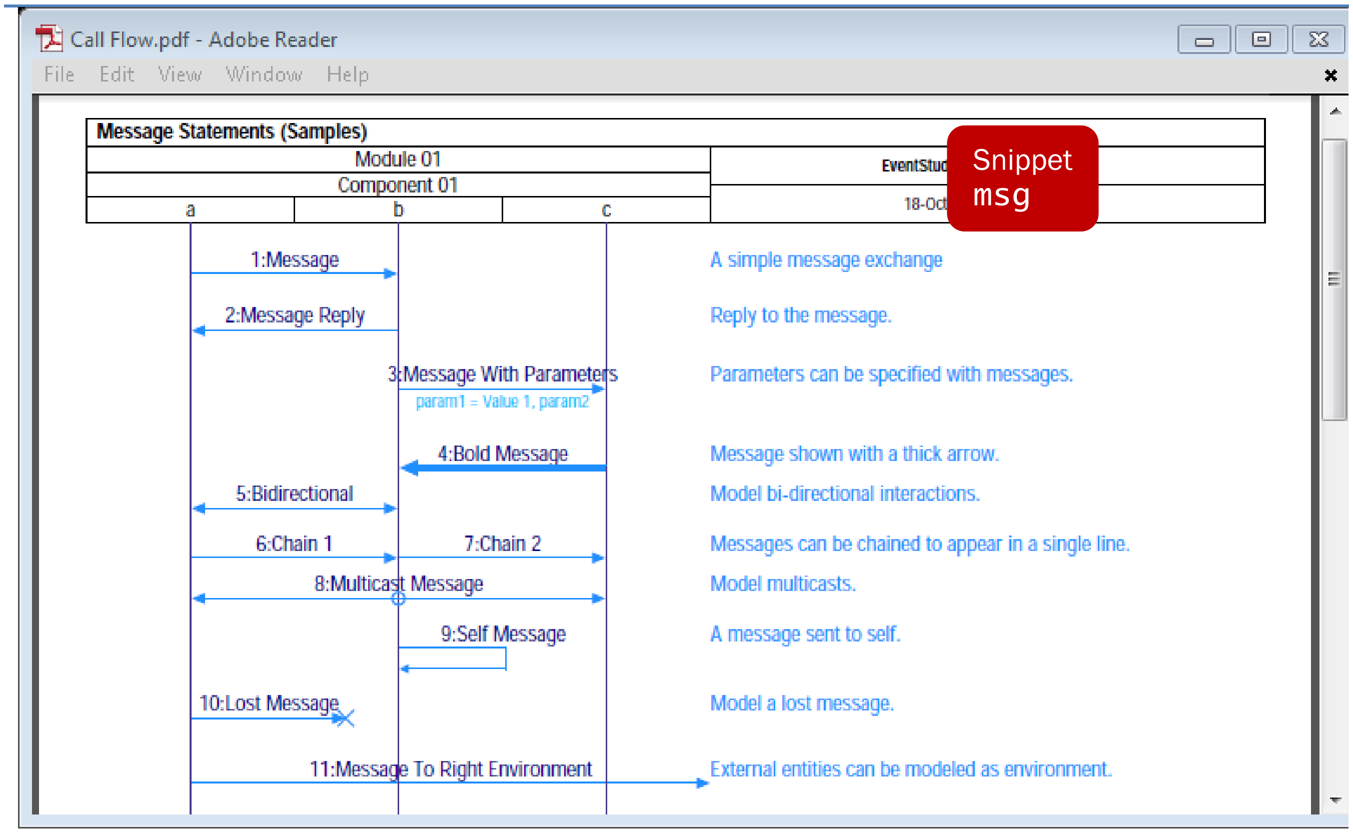


```
#include "inc.fdl"
system: Earth
subsystem:"North America" in Earth, Asia in Earth
module: Canada in "North America", "United States" in "North America"
module: India in Asia
component: California in "United States", Nevada in "United States"
component: Quebec in Canada, Rajasthan in India
eternal: "Los Angeles" in California, "San Francisco" in California
eternal: "Las Vegas" in Nevada, Montreal in Quebec, Jaipur in Rajasthan
feature "Action Statements" {
    "San Francisco" action "Object Level Action"
    California action "Component Level Action"
    "United States" action "Module Level Action"
    "North America" action "Subsystem Level Action"
    Earth action "System Level Action"
}
```

Explore More

- System, Subsystem, Module and Component declaration
- Eternal Object declaration
- Dynamic Object declaration

2.2 Messages



A simple message exchange

Reply to the message.

Parameters can be specified with messages.

Message shown with a thick arrow.

Model bi-directional interactions.

Messages can be chained to appear in a single line.

Model multicasts.

A message sent to self.

Model a lost message.

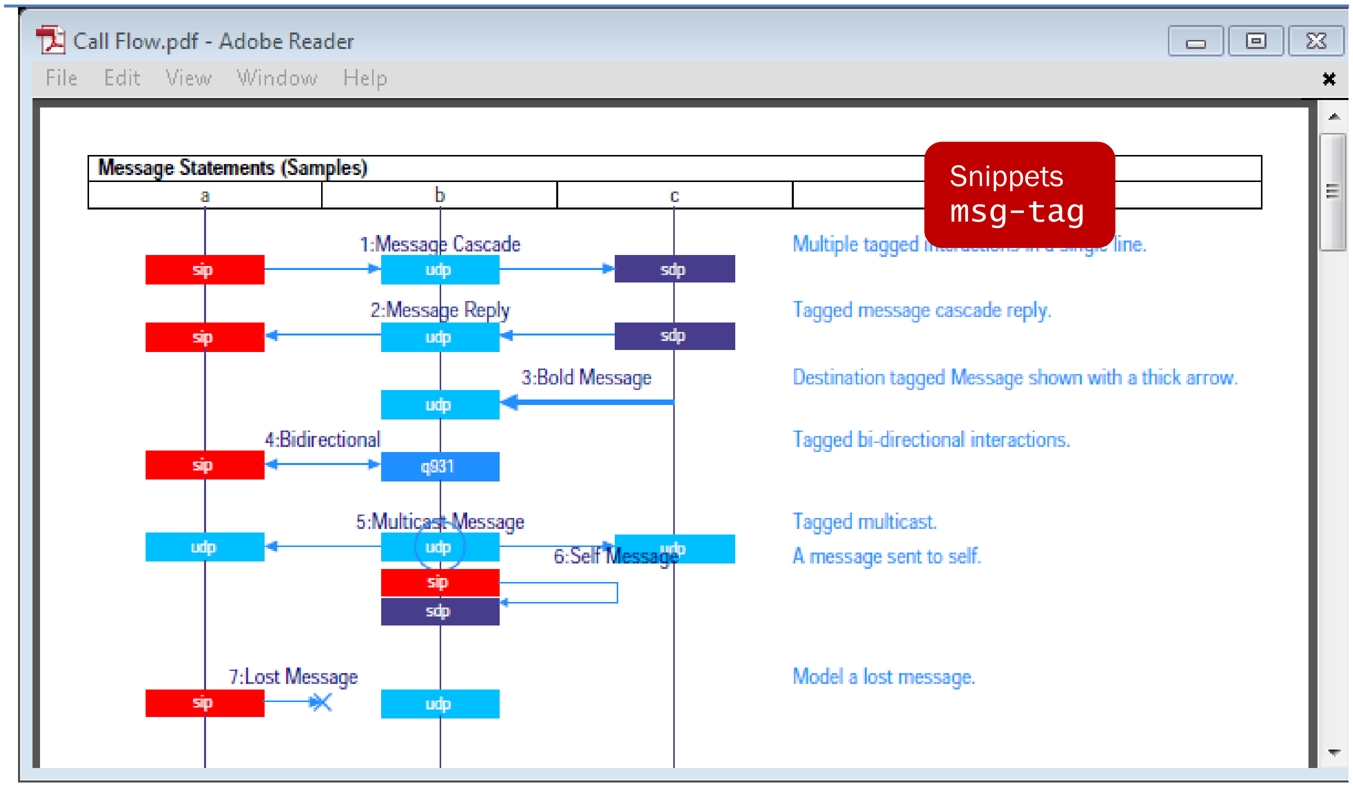
External entities can be modeled as environment.

```
module: Module_01
component: Component_01 in Module_01
eternal: a in Component_01, b in Component_01, c in Component_01
feature "Message Statements" {
   Message: a -> b            % A simple message exchange
   "Message Reply": a <- b     % Reply to the message. *|
   "Message With Parameters" (param1="Value 1", param2): b -> c
    % Parameters can be specified with messages.
   "Bold Message": b <= c     % Message shown with a thick arrow.
   "Bidirectional": a <-> b   % Model bi-directional interactions.
   chain {
      "Chain 1": a -> b
      "Chain 2": b -> c
    } % Messages can be chained to appear in a single line.
   b multicasts "Multicast Message" to a, c      % Model multicasts.
   "Self Message": b -> b   % A message sent to self.
   "Lost Message": a ->X b  % Model a lost message.
   "Message to Right Environment": a -> env_r
   % External entities can be modeled as environment.
}
```

Explore More

- Message statement
- Chain statement
- Multicast statement

2.3 Message Tagging

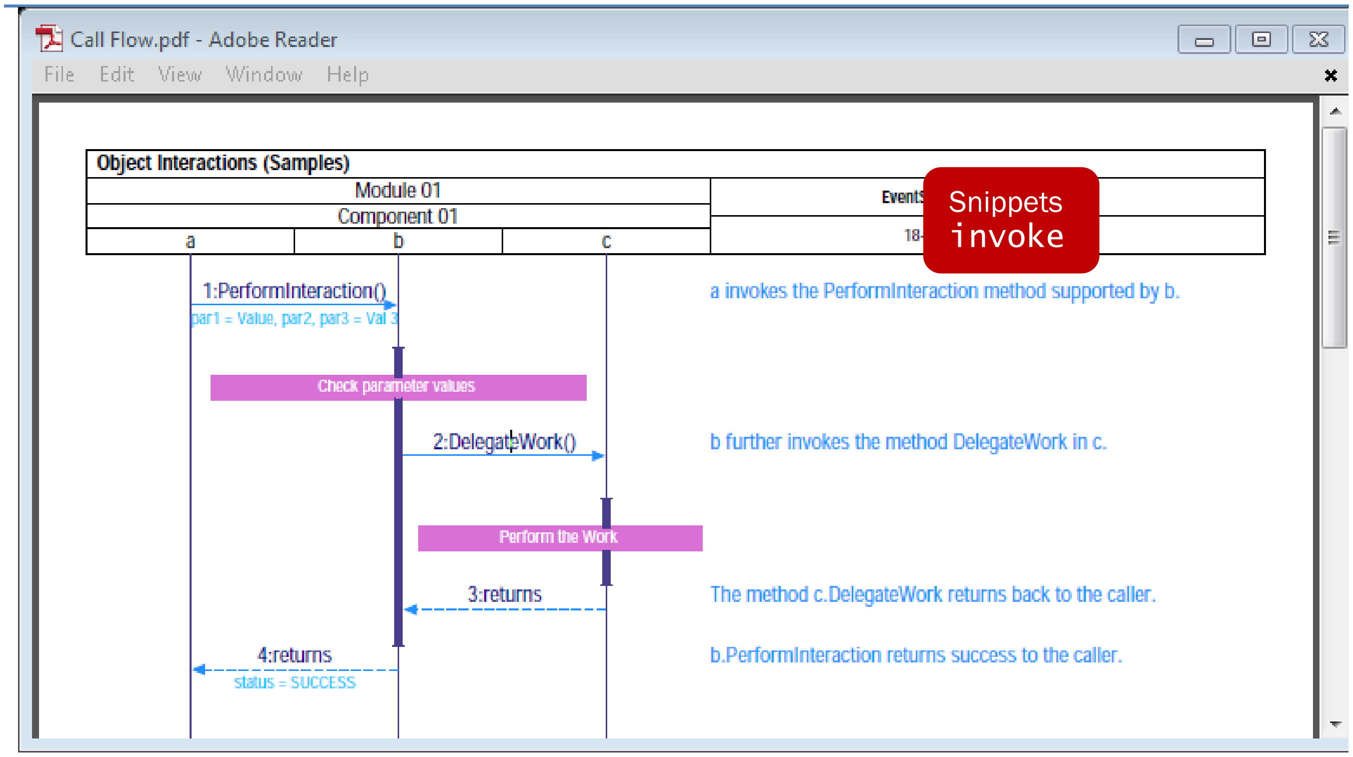

Call Flow.pdf - Adobe Reader
File Edit View Window Help
Snippets
msg-tag
Message Statements (Samples)
a
b
c
1:Message Cascade
2:Message Reply
3:Bold Message
4:Bidirectional
5:Multicast Message
6:Self Message
7:Lost Message
sip
udp
sdp
q931
Multiple tagged interactions in a single line.
Tagged message cascade reply.
Destination tagged Message shown with a thick arrow.
Tagged bi-directional interactions.
Tagged multicast.
A message sent to self.
Model a lost message.

```
eternal: a, b, c
feature "Message Statements" {
    "Message Cascade": a-sip -> b-udp -> c-sdp
    % Multiple tagged interactions in a single line.
    "Message Reply": a-sip <- b-udp <- c-sdp
    % Tagged message cascade reply.
    "Bold Message": b-udp <= c
    % Destination tagged Message shown with a thick arrow.
    "Bidirectional": a-sip <-> b-q931 % Tagged bi-directional interactions.
    b-udp multicasts "Multicast Message" to a-udp, c-udp
    % Tagged multicast.
    "Self Message": b-sip -> b-sdp     % A message sent to self.
    "Lost Message": a-sip ->X b-udp    % Model a lost message.
}
```

Explore More

- Message statement
- Chain statement
- Multicast statement

2.4 Object Interactions



a invokes the PerformInteraction method supported by b.

b further invokes the method DelegateWork in c.

The method c.DelegateWork returns back to the caller.

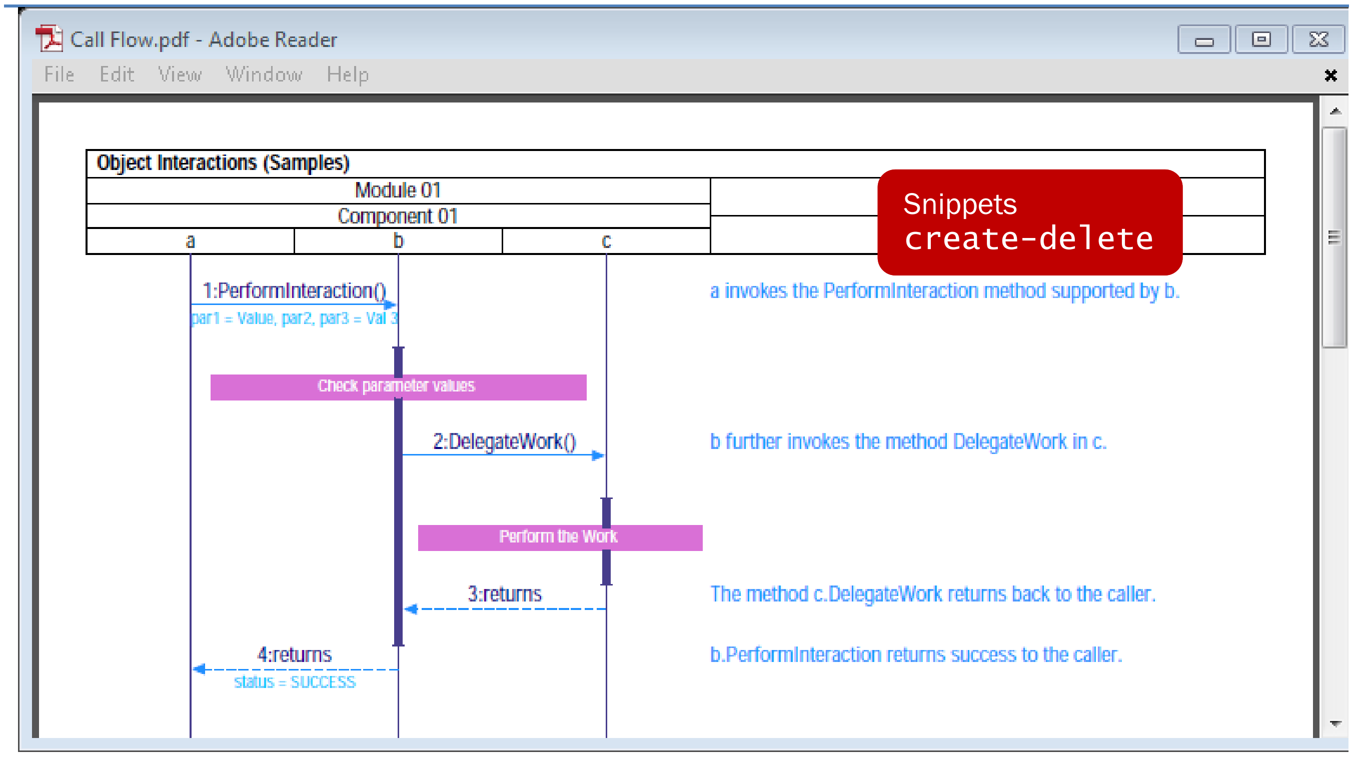
b.PerformInteraction returns success to the caller.

```
module: Module_01
component: Component_01 in Module_01
eternal: a in Component_01, b in Component_01, c in Component_01
feature "Object Interactions" {
    a invokes b.PerformInteraction(par1 = Value, par2, par3 = "Val 3")
    % a invokes the PerformInteraction method supported by b.
    b action "Check parameter values"
    b invokes c.DelegateWork
    % b further invokes the method DelegateWork in c.
    c action "Perform the Work"
    c.DelegateWork returns
    % The method c.DelegateWork returns back to the caller.
    b.PerformInteraction returns (status = SUCCESS)
    % b.PerformInteraction returns success to the caller.
}
```

Explore More

- Invokes and Returns statements

2.5 Object Creation and Deletion



a invokes the PerformInteraction method supported by b.

b further invokes the method DelegateWork in c.

The method c.DelegateWork returns back to the caller.

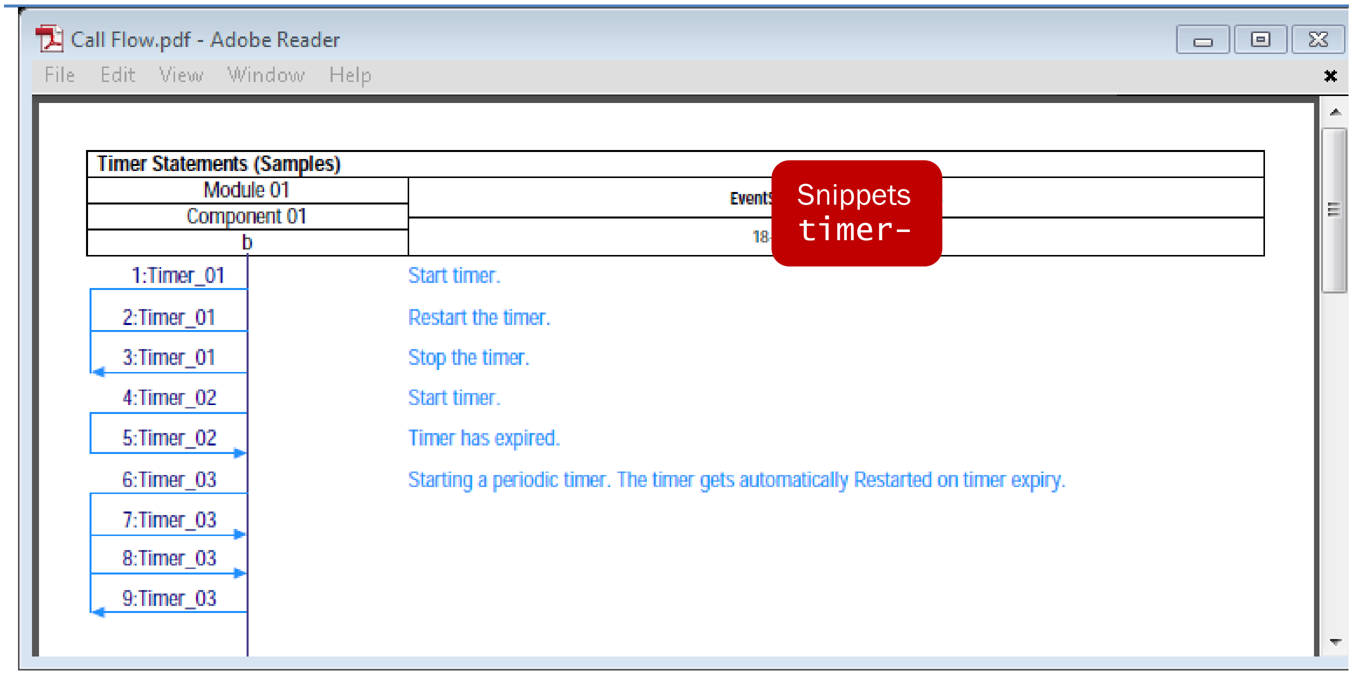
b.PerformInteraction returns success to the caller.

```
module: Module_01
component: Component_01 in Module_01
eternal: a in Component_01
dynamic: b in Component_01, c1 | c2 in Component_01
/* Dynamic object c1 and c2 share an axis. Only one of them
can be active at a given time */
feature "Object Creation and Deletion" {
    a creates b
    % Eternal object 'a' dynamically creates dynamic object 'b'.
    b creates c1(par1, par2="Value 2")
    % Dynamic object 'b' dynamically creates dynamic object 'c1'.
    b deletes c1
    % Dynamic object 'b' deletes dynamic object 'c1'.
    a deletes b
    % Eternal object 'a' deletes dynamic object 'b'.
    create c2
    % Anonymous create. Note that c2 reuses the axis used by c1.
    delete c2    |* Anonymous delete.
}
```

Explore More

- Create statement
- Delete statement

2.6 Timers



Timer Statements (Samples)

Module 01

Component 01

b

1:Timer_01 Start timer.

2:Timer_01 Restart the timer.

3:Timer_01 Stop the timer.

4:Timer_02 Start timer.

5:Timer_02 Timer has expired.

6:Timer_03 Starting a periodic timer. The timer gets automatically Restarted on timer expiry.

7:Timer_03

8:Timer_03

9:Timer_03

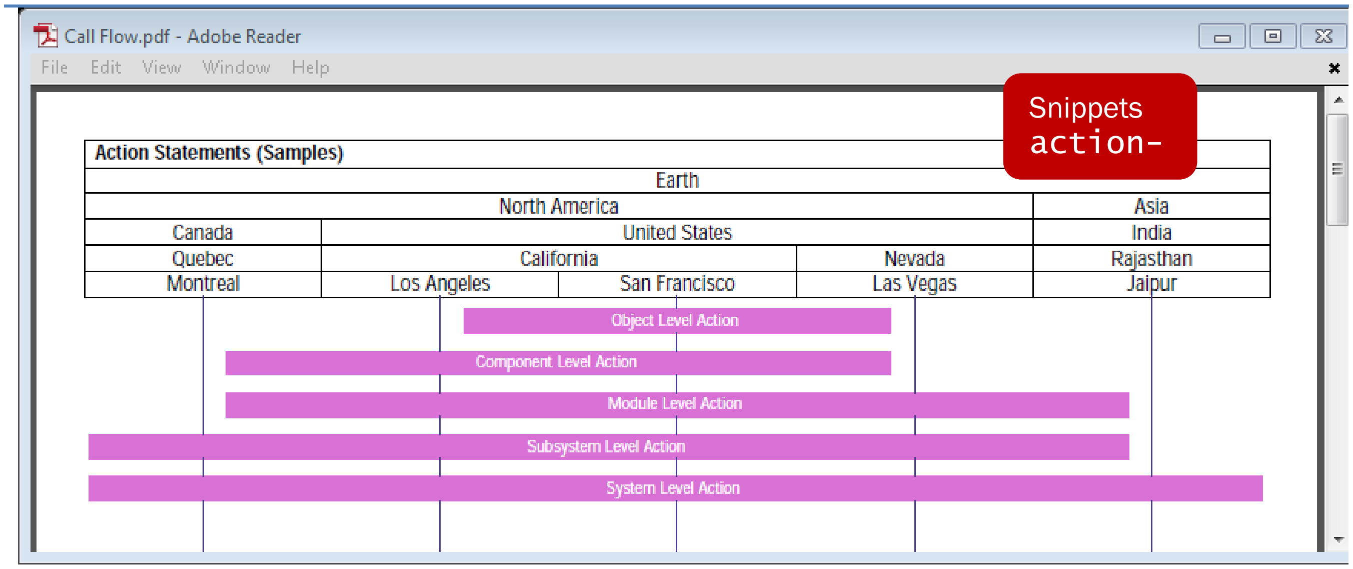
```
module: Module_01
component: Component_01 in Module_01
eternal: a in Component_01, b in Component_01, c in Component_01
feature "Timer Statements" {
    b starts Timer_01     % Start timer.
    b restarts Timer_01   % Restart the timer.
    b stops Timer_01      % Stop the timer.

    b starts Timer_02     % Start timer.
    timeout Timer_02      % Timer has expired.

    b starts periodic Timer_03
    |* Starting a periodic timer. The timer gets automatically
    Restarted on timer expiry. *|
    timeout Timer_03
    timeout Timer_03
    b stops Timer_03
}
```

Explore More

- Timer Start (One Shot and Periodic) statement
- Timer Restart statement
- Timer Stop statement
- Timeout statement

2.7 Actions

2.7.1 Single Action

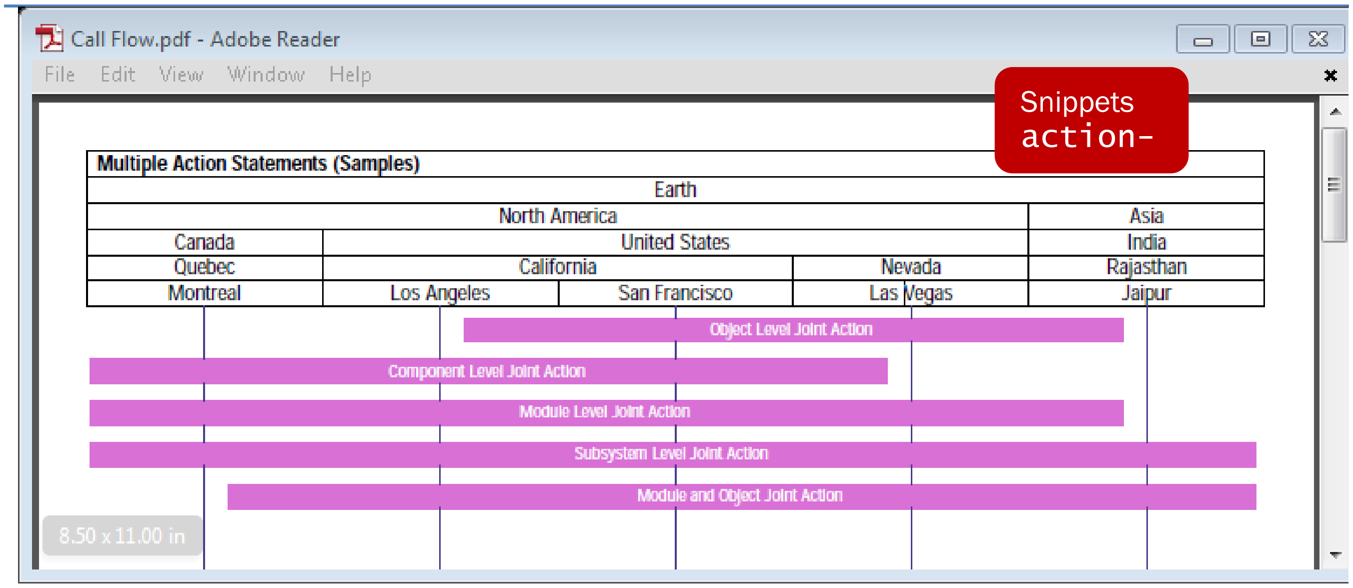


```
#include "inc.fdl"
system: Earth
subsystem:"North America" in Earth, Asia in Earth
module: Canada in "North America", "United States" in "North America"
module: India in Asia
component: California in "United States", Nevada in "United States"
component: Quebec in Canada, Rajasthan in India
eternal: "Los Angeles" in California, "San Francisco" in California
eternal: "Las Vegas" in Nevada, Montreal in Quebec, Jaipur in Rajasthan
feature "Action Statements" {
    "San Francisco" action "Object Level Action"
    California action "Component Level Action"
    "United States" action "Module Level Action"
    "North America" action "Subsystem Level Action"
    Earth action "System Level Action"
}
```

Explore More

- Action statement

2.7.2 Multiple Action

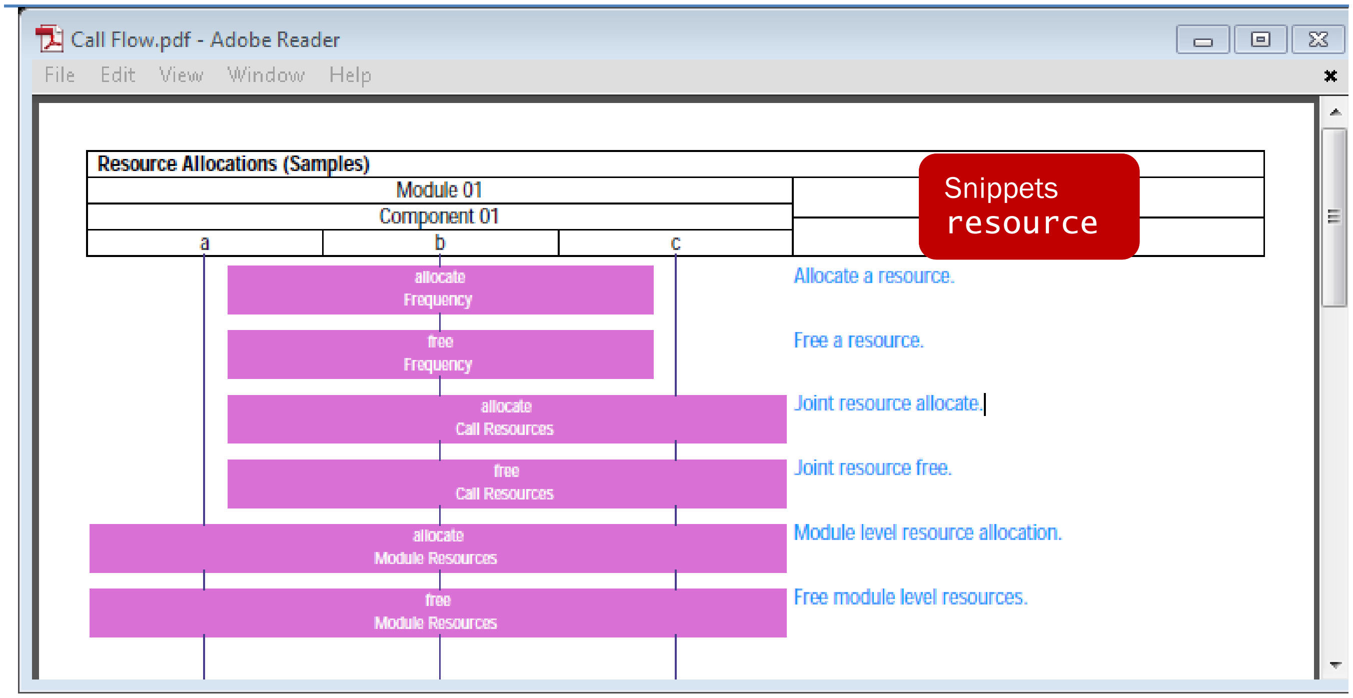

Call Flow.pdf - Adobe Reader
File Edit View Window Help
Snippets
action-
Multiple Action Statements (Samples)
Earth
North America
Asia
Canada
United States
India
Quebec
California
Nevada
Rajasthan
Montreal
Los Angeles
San Francisco
Las Vegas
Jaipur
Object Level Joint Action
Component Level Joint Action
Module Level Joint Action
Subsystem Level Joint Action
Module and Object Joint Action
8.50 x 11.00 in

```
#include "inc.fdl"
system: Earth
subsystem:"North America" in Earth, Asia in Earth
module: Canada in "North America", "United States" in "North America"
module: India in Asia
component: California in "United States", Nevada in "United States"
component: Quebec in Canada, Rajasthan in India
eternal: "Los Angeles" in California, "San Francisco" in California
eternal: "Las Vegas" in Nevada, Montreal in Quebec, Jaipur in Rajasthan
feature "Multiple Action Statements" {
   "San Francisco", "Las Vegas" action "Object Level Joint Action"
   California, Quebec action "Component Level Joint Action"
   "United States", Canada action "Module Level Joint Action"
   "North America", Asia action "Subsystem Level Joint Action"
   "United States", Jaipur action "Module and Object Joint Action"
}
```

Explore More

- Action statement
- Action Begin statement
- Action End statement

2.8 Resource Allocations

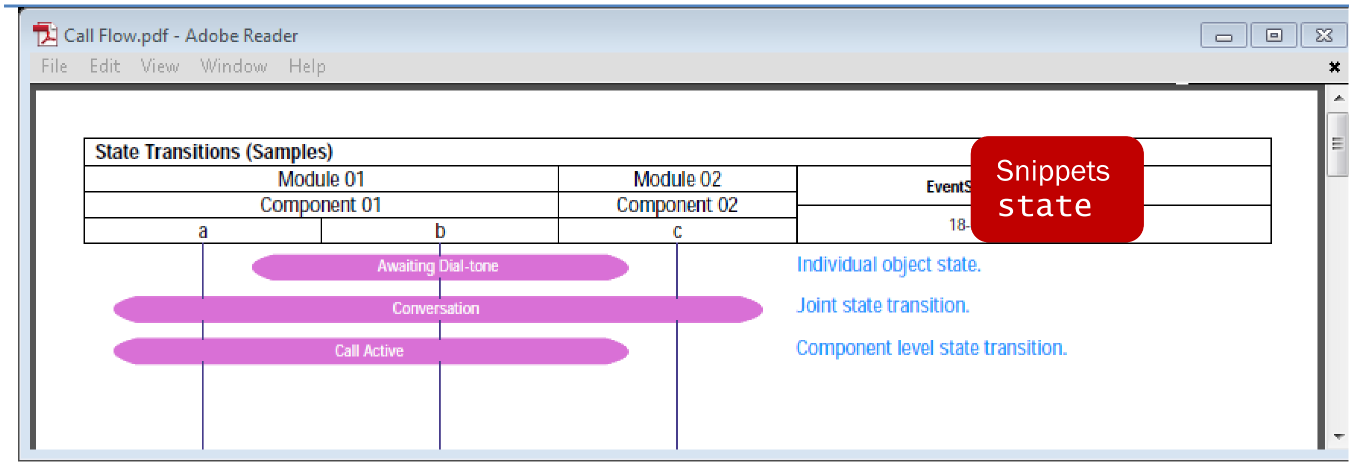


```
module: Module_01
component: Component_01 in Module_01
eternal: a in Component_01, b in Component_01, c in Component_01
feature "Resource Allocations" {
   b allocates "Frequency"    % Allocate a resource.
   b frees "Frequency"        % Free a resource.
   b, c allocate "Call Resources"    % Joint resource allocate.
   b, c free "Call Resources"        % Joint resource free.
   Module_01 allocates "Module Resources"
   % Module level resource allocation.
   Module_01 frees "Module Resources"
   % Free module level resources.
}
```

Explore More

- Resource Allocate statement
- Resource Free statement

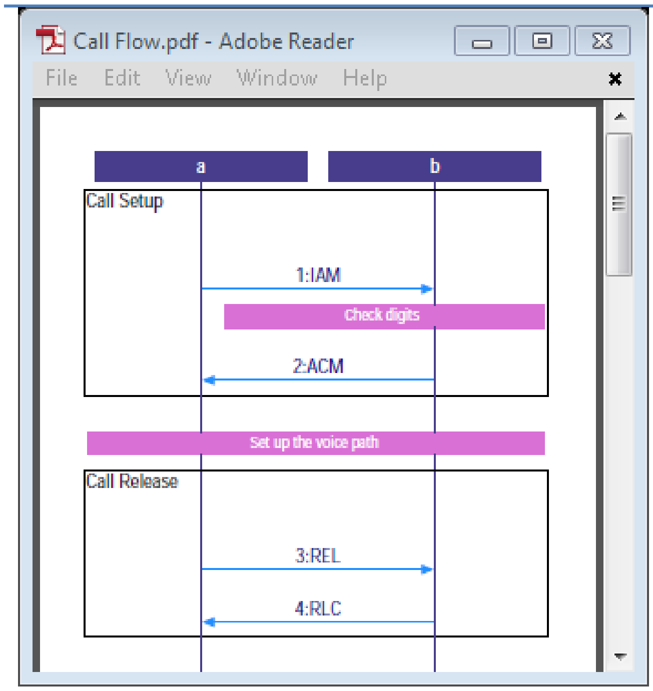
2.9 State Transitions

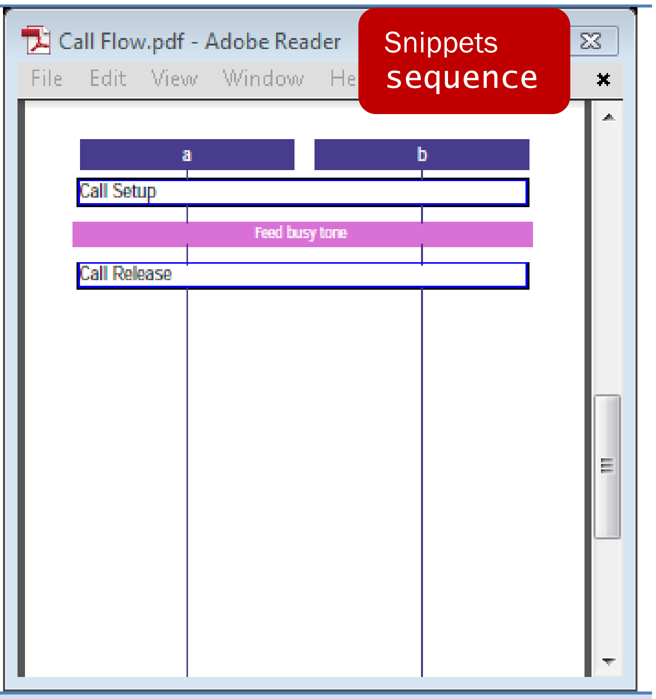


```
module: Module_01, Module_02
component: Component_01 in Module_01, Component_02 in Module_02
eternal: a in Component_01, b in Component_01, c in Component_02
feature "State Transitions" {
   b state = "Awaiting Dial-tone"        % Individual object state.
   a,b,c state = "Conversation"          % Joint state transition.
   Component_01 state = "Call Active"
   % Component level state transition.
}
```

Explore More
- State Change statement

2.10 Sequence Groupings



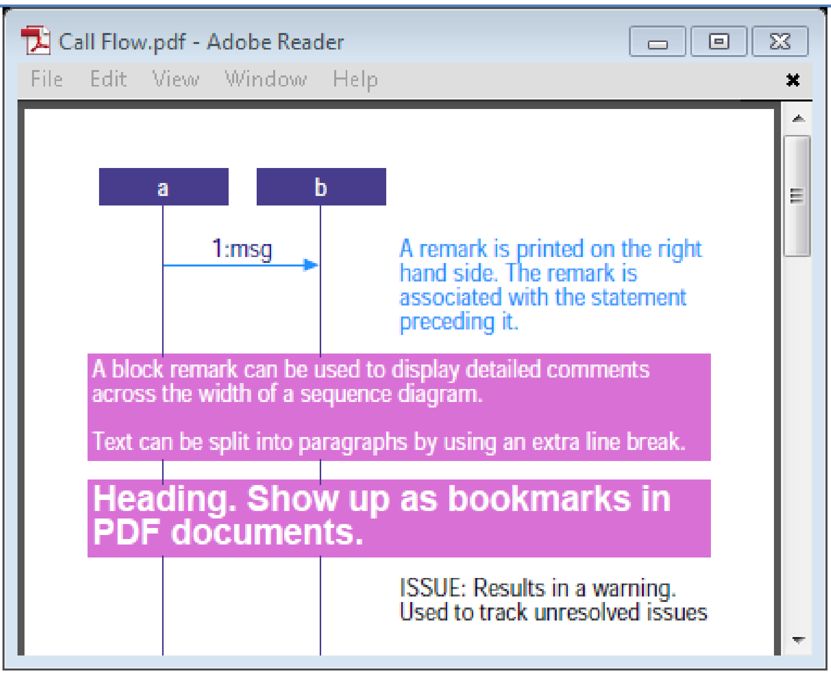


```
module: Module_01
component: Component_01 in Module_01
eternal: a in Component_01, b in Component_01, c in Component_01
feature "Sequence Grouping" {
  sequence "Call Setup" {
     IAM: a -> b
     b action "Check digits"
     ACM: a <- b
  }
  a, b action "Set up the voice path"
  sequence "Call Release" {
     REL: a -> b
     RLC: a <- b
  }
}
```

Explore More

- Sequence Block Statement

2.11 Remarks



A remark is printed on the right hand side. The remark is associated with the statement preceding it.

A block remark can be used to display detailed comments across the width of a sequence diagram.

Text can be split into paragraphs by using an extra line break.

Heading. Show up as bookmarks in PDF documents.

ISSUE: Results in a warning. Used to track unresolved issues

```
module: Module_01
component: Component_01 in Module_01
eternal: a in Component_01, b in Component_01, c in Component_01
feature "Remarks" {
   msg: a -> b |* A remark is printed on the right-hand side.
   The remark is associated with the statement preceding it. *|

   ack: a <- b  % Single line remark

   |= A block remark can be used to display detailed comments across
   the width of a sequence diagram.

   Text can be split into paragraphs by using an extra line break.=|

   /* C-style comment ignored in document generation. */

   heading "Heading. Show up as bookmarks in PDF documents."

   issue "Results in a warning. Used to track unresolved issues"
}
```

Explore More

- Remark statement
- Block Remark statement
- Comment statement
- Heading statement
- Issue statement

2.12 Styling

The inc.FDL file defines axis and interaction styles

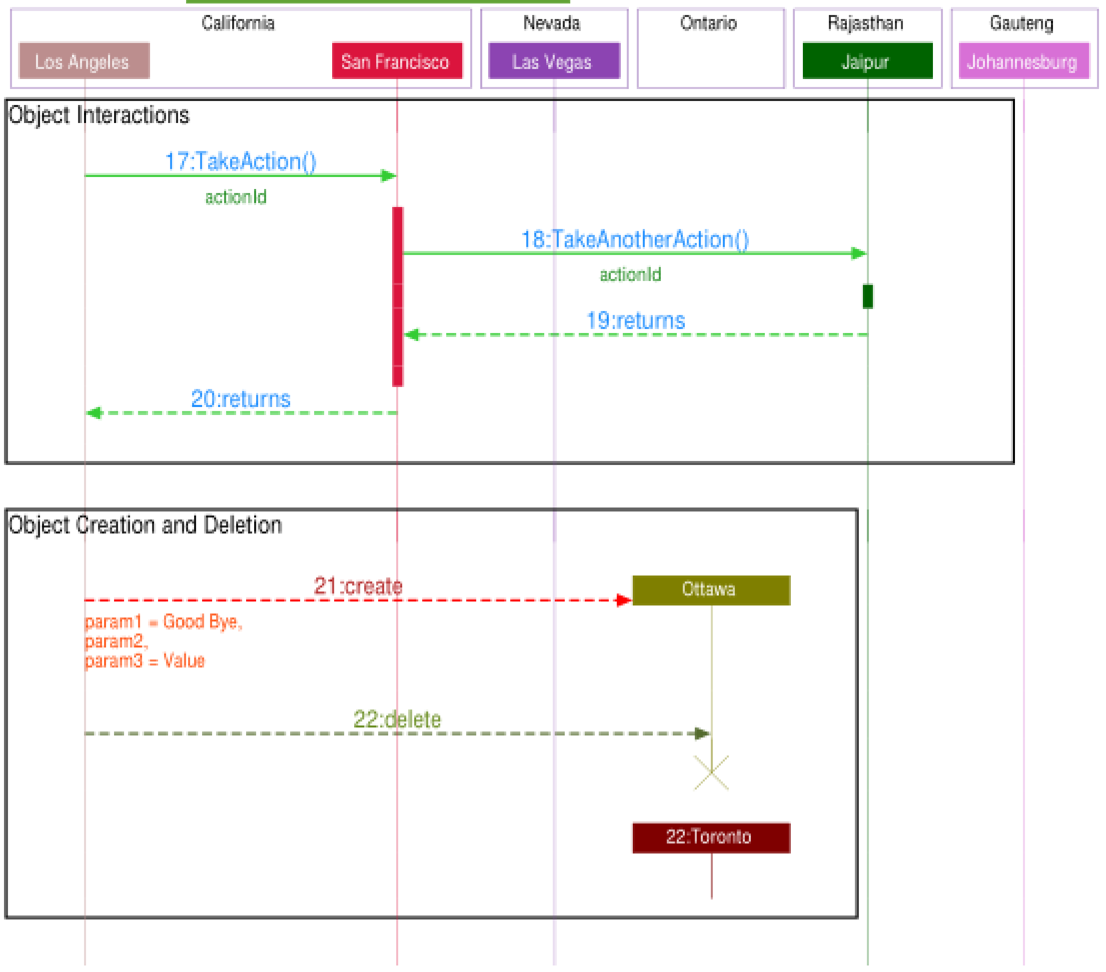
2.12.1 Axis Styles

Change color, font and other attributes assigned to the individual axis by specifying a style prefix to system to object level entities. Here is the axis styling that is predefined in the inc.FDL file.

Snippets
style

```
eternal : [_user] "Los Angeles" in California
eternal: separator in California
eternal: [_radio] "San Francisco" in California
eternal : [_client] "Las Vegas": DesertCity in Nevada
eternal : [_server] Jaipur: DesertCity in Rajasthan
eternal : [_backend] Johannesburg in Gauteng
dynamic : [_core] Toronto |[_router] Ottawa in Ontario
```

Axis style specification



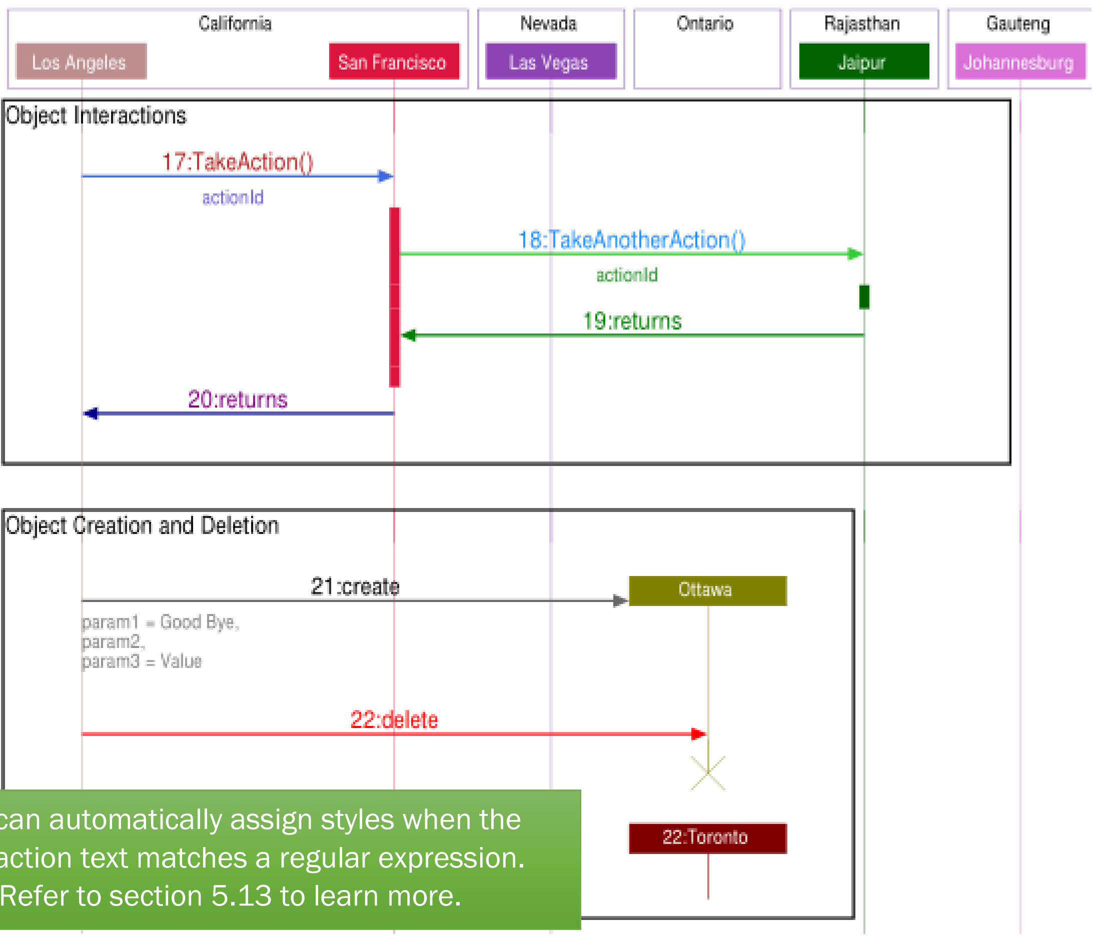
2.12.2 Interaction Styles

Change color, font and other attributes assigned to messages, actions, object interactions with inline style specification.

```
sequence "Object Interactions" {
[_redblue] "Los Angeles" invokes "San Francisco".TakeAction(actionId)
   % Model method invocation and return.
   [_bluegreen] "San Francisco" invokes Jaipur.TakeAnotherAction(actionId)
   [_good] Jaipur.TakeAnotherAction returns
[_purpleblue] "San Francisco".TakeAction returns
}
   sequence "Object Creation and Deletion" {
   [_grey]  "Los Angeles" creates Ottawa(param1 = "Good Bye", param2,
                                             param3 = Value)
   [_error] "Los Angeles" deletes Ottawa
   create Toronto
   delete Toronto
}
```

Snippets
style

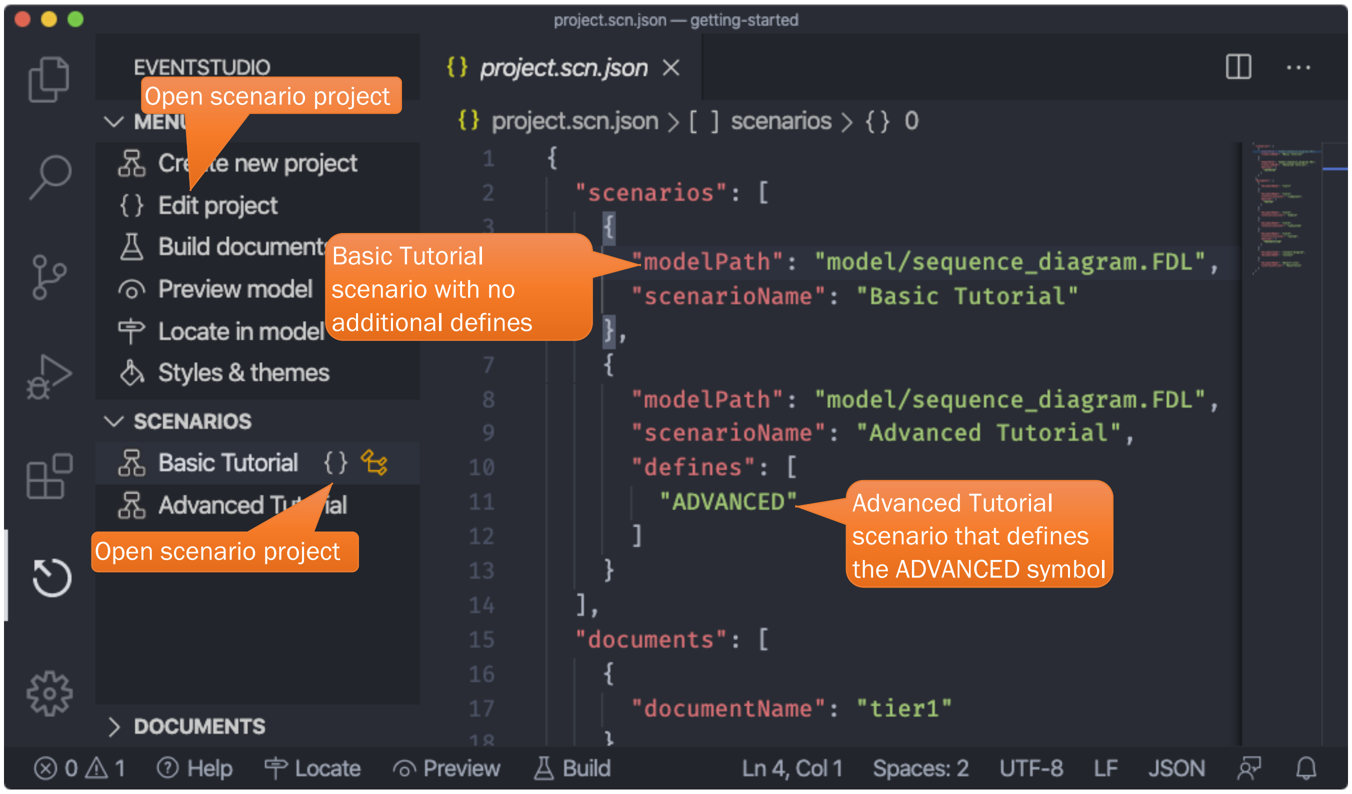
Interaction styles



You can automatically assign styles when the interaction text matches a regular expression. Refer to section 5.13 to learn more.

3 SCENARIOS AND DOCUMENTS

Scenarios and documents are defined by the `project.scn.json` file in the project directory. You can open the file by clicking on the EventStudio sidebar. The following figure also shows scenarios defined in the tutorial project.


project.scn.json — getting-started
EVENTSTUDIO
MENU
Create new project
Edit project
Build documents
Preview model
Locate in model
Styles & themes
SCENARIOS
Basic Tutorial
Advanced Tutorial
DOCUMENTS
project.scn.json
project.scn.json > [ ] scenarios > { } 0
Open scenario project
Basic Tutorial scenario with no additional defines
Advanced Tutorial scenario that defines the ADVANCED symbol
Open scenario project
0 1 Help Locate Preview Build
Ln 4, Col 1 Spaces: 2 UTF-8 LF JSON

```
{
  "scenarios": [
    {
      "modelPath": "model/sequence_diagram.FDL",
      "scenarioName": "Basic Tutorial"
    },
    {
      "modelPath": "model/sequence_diagram.FDL",
      "scenarioName": "Advanced Tutorial",
      "defines": [
        "ADVANCED"
      ]
    }
  ],
  "documents": [
    {
      "documentName": "tier1"
```

A sample `project.scn,json` file is shown below.

```
{
    "scenarios": [
      {
        "modelPath": "model/sequence_diagram.FDL",
        "scenarioName": "Basic Tutorial"

      },
      {
        "modelPath": "model/sequence_diagram.FDL",
        "scenarioName": "Advanced Tutorial",
        "defines": [ "ADVANCED"]
      }
    ],
    "documents": [
      {
        "documentName": "tier1"
      },
      {
        "documentName": "component-poster",
        "interactionLevel": "component",
        "defines": [ "POSTER"]

       },
      {
        "documentName": "tier3",
        "interactionLevel": "module"
      }
    ]
}
```

The file is divided into "scenarios" and a "documents" sections. These sections are described below:

3.1 Scenarios

The "scenarios" section is defined as an array of scenario objects. The scenario objects identify the models and their scenarios that should be included in the generated documents.

Each scenario object can contain the following fields:

"modelPath"	Mandatory	Specify a relative path from the project.scn.json file to the FDL file. Note that file name should use double slashes to comply with JSON formatting rules.
"scenarioName"	Mandatory	A title that summarizes the scenario. This title will be used in PDF bookmarks.
"defines"	Optional	A list of preprocessor defines specific to this scenario. Use #ifdef and #ifndefs in FDL files to conditionally identify the contents that should be included for this scenario.
"legs"	Optional	A list of scenario legs that are being called out for the scenario. If a scenario leg is not called out, EventStudio defaults to the first leg in an FDL case statement. Note that this primitive is defined for backward compatibility with EventStudio 6 with the now deprecated case-leg-endcase statement.

3.2 Documents

The "documents" section is defined as an array of document objects. The document objects specify the document type, filters and preprocessor defines for the document.

Each scenario object can contain the following fields:

"documentFormat"	Optional (defaults to "pdf" when omitted)	"Choose between pdf, emf (Microsoft Word Picture), xml or html formats. pdf and emf formats are available for sequence, context or collaboration diagrams. Summary documents and unit test

		procedures are generated in html. Use xml when exporting interactions to be used in other tools.
"documentName"	Mandatory	Defines the file name for the document without the file extension.
"documentType"	Optional (defaults to "sequence-diagram" when omitted)	Specify the document type. Choose between: • "sequence-diagram" • "context-diagram" • "collaboration-diagram" • "export-document" • "statistics-document" • "unit-tests" • "summary-document"
"internalMargin"	Optional (defaults to "large" when omitted)	Padding from the page boundaries to the objects. Choose between "small", "medium" and "large".
"objectSpacing"	Optional (defaults to "large" when omitted)	Spacing between objects in a context diagram. Choose between "small", "medium" and "large".
"expandSequence"	Optional (defaults to "expand-always" when omitted)	Specify how sequences should be expanded when they occur in multiple scenarios. Choose between "do-not-expand", "expand-first-time-only" or "expand-always".
"exclude"	Optional	List of items to be excluded from the generated sequence diagram. You can exclude: • "timers" • "resources" • "actions" • "remarks" • "blockRemarks" • "states" • "parameters"
"defines"	Optional	Preprocessor defines specific to this document. Use #ifdef and #ifndefs in FDL files to conditionally identify the contents that should be included in this document. POSTER and PRESENTATION defines may be used here to configure predefined settings for these document types.

"interfaceFilter"	Optional	Specify the entity that should be the focus of this diagram. System, subsystem, module, component or objects may be specified as entities. You may also specify types when more than one instance of a type is included in a document.
" textFilter"	Optional	Specify a string pattern that is contained in all the messages, methods and actions you wish to include in a document.
"regexTextFilter"	Optional	Specify a regular expression that should match the messages, methods and actions you wish to include in a document.
"interactionLevel"	Optional	Specify the level of abstraction in generated documents. Choose between "component", "module", "subsystem", "system" and "env" to select the level of interactions that should be shown. For example, if you select "subsystem", the document will focus on only inter subsystem interactions.
"columnWidth"	Optional (defaults to "large" when omitted)	Spacing between objects axis in a sequence diagram. Choose between "small", "medium" and "large".
"remarkWidth"	Optional (defaults to "large" when omitted)	Width of the remark column on the right side of a sequence diagram. Choose between "small", "medium" and "large".

3.2.1 Required Document Fields for Different Document Types

Sequence Diagram	• Use case or message sequence chart in Adobe PDF or Microsoft Word Picture (EMF) format.
	• Interactions between entities are listed in time sequence.
	• All scenarios defined in the Scenario Project are included in the diagram.
	
	Example:
	`"documents":`
	`[`
	`  {`
	`     "documentName":   "Sample PDF sequence diagram"`
	`  },`
	`  {`
	`     "documentName":   "Sample MS Word sequence diagram",`
	`     "documentFormat": "emf"`

	`  }`
	`]`

Interface Sequence Diagram	• Sequence diagram that just include interactions with a specified entity or entity type.
	• You can generate the following:
	  ○ System interface diagrams
	  ○ Subsystem interface diagrams
	  ○ Module interface diagrams
	  ○ Subsystem interface diagrams
	  ○ Eternal or dynamic object interaction diagrams
	
	Example:
	"documents":
	`[`
	`  {`
	`     "documentName":    "Desert City type interactions",`
	`     "interfaceFilter": "DesertCity"`
	`  },`
	`  {`
	`     "documentName":    "Las Vegas interactions",`
	`     "interfaceFilter": "Las Vegas"`
	`  }`
	`]`

Interaction Sequence Diagram	• Sequence diagrams that allow you to zoom in and out of different levels of abstraction. You can also generate documents that just contain message interactions for a specific tag (Refer to
	• Message Tagging).
	• You can generate the following:
	  ○ Inter system interaction diagram
	  ○ Inter subsystem interaction diagram
	  ○ Inter module interaction diagram
	  ○ Inter component interaction diagram
	
	Example:
	"documents":
	`[`
	`  {`
	`     "documentName":     "System level sequence diagram",`
	`     "interactionLevel": "system"`
	`  },`
	`  {`
	`     "documentName":     "Component level interactions",`
	`     "interactionLevel": "component"`
	`  }`
	`]`

Message Filter Sequence Diagrams	• Sequence diagrams listing only messages that contain a specified string or match a regular expression.

	• For example, a regular expression of "{Request}|{Response}" will generate a document with messages that contain the strings "Request" or "Response" in the message or method name.
	
	Example:
	"documents":
	`[`
	`  {`
	`     "documentName": "Request or response messages",`
	`     "regexTextFilter": "{Request}|{Response}"`
	`  },`
	`  {`
	`     "documentName": "Messages containing Attach",`
	`     "textFilter": "Attach"`
	`  }`
	`]`

Unit Test Procedures	• Unit test procedures for a specified object.
	• The scenarios are tabulated as steps of actions and the associated expected results.
	• The procedures may be used for as a guide for code reviews.
	
	Example:
	"documents":
	`[`
	`  {`
	`     "documentName": "Las Vegas Unit Tests",`
	`     "documentType": "unit-tests"`
	`  }`
	`]`

Summary Statistics Document	• Object wise summary of all interactions.
	• The document summary can be generated for all objects or objects within a module or a component.
	
	Example:
	"documents":
	`[`
	`  {`
	`     "documentName": "Object wise summary for all",`
	`     "documentType": "summary-document"`
	`  },`
	`  {`
	`     "documentName":    "Object wise for Las Vegas",`
	`     "documentType":    "summary-document",`
	`     "interfaceFilter": "Las Vegas"`
	`  },`
	`]`

Statistics Document	• Matrix of module, component and object level statistics.

	• Total number of message interactions between two objects can be tracked by following the source object row and destination object column.
	
	"documents":
	`[`
	`  {`
	`     "documentName": "Overall statistics",`
	`     "documentType": "statistics-document"`
	`  }`
	`]`

Collaboration Diagram	• Collaboration Diagram or Context Diagram in Adobe PDF or Microsoft Word Picture (EMF) formats.
	• The time component of the interactions is de-emphasized in these diagrams.
	
	Example:
	"documents":
	`[`
	`  {`
	`     "documentName": "PDF context diagram",`
	`     "documentType": "context-diagram"`
	`  },`
	`  {`
	`     "documentName":   "MS Word collaboration diagram",`
	`     "documentFormat": "emf",`
	`     "documentType":   "collaboration-diagram"`
	`  }`
	`]`

Interface Collaboration Diagram	• Analogous to Interface Sequence Diagrams
	
	Example:
	"documents":
	`[`
	`  {`
	`     "documentName":    "PDF Desert City Context",`
	`     "documentType":    "context-diagram",`
	`     "interfaceFilter": "DesertCity"`
	`  },`
	`  {`
	`     "documentName":    "MS Word Las Vegas collab diagram",`
	`     "documentFormat":  "emf",`
	`     "documentType":    "collaboration-diagram",`
	`     "interfaceFilter": "Las Vegas"`
	`  }`
	`]`

Interaction Collaboration Diagrams	• Analogous to Interaction Sequence Diagrams
	
	Example:
	"documents":
	`[`
	`  {`

	`"documentName" : "PDF Module Context", "documentType" : "context-diagram", "interactionLevel": "module" }, { "documentName" : "MS Word Component collab diagram", "documentFormat" : "emf", "documentType" : "collaboration-diagram", "interactionLevel": "Component" } ]`
Message Filter Collaboration Diagram	• Analogous to Message Filter Sequence Diagram Example: "documents": `[ { "documentName" : "PDF Request-Response Context", "documentType" : "context-diagram", "regexTextFilter" : "{Request}|{Response}" }, { "documentName" : "MS Word Attach collab diagram", "documentFormat": "emf", "documentType" : "collaboration-diagram", "textFilter" : "Attach" } ]`
XML Document	• Export to XML • The generated XML can be transformed into code and other documents using XSLT.
Interface XML Document	• Analogous to Interface Sequence Diagrams
Interaction XML Document	• Analogous to Interaction Sequence Diagrams
Message Filter XML Document	• Analogous to Message Filter Sequence Diagram

3.3 Predefined Sequence Diagram Formatting Options

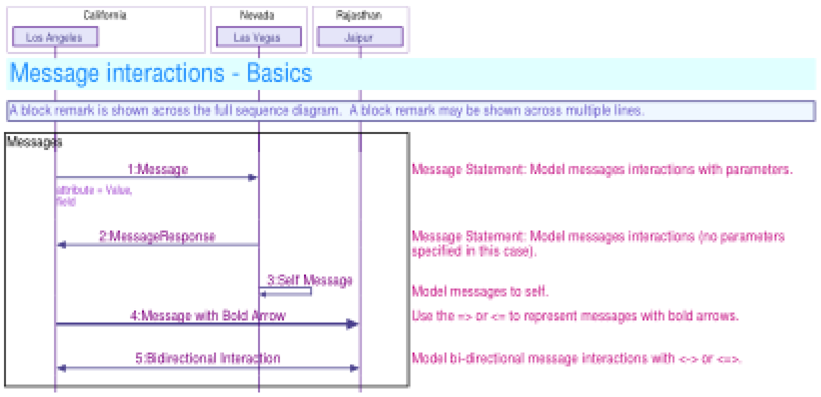
The starter projects created with the "EventStudio: New Scenario Project" command support convenient defines that you can use to quickly adjust the layout of generated sequence diagrams.

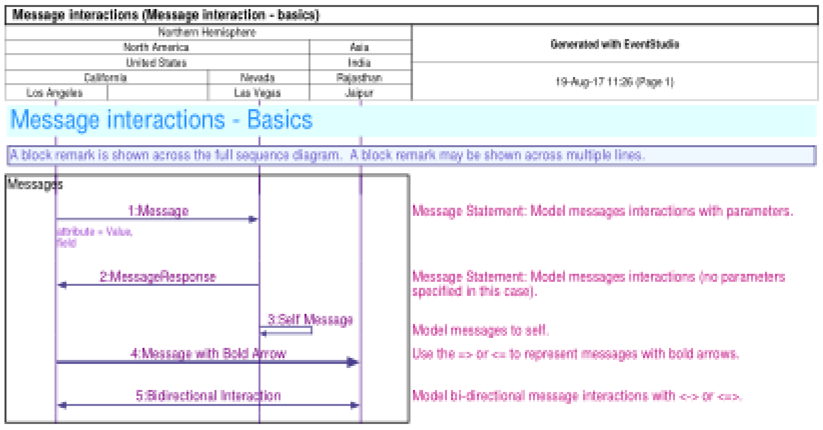
3.3.1 Choose Sequence Diagram Header and Remark format

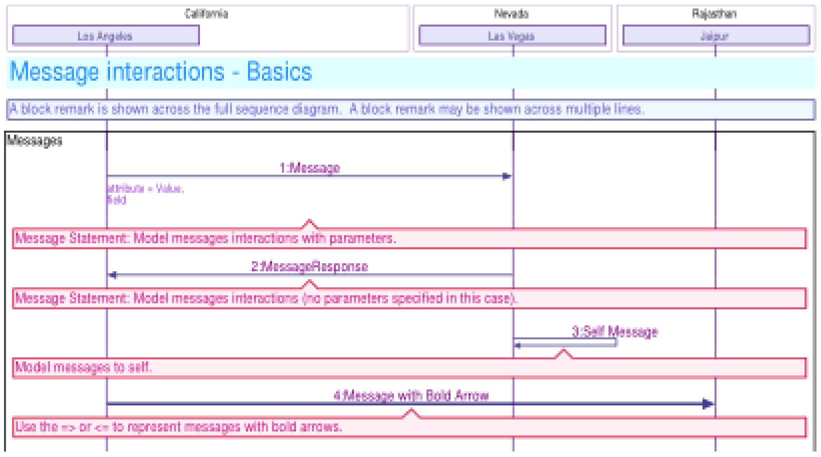
The following options are available for choosing header and remark layout:

- Choose between a single tier header and a full multi-tier header (`USE_MULTI_TIER_HEADER`).
- Choose between remarks on the right side or callout remarks (`USE_CALLOUT_REMARKS`).

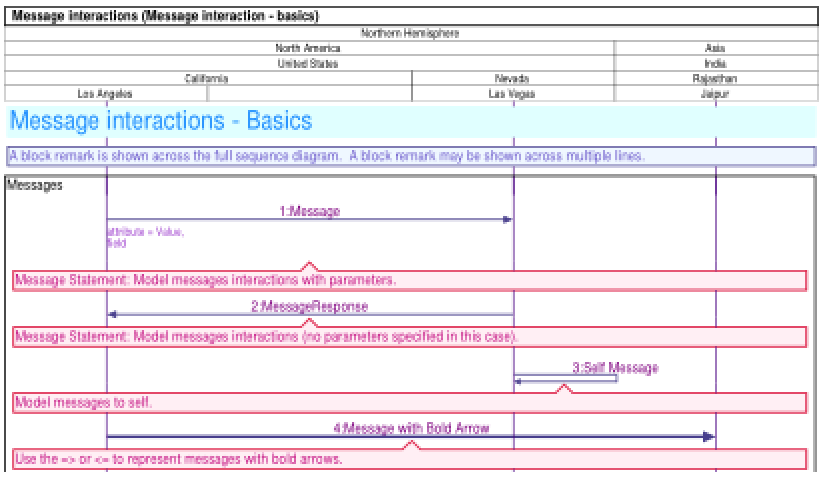
Document definition	Generated sequence diagram

`"documents": [`
`  {`
`    "documentName":"tier1"`
`  }`
`]`
	 California
	Los Angeles
	Nevada
	Las Vegas
	Rajasthan
	Jaipur
	Message interactions - Basics
	A block remark is shown across the full sequence diagram. A block remark may be shown across multiple lines.
	Messages
	1:Message
	attribute = Value,
	field
	Message Statement: Model messages interactions with parameters.
	2:MessageResponse
	Message Statement: Model messages interactions (no parameters specified in this case).
	3:Self Message
	Model messages to self.
	4:Message with Bold Arrow
	Use the => or <= to represent messages with bold arrows.
	5:Bidirectional Interaction
	Model bi-directional message interactions with <-> or <=>.

`"documents": [`
`  {`
`    "documentName":"tier1",`
`    "defines": [`
`    "USE_MULTI_TIER_HEADER"`
`    ]`
`  }`
`]`
	 Message interactions (Message interaction - basics)
	Northern Hemisphere
	North America
	Asia
	United States
	India
	California
	Nevada
	Rajasthan
	Los Angeles
	Las Vegas
	Jaipur
	Generated with EventStudio
	19-Aug-17 11:26 (Page 1)
	Message interactions - Basics
	A block remark is shown across the full sequence diagram. A block remark may be shown across multiple lines.
	Messages
	1:Message
	attribute = Value,
	field
	Message Statement: Model messages interactions with parameters.
	2:MessageResponse
	Message Statement: Model messages interactions (no parameters specified in this case).
	3:Self Message
	Model messages to self.
	4:Message with Bold Arrow
	Use the => or <= to represent messages with bold arrows.
	5:Bidirectional Interaction
	Model bi-directional message interactions with <-> or <=>.

`"documents": [`
`  {`
`    "documentName":"tier1",`
`    "defines": [`
`      "USE_CALLOUT_REMARKS"`
`    ]`
`  }`
`]`
	 California
	Los Angeles
	Nevada
	Las Vegas
	Rajasthan
	Jaipur
	Message interactions - Basics
	A block remark is shown across the full sequence diagram. A block remark may be shown across multiple lines.
	Messages
	1:Message
	attribute = Value,
	field
	Message Statement: Model messages interactions with parameters.
	2:MessageResponse
	Message Statement: Model messages interactions (no parameters specified in this case).
	3:Self Message
	Model messages to self.
	4:Message with Bold Arrow
	Use the => or <= to represent messages with bold arrows.

```
"documents": [
    {
      "documentName":"tier1",
      "defines": [
      "USE_CALLOUT_REMARKS",
      "USE_MULTI_TIER_HEADER"
      ]
    }
  ]
```


Message interactions (Message interaction - basics)
Northern Hemisphere
North America
United States
California
Los Angeles
Nevada
Las Vegas
Asia
India
Rajasthan
Jaipur
Message interactions - Basics
A block remark is shown across the full sequence diagram. A block remark may be shown across multiple lines.
Messages
1:Message
attribute = Value,
field
Message Statement: Model messages interactions with parameters.
2:MessageResponse
Message Statement: Model messages interactions (no parameters specified in this case).
3:Self Message
Model messages to self.
4:Message with Bold Arrow
Use the => or <= to represent messages with bold arrows.

3.3.2 Choose Between Document, Presentation and Poster Layouts

Three predefined layout modes are in the inc.FDL file for the project:

Regular sequence diagram	Presentation	Poster
• US Letter paper size • Portrait mode	• US legal paper size • Landscape mode	• US B paper size • Portrait

`"documents": [`
` {`
`  "documentName":"tier1"`
` }`
`]`
	`"documents": [`
	`{`
	` "documentName":"tier1",`
	`  "defines": [`
	`   "PRESENTATION"`
	`   ]`
	` }`
	`]`
		`"documents": [`
		`{`
		` "documentName":"tier1",`
		`  "defines": [`
		`   "POSTER"`
		`   ]`
		` }`
		`]`

The above definitions are the defaults. You can change these by just editing the inc.FDL file defaults.

```
#define DEFAULT_PAPER_SIZE "letter"
/* "letter", "legal", "B", "C", "A4", or "A3" */

#define DEFAULT_ORIENTATION "portrait-only"
/* "portrait-only" or "landscape-only" */

#define PRESENTATION_PAPER_SIZE "legal"
/* "letter", "legal", "B", "C", "A4", or "A3" */

#define PRESENTATION_ORIENTATION "landscape-only"
/* "portrait-only" or "landscape-only" */
```

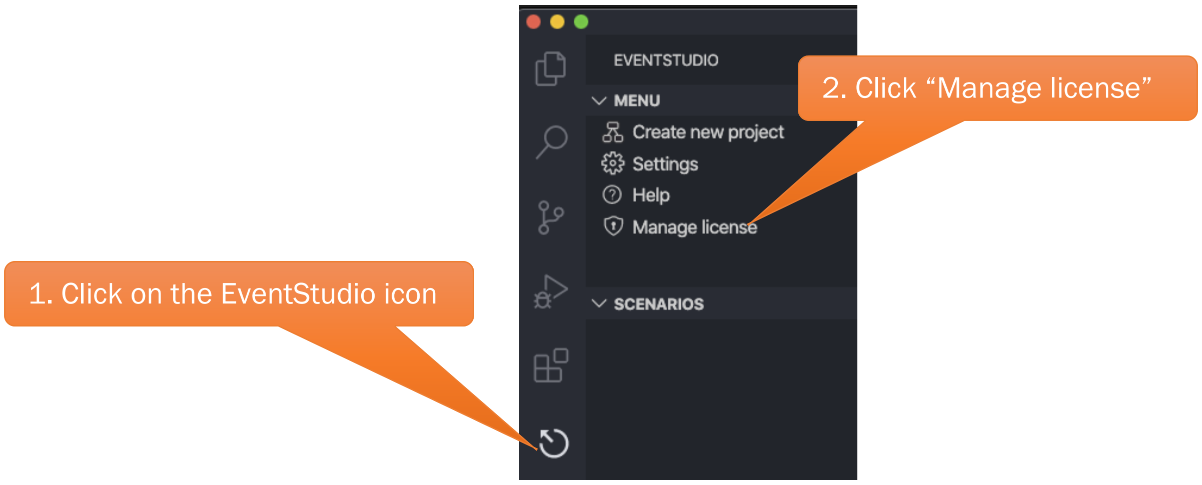
```
#define POSTER_PAPER_SIZE "B"
/* "letter", "legal", "B", "C", "A4", or "A3" */

#define POSTER_ORIENTATION "portrait-only"
/* "portrait-only" or "landscape-only" */
```

4 HOW TO...

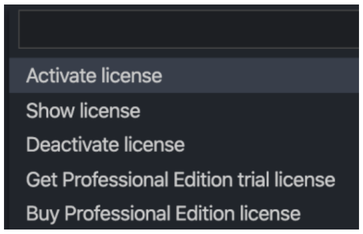
4.1 Activate a License

Follow this procedure to activate a full or trial license.

1. Click the EventStudio icon to display the EventStudio sidebar. Then click "Manage license".



2. Click on the "Activate license" option.


Activate license
Show license
Deactivate license
Get Professional Edition trial license
Buy Professional Edition license

3. Follow the prompts to upload the license file.

4.2 Add New Scenarios and Documents

Adding a new scenario is best explained with an example.

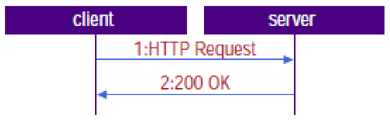
4.2.1 Before Adding Scenario

Let's start with a project that contains the following two files.

project.scn.json	**http.FDL**

{
  "scenarios": [
  {
   "modelPath": "model/http.FDL",
   "scenarioName": "200 OK",
   "defines": [**"SUCCESSFUL"**]
  }
 ],
 "documents": [
  {
    "documentName": "summary-flow"
  }
 ]
}
	eternal: client, server
	feature "Web Browsing" {
	
	#ifdef STARTUP
	    SYN: client -> server
	    "SYN+ACK": client <- server
	    ACK: client -> server
	#endif
	
	   "HTTP Request": client -> server
	
	#ifdef SUCCESSFUL
	   "200 OK": client <- server
	#else
	   "404 Not Found": client <-
	server
	#endif
	}

At this point the generated sequence diagram is generated for one scenario. The diagram looks like:

Scenario	**summary-flow.pdf**
"200 OK"	

4.2.2 After Adding Scenario

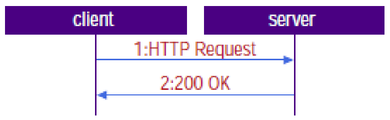
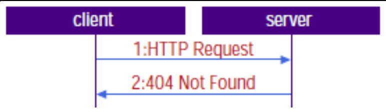
A new scenario addition is shown in the highlighted text. The "modelPath" and "scenarioName" need to be specified for a new scenario. You may also specify additional preprocessor defines that are needed for a scenario. In the highlighted case, not defining a preprocessor selected the **#else-#endif** part of the FDL file.

project.scn.json	**http.FDL**

{
  "scenarios": [
  {
   "modelPath": "model/http.FDL",
   "scenarioName": "200 OK",
   "defines": [**"SUCCESSFUL"**]
  },
	eternal: client, server
	feature "Web Browsing" {
	
	#ifdef STARTUP
	    SYN: client -> server
	    "SYN+ACK": client <- server
	    ACK: client -> server

```
  {
    "modelPath": "model/http.FDL",
    "scenarioName": "404 Not found"
  }
 ],
 "documents": [
  {
    "documentName": "summary-flow"
  }
 ]
}
```
	```
	#endif

	   "HTTP Request": client -> server

	#ifdef SUCCESSFUL
	   "200 OK": client <- server
	#else
	   "404 Not Found": client <-
	server
	#endif
	}
	```

At this point the generated sequence diagram shows two scenarios in the summary-flow.pdf file.

Scenario	summary-flow.pdf
"200 OK"	
"404 Not found"	

4.2.3 Add a Document

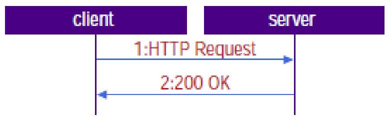
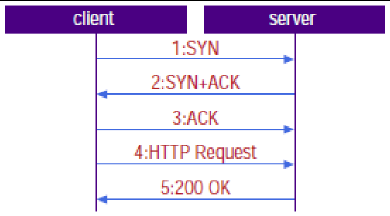
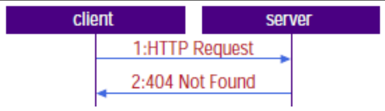
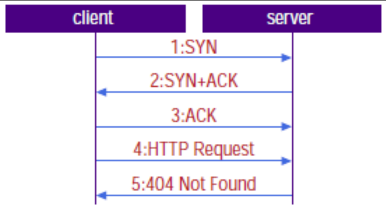
Now let's add a document to our project. The highlighted text adds a new document to the project. This document is defined with the "documentName" and "defines" fields. The "defines" field includes the "STARTUP" define. This results in generation of the "detailed-flow.pdf" file with both scenarios showing the interactions enclosed in **#ifdef STARTUP** and **#endif**

project.scn.json	http.FDL

```
{
 "scenarios": [
  {
    "modelPath": "model/http.FDL",
    "scenarioName": "200 OK",
    "defines": [ "SUCCESSFUL" ]
  },
  {
    "modelPath": "model/http.FDL",
    "scenarioName": "404 Not found"
  }
 ],
 "documents": [
```
	```
	eternal: client, server
	feature "Web Browsing" {

	#ifdef STARTUP
	    SYN: client -> server
	    "SYN+ACK": client <- server
	    ACK: client -> server
	#endif

	   "HTTP Request": client -> server

	#ifdef SUCCESSFUL
	```

```
  {
    "documentName": "summary-flow"
  },
  {
    "documentName": "detailed-flow",
    "defines": [ "STARTUP" ]
  }
 ]
}
```
	```
	   "200 OK": client <- server
	#else
	   "404 Not Found": client <-
	server
	#endif
	}
	```

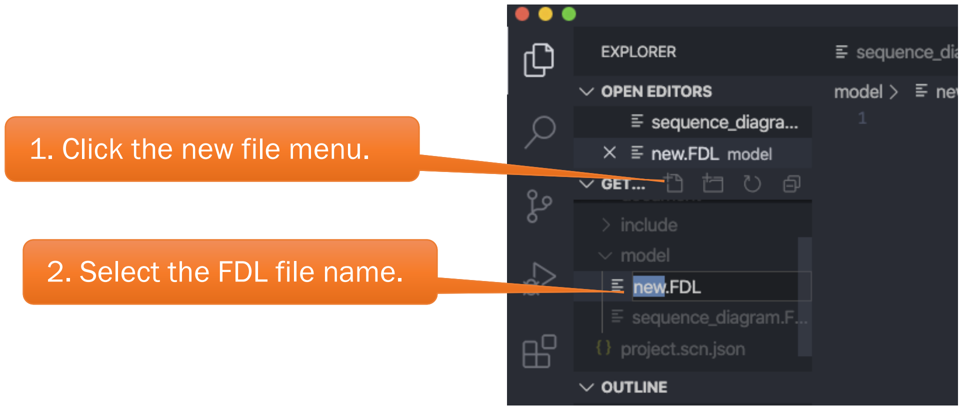
EventStudio now generates two documents ("summary-flow.pdf" and "detailed-flow.pdf").

Scenario	summary-flow.pdf	detailed-flow.pdf
"200 OK"		
"404 Not found"		 client server 1:SYN 2:SYN+ACK 3:ACK 4:HTTP Request 5:404 Not Found

Learn more about the project files in the Scenarios and Documents chapter.

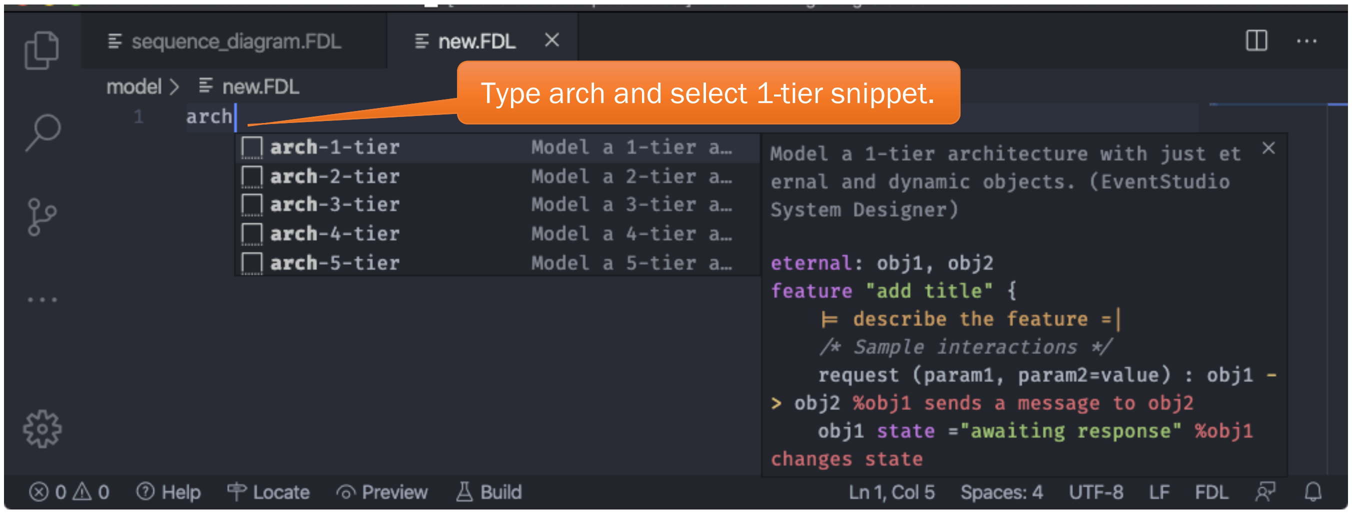
4.3 Add a New FDL File to the Project

1. Select the model directory and then click the new document icon. Alternatively, right-click on the model folder and select new file.



2. Create new.FDL.

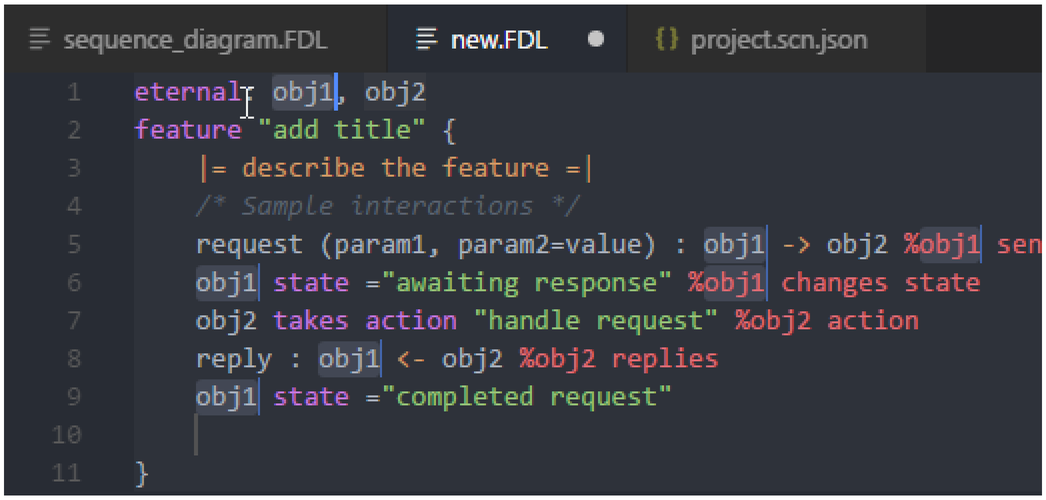
3. Type arch to select a 1-tier architecture snippet.



```
arch
arch-1-tier     Model a 1-tier a…
arch-2-tier     Model a 2-tier a…
arch-3-tier     Model a 3-tier a…
arch-4-tier     Model a 4-tier a…
arch-5-tier     Model a 5-tier a…
```

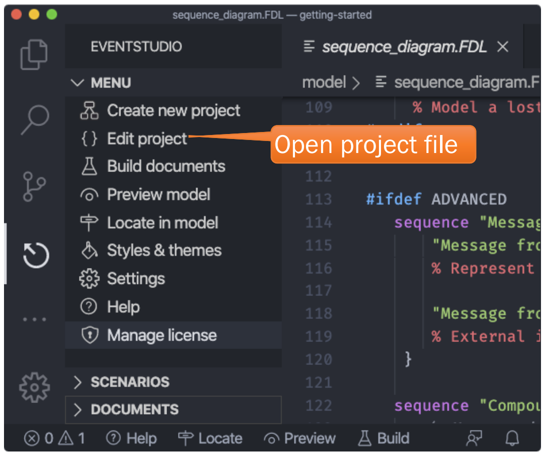
```
Model a 1-tier architecture with just et
ernal and dynamic objects. (EventStudio
System Designer)

eternal: obj1, obj2
feature "add title" {
    |= describe the feature =|
    /* Sample interactions */
    request (param1, param2=value) : obj1 -
> obj2 %obj1 sends a message to obj2
    obj1 state ="awaiting response" %obj1
changes state
```

4. Use tabs to replace obj1 and obj2 in the snippet. Multiple occurrences of obj1 and obj2 are automatically substituted.



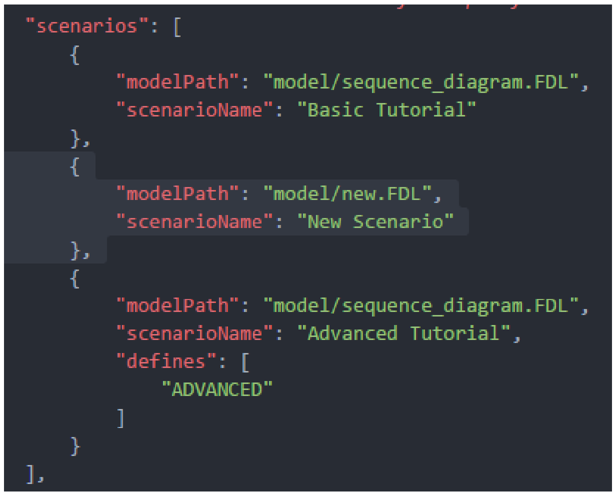
```
eternal: obj1, obj2
feature "add title" {
    |= describe the feature =|
    /* Sample interactions */
    request (param1, param2=value) : obj1 -> obj2 %obj1 sen
    obj1 state ="awaiting response" %obj1 changes state
    obj2 takes action "handle request" %obj2 action
    reply : obj1 <- obj2 %obj2 replies
    obj1 state ="completed request"

}
```

5. Now open the project file.



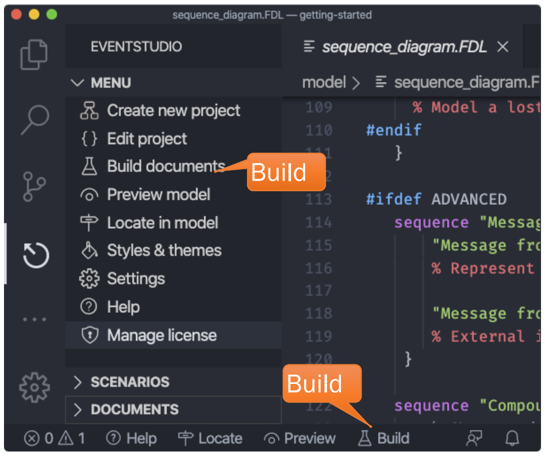
```
            % Model a lost

    #ifdef ADVANCED
        sequence "Messag
            "Message fro
            % Represent

            "Message fro
            % External i
        }

        sequence "Compou
```

6. Add the new scenario to the file. The scenarios are added to the document in the order defined here.



```
"scenarios": [
    {
        "modelPath": "model/sequence_diagram.FDL",
        "scenarioName": "Basic Tutorial"
    },
    {
        "modelPath": "model/new.FDL",
        "scenarioName": "New Scenario"
    },
    {
        "modelPath": "model/sequence_diagram.FDL",
        "scenarioName": "Advanced Tutorial",
        "defines": [
            "ADVANCED"
        ]
    }
],
```

7. Click the Build button.



```
            % Model a lost
    #endif
        }

    #ifdef ADVANCED
        sequence "Messag
            "Message fro
            % Represent

            "Message fro
            % External i
        }

        sequence "Compou
```

4.4 Invoke EventStudio from Command Line

4.4.1 Add EventStudio to the Path

Windows

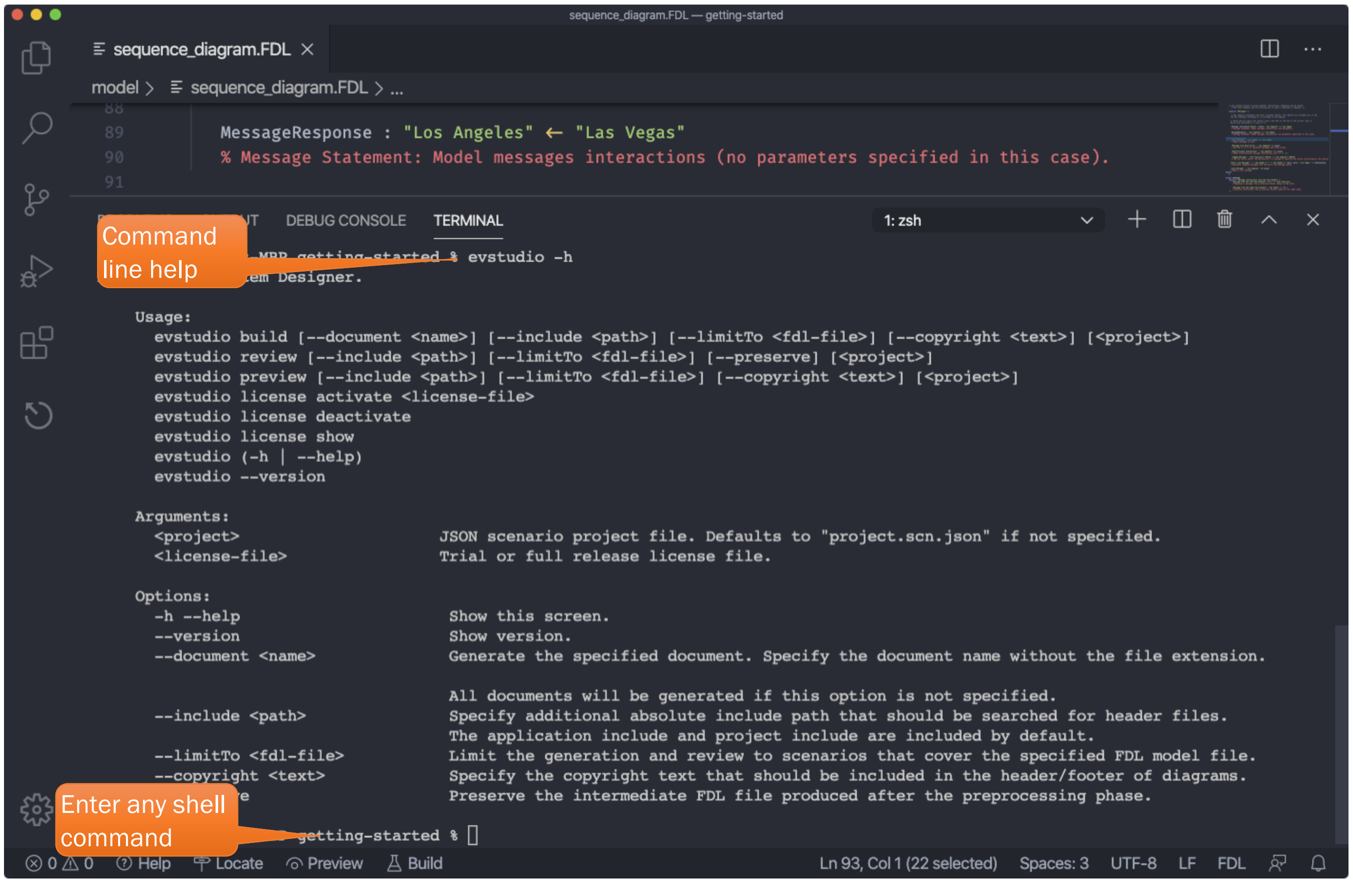
1. Navigate to the `%USERPROFILE%\.vscode\extensions` directory.
2. Change to the EventStudio extension directory.
   a. The directory depends upon the version of the EventStudio extension. For example, the path for the 8.0.2 release, the complete path is: `%USERPROFILE%\.vscode\extensions\eventhelix.eventstudio-8.0.2`
3. Add this directory to the search path.

macOS

1. Navigate to the `~/.vscode/extensions/eventhelix.eventstudio-8.0.2` directory.
2. Change to the EventStudio extension directory.
   a. The directory depends upon the version of the EventStudio extension. For example, the path for the 8.0.2 release, the complete path is: `~/.vscode/extensions/eventhelix.eventstudio-8.0.2`
3. Add this directory to the search path.

4.4.2 Open the Integrated Terminal in Visual Studio Code

1. Open the project in Visual Studio Code.
2. Open the integrated terminal with the `Ctrl+`` (i.e. `Ctrl+backtick`).
3. Open the command window and type: `evstudio -h`



```
    MessageResponse : "Los Angeles" ← "Las Vegas"
    % Message Statement: Model messages interactions (no parameters specified in this case).
```

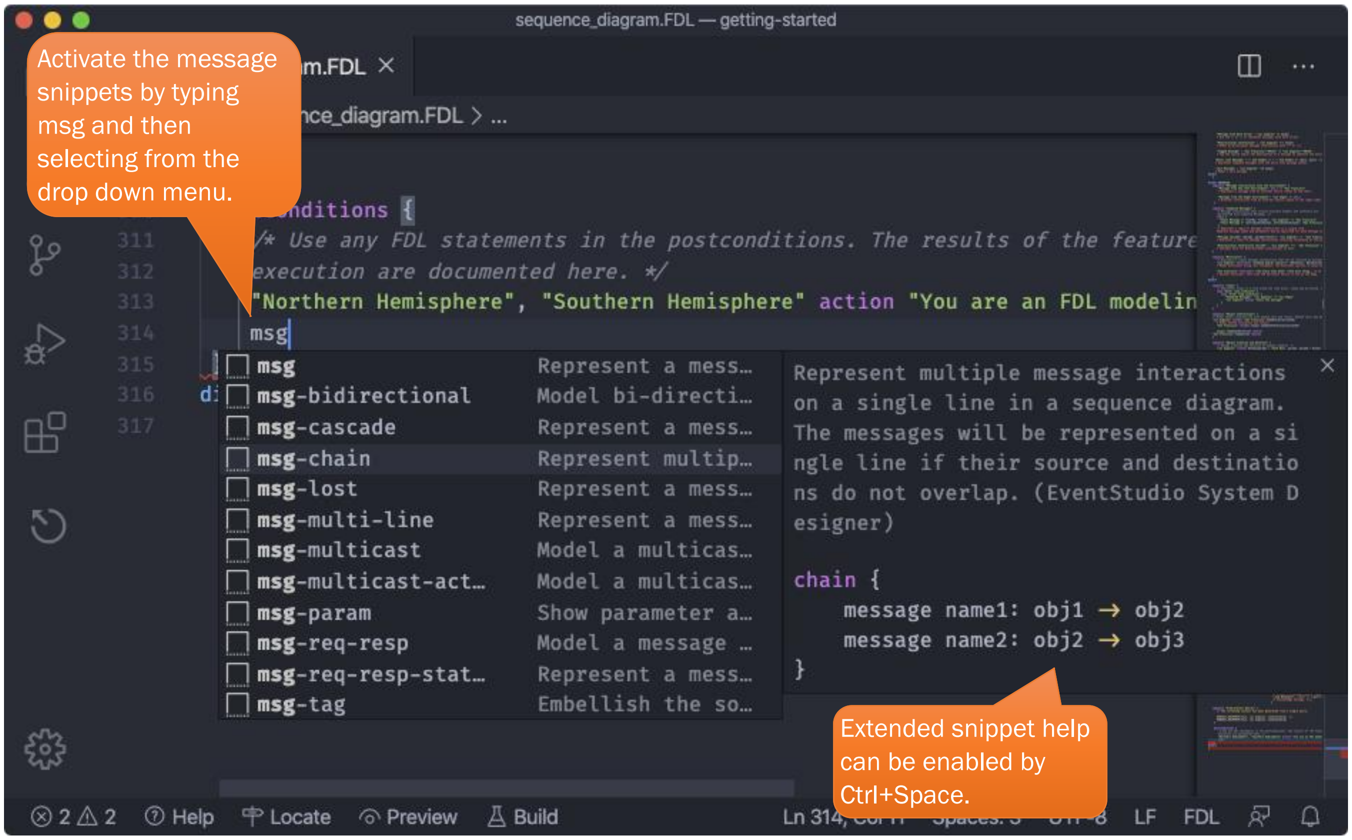
```
MBP getting-started % evstudio -h
em Designer.

  Usage:
    evstudio build [--document <name>] [--include <path>] [--limitTo <fdl-file>] [--copyright <text>] [<project>]
    evstudio review [--include <path>] [--limitTo <fdl-file>] [--preserve] [<project>]
    evstudio preview [--include <path>] [--limitTo <fdl-file>] [--copyright <text>] [<project>]
    evstudio license activate <license-file>
    evstudio license deactivate
    evstudio license show
    evstudio (-h | --help)
    evstudio --version

  Arguments:
    <project>                  JSON scenario project file. Defaults to "project.scn.json" if not specified.
    <license-file>             Trial or full release license file.

  Options:
    -h --help                   Show this screen.
    --version                   Show version.
    --document <name>           Generate the specified document. Specify the document name without the file extension.

                                All documents will be generated if this option is not specified.
    --include <path>            Specify additional absolute include path that should be searched for header files.
                                The application include and project include are included by default.
    --limitTo <fdl-file>        Limit the generation and review to scenarios that cover the specified FDL model file.
    --copyright <text>          Specify the copyright text that should be included in the header/footer of diagrams.
               e                Preserve the intermediate FDL file produced after the preprocessing phase.

getting-started % 
```

4. Now enter `evstudio build` to generate documents.
5. Explore the other commands:
    a. `review`: Only review the FDL model in the project. No documents are generated.
    b. `preview`: Generate a preview document. You can limit the review to a single FDL file using the `limitTo` switch.
    c. `convert`: Convert EventStudio 6 projects to the new release.

4.5 Insert FDL Snippets

You can insert FDL code snippets to define commonly occurring design patterns. This involves:

1. Typing the snippet prefix in the editor (msg snippet is show below).


sequence_diagram.FDL — getting-started
m.FDL
nce_diagram.FDL > ...
Activate the message snippets by typing msg and then selecting from the drop down menu.
Extended snippet help can be enabled by Ctrl+Space.
⊗ 2 ⚠ 2
Help
Locate
Preview
Build
Ln 314, Col 11
LF
FDL

```
nditions {
    /* Use any FDL statements in the postconditions. The results of the feature
    execution are documented here. */
    "Northern Hemisphere", "Southern Hemisphere" action "You are an FDL modelin
    msg
```

```
msg                       Represent a mess…
msg-bidirectional         Model bi-directi…
msg-cascade               Represent a mess…
msg-chain                 Represent multip…
msg-lost                  Represent a mess…
msg-multi-line            Represent a mess…
msg-multicast             Model a multicas…
msg-multicast-act…        Model a multicas…
msg-param                 Show parameter a…
msg-req-resp              Model a message …
msg-req-resp-stat…        Represent a mess…
msg-tag                   Embellish the so…
```

```
Represent multiple message interactions
on a single line in a sequence diagram.
The messages will be represented on a si
ngle line if their source and destinatio
ns do not overlap. (EventStudio System D
esigner)

chain {
    message name1: obj1 → obj2
    message name2: obj2 → obj3
}
```

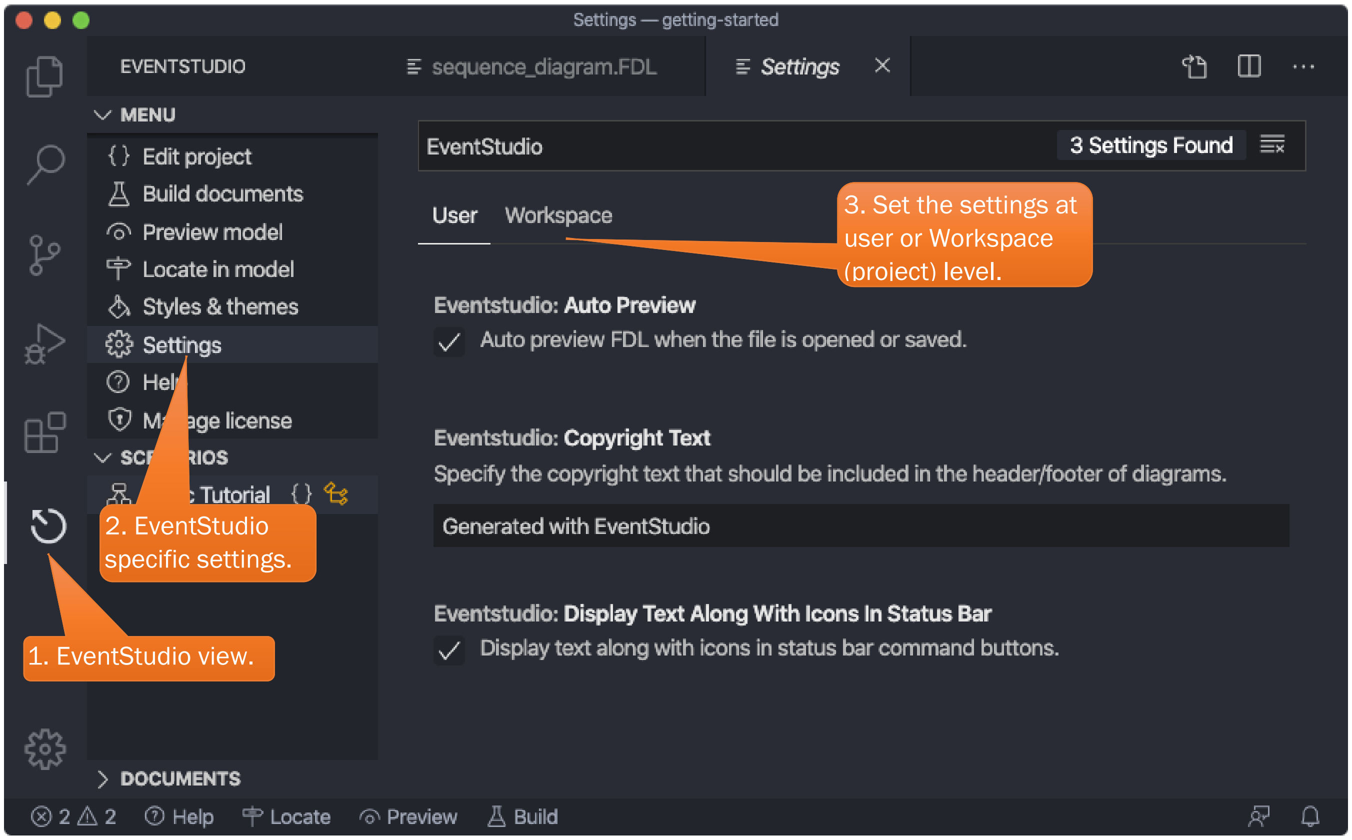
2. Select the snippet and fill in the blanks.
3. Explore the following built in snippet prefixes:

Snippet Prefix	Description
`arch`	Choose single tier to 5-tier architecture snippets
`msg`	Choose between different flavors of message snippets
`invoke`	Method invoke and return snippets
`create`	Create-delete objects

`action`	Action snippets
`timer`	Timer snippets
`resource`	Resource snippets
`state`	State transition snippets
`sequence`	Sequence group snippets

4.6 Change EventStudio Configuration

1. Select the Settings menu from the EventStudio sidebar.
2. You can choose between user level and workspace (i.e. project) level settings.



EventStudio

Eventstudio: Auto Preview
✓ Auto preview FDL when the file is opened or saved.

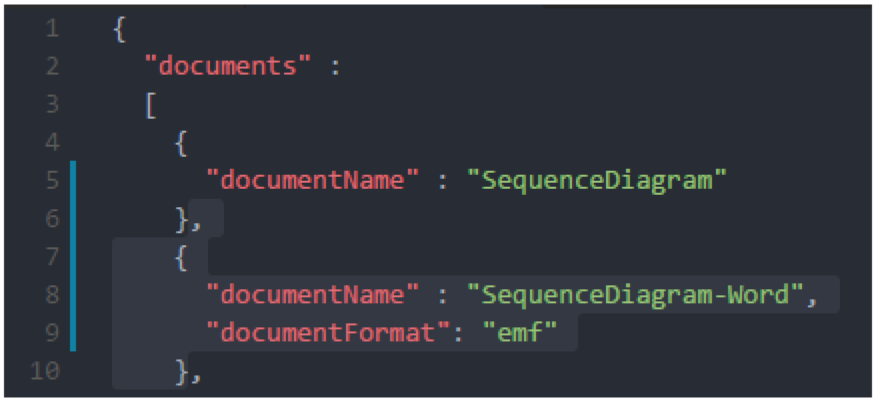
Eventstudio: Copyright Text
Specify the copyright text that should be included in the header/footer of diagrams.
Generated with EventStudio

Eventstudio: Display Text Along With Icons In Status Bar
✓ Display text along with icons in status bar command buttons.

4.7 Export to Microsoft Word

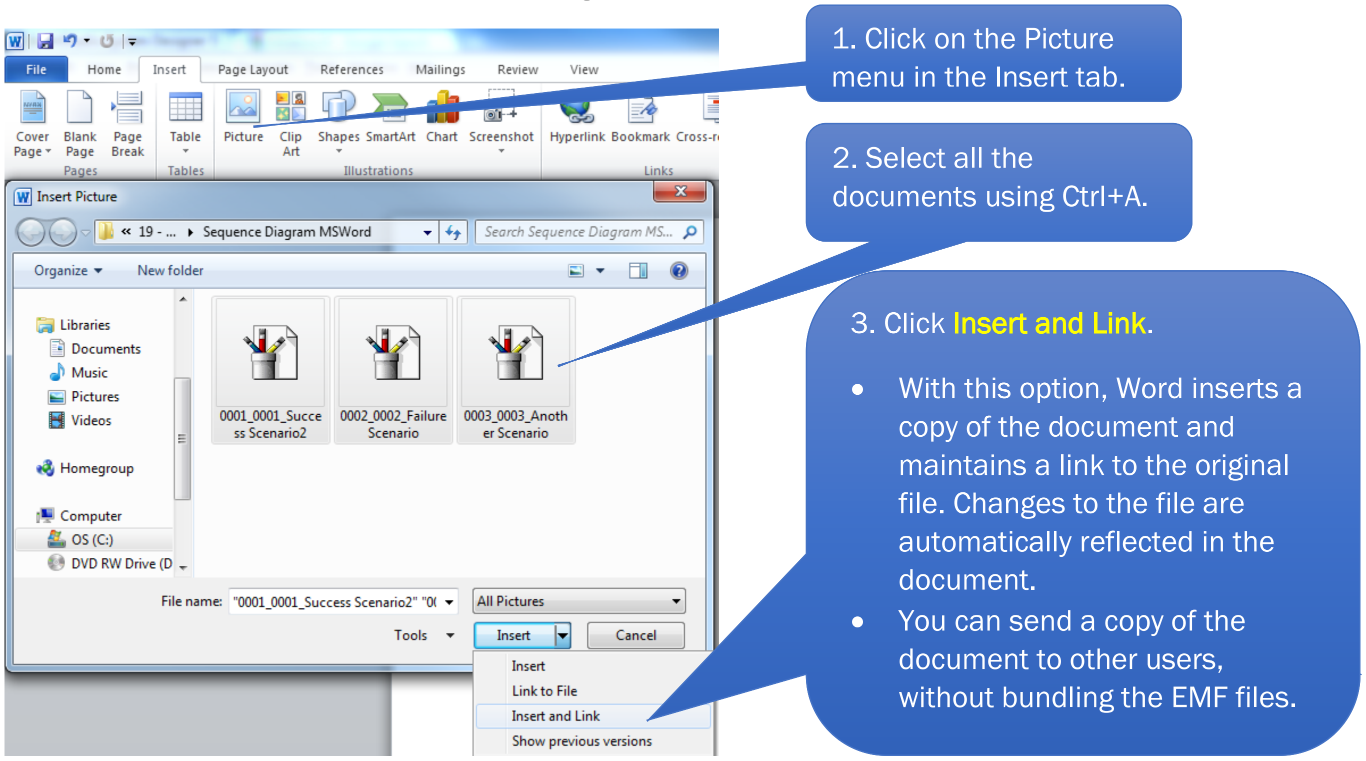
Windows only.

1. Create a Microsoft Word Picture (EMF) as shown in Add a Document.
   a. Just set the "documentFormat" to "emf" (as shown in the following figure).



```
{
  "documents" :
  [
    {
      "documentName" : "SequenceDiagram"
    },
    {
      "documentName" : "SequenceDiagram-Word",
      "documentFormat": "emf"
    },
```

2. Open the Microsoft Word document where you wish to insert the document.
3. Position the cursor in the document where you want to insert the "Word Picture EMF" files.
4. Follow the instructions specified along with the screenshot:



1. Click on the Picture menu in the Insert tab.

2. Select all the documents using Ctrl+A.

3. Click Insert and Link.

- With this option, Word inserts a copy of the document and maintains a link to the original file. Changes to the file are automatically reflected in the document.
- You can send a copy of the document to other users, without bundling the EMF files.

Notes:

1. The document in "Word Picture EMF" graphics format is generated as a directory of the name same as that of the document in the path where the scenario project is present.
2. The document consists of a set of files in this directory. Each page of the document is represented by a file. The file naming is: <scenario number>_<page within scenario>_<scenario name>.emf.

Example

Consider a Scenario Project in the directory "C:\MyDoc\ " with "Docking" and "Undocking" scenarios. The document "Spaceport Collaboration Diagram Word" is generated in the directory "C:\MyDoc\documents\Spaceport Collaboration Diagram Word\" where the files are sequentially defined as 0001_0001_Docking.emf, 0002_0001_Docking.emf, 0002_0001_Undocking.emf and so on.

4.8 Convert EventStudio 6 Scenario Projects to JSON

Windows only.

Use the EventStudio command-line to convert.

For example, the following command converts the EventStudio 6 project "Message sequence charts.scn" and stores the reorganized project into a different directory (recommended).

```
evstudio convert "Message sequence charts.scn" --reorganize "..\MSC7"
```

If you wish to convert without changing the FDL file paths, omit the reorganize switch. For example:

```
evstudio convert "Message sequence charts.scn"
```

5 FDL REFERENCE

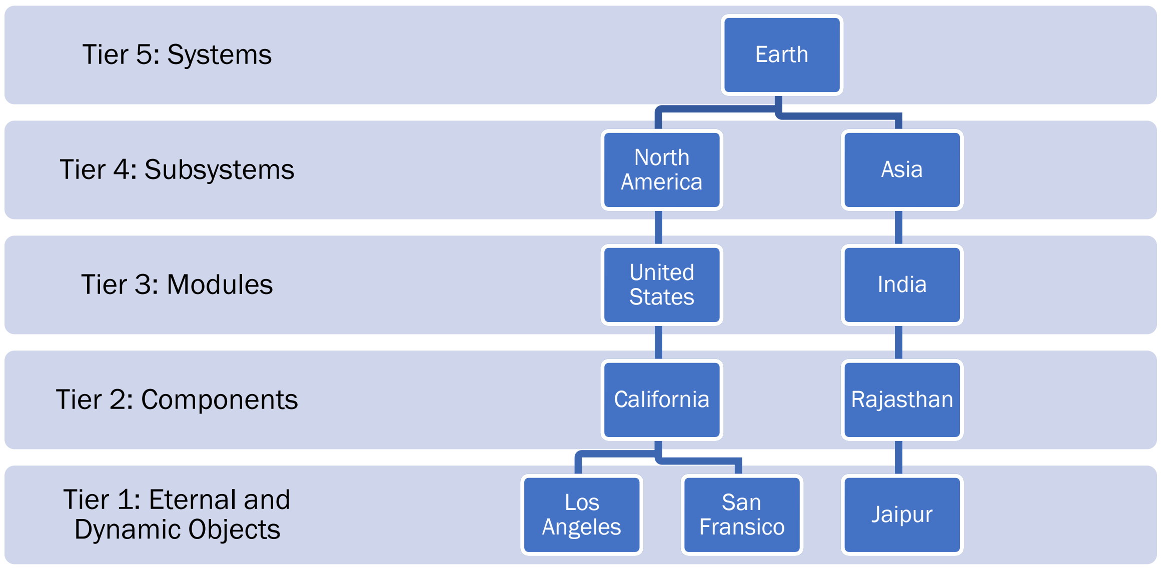
This chapter provides a comprehensive reference to FDL. The following topics are covered:

- Architecture and Feature Definition
- Multiple Scenario Modeling
- Object Interactions
- Object Creation and Deletion
- Timers
- Resource Allocations
- State Transitions
- Remarks
- Preprocessor
- Styles, Themes and Document Layout

5.1 Architecture and Feature Definition

5.1.1 System, Subsystem, Module and Component declaration

EventStudio lets you represent hierarchical architectures like the one shown below.



The FDL representation of the architecture is shown below.

```
system: Earth
subsystem: "North America" in Earth, Asia in Earth
module: "United States" in "North America”
module: India in Asia
component: California in "United States”
component: Rajasthan in India
eternal: "Los Angeles" in California, "San Francisco" in California
eternal: Jaipur in Rajasthan
```

You can choose to model a complex system as a 5-tier architecture. At the other end of the spectrum, you may just use a single tier architecture.

The architecture definition options are shown in the following table:

	5 Tier	4 Tier	3 Tier	2 Tier	Single Tier
system	Present				
subsystem	Present	Present			
module	Present	Present	Present		
component	Present	Present	Present	Present	
eternal or dynamic object	Present	Present	Present	Present	Present

5.1.2 Eternal Object declaration

This statement is used to declare an object that is created at system startup and exists throughout the life of the system. An eternal object declaration statement must be defined before the feature block.

5.1.2.1 EXAMPLES

```
eternal: call_mgr
```
An eternal object call_mgr is declared at the topmost level in two-tier decomposition architecture.

```
eternal: call_mgr in frontend
```
An eternal object call_mgr is declared inside the frontend component.

```
eternal: call_mgr1:call_mgr in frontend1
eternal: call_mgr2:call_mgr in frontend2
```
Two instances of eternal objects of the type call_mgr. If you generated an interface sequence diagram for call_mgr, it will include interactions involving call_mgr1 and call_mgr2.

```
eternal: "I-CSCF" in IMS, "S-CSCF" in IMS
```
Eternal objects are declared as strings.

```
eternal: [object_style] sip_call_manager in server
```
object_style is applied to the sip_call_manager (For details refer section 5.13.1).

5.1.3 Dynamic Object declaration

This statement is used to declare an object that will be dynamically created (5.5.1) and deleted (5.5.2).

5.1.3.1 EXAMPLES

```
dynamic: call in frontend
feature "Call Setup" {
      msg1: phone -> call_mgr
      call_mgr creates call
      dialtone: call -> phone
      ...
      call_over: call -> call_mgr
      call_mgr deletes call
}
```

Here, a dynamic object call is declared inside the frontend component. The Call Setup feature shows that the call_mgr object creates call object. Once the call is over, call_mgr deletes the call object.

`dynamic: call1:call in frontend1, call2:call in frontend2`
Two instances of dynamic objects of the same type.

```
dynamic: "Originating Call" in "SIP Server"
dynamic: "Terminating Call" in "SIP Server"
```

Dynamic objects declared as strings.

```
dynamic: ISUPCall | SIPCall in "Signaling Gateway"
```

This is an example of two dynamic objects sharing the same axis. Only one of these objects can be active at a time. Organizing dynamic objects this way is useful when different scenarios instantiate different objects and dedicating a separate instance axis for each object would lead to space wastage.

```
dynamic: [call_style] sip_call in client
dynamic: [call_style] h323_call in server
```

call_style is applied to the sip_call_manager (For details refer section 5.13.1).

5.1.4 Type declaration

FDL allows defining types for a system, subsystem, module, component, eternal object or dynamic object. This is used to specify that two or more different entities are really instances of the same entity type. A type declaration statement can be put only inside the FDL declaration block at the beginning of the FDL file. There can be no type declaration statement inside the feature {} block in the FDL file.

It may be noted that defining the types will have no visual impact on the PDF Sequence Diagrams. The main difference can be seen in the definition of Unit Test Procedures and Interface documents. If the user specifies the type for generating a document, EventStudio will include interfaces of all instances of the given type.

5.1.4.1 EXAMPLES

`module: calling_party:customer, called_party:customer`

`component: "P-CSCF 1":"P-CSCF" in IMS, "P-CSCF 2":"P-CSCF" in IMS`

```
eternal: call_mgr1:call_mgr in frontend1
eternal: call_mgr2:call_mgr in frontend2
```

`dynamic: call1:call in frontend1, call2:call in frontend2`

5.1.5 Environment interactions

FDL allows the user to depict message interactions with the external world. This can be done by sending messages to the left environment (env_l) or right environment (env_r). Environment interactions can be very useful in clearly spelling out what is external to the system being designed.

env_l and env_r are declared internally, thus there is no declaration statement for environments. env_l and env_r can be used in the message statement as source or destination of messages.

5.1.5.1 EXAMPLES

`pick_up_phone: env_l -> phone`

Here, a pick_up_phone message is received from the left environment. This will be depicted as an arrow from the left edge of the document to the axis corresponding to phone.

`call_setup_request: core -> env_r`

Core sends a message to the right environment. This will be depicted as an arrow from the core axis to the right environment.

5.1.6 Feature statement

This statement encloses all the message interactions for a feature. The **feature {}** block follows the declarations in FDL file. A title for the feature should be specified in the feature declaration. An FDL file defines feature interactions for one feature. Hence there can be one feature {} block in an FDL file.

A theme may be associated with a feature {} block. A theme specifies the layout options and the fonts.

The feature title that is specified in the statement is displayed at the top, along the complete width of the output page of the Sequence Diagram.

EventStudio performs various checks on the feature {} block. It warns in the following cases:

- Timers have been left running at feature end.
- Dynamic object is not deleted before the feature ends.
- Resources have not been deallocated at feature end.

5.1.6.1 EXAMPLES

```
feature "Call Setup" {
  msg1: phone -> call_mgr
  msg2: call_mgr -> core
}
```

This feature {} block encloses all the feature interactions for the call setup feature.

```
{MyTheme} feature "Call Setup" {
  msg1: phone -> call_mgr
  msg2: call_mgr -> core
}
```

Here a feature {} block is shown with a theme specification for the feature. The theme specifies the layout options and the font sizes.

5.2 Multiple Scenario Modeling

Systems engineering involves modeling a myriad of alternative and failure scenarios for a feature. With FDL, multiple scenarios can be modeled by just specifying how the alternative and failure scenarios differ from the base scenarios. C-style #ifdef and #ifndef statements should be used to identify how the alternative and error scenarios deviate from the core scenarios.

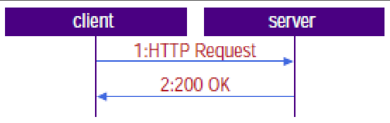
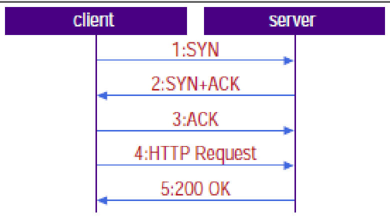
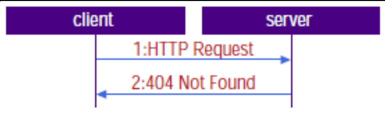
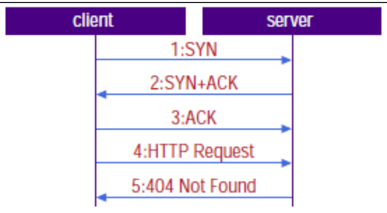
project.scn.json

```
{ "scenarios": [
  {
   "modelPath": "model/http.FDL",
   "scenarioName": "200 OK",
   "defines": [ "SUCCESSFUL" ]
  },
  {
   "modelPath": "model/http.FDL",
   "scenarioName": "404 Not found"
  } ],
 "documents": [
  {
    "documentName": "summary-flow"
  },
  {
    "documentName": "detailed-flow",
    "defines": [ "STARTUP" ]
  }]}
```

http.FDL

```
eternal: client, server
feature "Web Browsing" {

#ifdef STARTUP
    SYN: client -> server
    "SYN+ACK": client <- server
    ACK: client -> server
#endif

   "HTTP Request": client -> server

#ifdef SUCCESSFUL
   "200 OK": client <- server
#else
   "404 Not Found": client <-
server
#endif
}
```

The generated diagrams and the associated scenarios show the rendering of scenario specific and document specific defines.

Scenario	summary-flow.pdf	detailed-flow.pdf
"200 OK"		 client server 1:SYN 2:SYN+ACK 3:ACK 4:HTTP Request 5:200 OK
"404 Not found"		 client server 1:SYN 2:SYN+ACK 3:ACK 4:HTTP Request 5:404 Not Found

5.3 Messages

5.3.1 Message statement

This statement is used to represent message exchanges between the source and destination. The source and destination can be any of the following:

- Component (i.e. component without any other objects inside it)
- Eternal object
- Dynamic object
- Left Environment (env_l)
- Right Environment (env_r)

The message-statement also allows definition of message parameters of two forms:

- Message field
- Message field with value

The message name may be in identifier format (CallRequest) or string format ("Call Request"). The message field and value can be in identifier or string format.

Messages will be represented as arrows from source to destination. In Sequence Diagrams, message name is printed on the top of the arrow.

5.3.1.1 ADVANCED USES

- **Bidirectional message** interactions can be succinctly represented using the bidirectional form of the message statement. Bidirectional messages are represented with double headed arrows.
- **Important messages** may be represented using weighted arrows. These arrows are drawn with a thicker arrow.
- **Self-messages** are specified by using the same entity as source and destination. “Self-messages” are represented with arrows that return to the originating axis.
- **Message chains** can be used to depict a sequence of message interactions.
- **Lost messages** are represented with an X next to a partial message arrow.

5.3.1.2 EXAMPLES

`Message1: src_proc -> dest_proc`

Message1 is being sent from src_proc to dest_proc component

```
"SYN": client -> server
"SYN+ACK": client <- server
"ACK": client -> server
```

The message sequence shown above demonstrates the use of the reverse arrow.

`Origination(subscriber_id, mode=NORMAL): v5_mgr -> dtmf_proc`
Origination message with parameters subscriber_id and mode is sent from v5_mgr object to dtmf_proc component. Here the mode variable has been given a value of NORMAL.

`Origination(subscriber_id, mode=NORMAL): dtmf_proc <- v5_mgr`
This example uses a reverse arrow.

```
Setup(call_id,
      phone_num,
      priority=HIGH,
      latency=LOW): v5_mgr -> env_r
```

Here is an example of a message statement with parameters spread over multiple lines. The Setup message is being sent from the v5_mgr to the right environment. The following parameters are being sent in the message:

- call_id
- phone_num
- priority with a value of HIGH
- latency with a value of LOW

`"Interview Employee": "Human Resources" -> Department`
A string message name and source have been used in this example.

`HTTP_Get(url="https://www.EventHelix.com"): WebBrowser -> Internet`
A string parameter value is used in this example to represent the URL value.

`"Purchase Order" ("Order Number" = "01-033-56"): User -> Server`
In this example, the message name, parameter type and value are all strings.

`VoicePath: Caller <-> Called`
A bidirectional voice path has been established between the caller and the called subscribers.

`VoicePath: Caller <=> Called`
A bidirectional voice path has been established between the caller and the called subscribers. This time represented with a thick arrow.

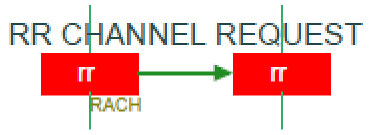
`"RTP Stream": Server => Client`
Important message interaction shown with a weighted arrow.

`"End of Dialing": CallHandler -> CallHandler`
The call handler schedules an internal “End of Dialing” message.

`[sip_style] "Invite": SIP_Phone -> SIP_Proxy`
A style, titled sip_style, is assigned to the "Invite" message. The message is formatted per the style specification.

`PathSetup: src -> dst <http://www.messagedef.com/PathSetup.html>`
A message with a hyperlink specification is shown here. Clicking on the message name will open a browser window for the specified URL.

`"RR CHANNEL REQUEST": Mobile-rr -> BSS-rr`
A message shown here with source and destination tagging. The tags can be used to denote protocol stack layers exchanging the message. The representation of the tags is shown below:



```
chain {
   "Call Setup": UT -> BTS
   "Call Setup": BTS -> BSC
   "Initial Call Message": BSC -> MSC
}
```

A chain statement enclosing messages is shown above. The chain messages are drawn in a single line.

`TCPSegment: client ->X server`
A TCP segment sent from the Client to the Server has been lost

`CourierPackage: Courier =>X Customer`
A courier has lost a package (shown with a bold arrow).

`TCPSegment (parameter1, parameter2):  server  X<- client`
This example uses the reverse lost message arrow.

`CourierPackage(a_parameter, b_parameter):  Customer  X<= Courier`
This example depicts bold reverse lost message arrow.

5.3.2 Chain statement

The Chain statement allows you to define a message chain to model a chain of messages that are triggered in a sequence. EventStudio attempts to draw all messages in a message in a single line.

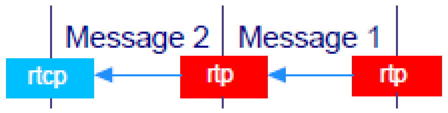
Multiple Message statements may be enclosed between the chain {} blocks.

5.3.2.1 EXAMPLE

```
chain {
    [rstyle] msg: a -> b
    msg2(par1="one", par2): b -> c
    [talk] conversation: c <-> d
}
```

The chain statement shown above will be drawn in a single line if the message arrows do not result in an overlap. If the message arrows result in an overlap, the message arrows are drawn in different lines.

`msg(par1="one", par2): a -> b -> c`
The forward cascaded message statement shown above is a special case of chain statement where message opcode and parameters of chain elements is same and destination of one chain element is the source of next chain element. This type of chain statement will be drawn in a single line if the message arrows do not result in an overlap. If the message arrows result in an overlap, the message arrows are drawn in different lines.



```
chain {
   "Message 1": A-rtp -> B-rtp
   "Message 2": B-rtp -> C-rtcp
}
```

A message chain shown with tags.

More examples:

```
msg(par1="one", par2): a => b => c
msg(par1="one", par2): c <- b <- a
msg(par1="one", par2): c <= b <= b <= a
msg(par1="one", par2): a <-> b <-> c
msg(par1="one", par2): a <=> b <=> c
```

5.3.3 Multicast statement

This statement is used to represent message sent from a given source to more than one destination at the same time. The source and destination can be any of the following:

Component (i.e. component without any other objects inside it)

- Eternal object
- Dynamic object

The multicast-statement also allows definition of message parameters of two forms:

- Message field
- Message field with value

The message name may be in identifier format (CallRequest) or string format ("Call Request"). The message field and value can be in identifier or string format.

A multicast statement can be put only inside the feature {} block in the FDL file.

Messages will be represented as arrows from source to destination. In Sequence Diagrams, message name is printed on the top of the arrow.

5.3.3.1 EXAMPLES

```
src_proc multicasts Message1 to dest_proc1, dest_proc2
```

Message1 is being sent from src_proc to dest_proc1 and dest_proc2 components

```
v5_mgr multicasts Orig(subscriber_id, mode=NORMAL) to dtmf_proc, call_mgr
```

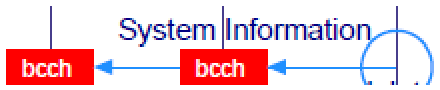
Origination message with parameters subscriber_id and mode is sent from v5_mgr object to dtmf_proc component and call_mgr object. Here the mode variable has been given a value of NORMAL.

```
WebTaxiServer multicasts "Taxi Request" to Taxi1, Taxi2
```

A string message name has been used in this example.

```
User multicasts "Purchase Order" ("01-033-56") to Server1, Server2
```

In this example, the message name, parameter type and value are all strings.



```
BaseStation multicasts "System Information" to UE1-bcch, UE2-bcch
```

Tags are used to identify the BCCH cannel in the multicast.

5.4 Object Interactions

5.4.1 Invokes and Returns statements

The "Invokes" statement is used to model a method invocation on an object. The return from a method is modeled by the "returns" statement. You can specify parameters for the "invokes" as well as "returns" statements. The parameter specification syntax is like the message parameter syntax.

Invokes and returns statements may be nested.

The method invocation is represented with an arrow like the message statement. Once a method is invoked, the axis is represented as a thin rectangular bar that lasts until method return.

The rectangular bar is not shown in the system interaction, subsystem interaction, module interaction and component interaction diagrams.

5.4.1.1 EXAMPLES

```
CallManager invokes Call.HandleInvite(msg="My Invite", "bool send")
  Call action "Verify the Invite Message"
  Call invokes MessageHandler.Send(SIP_OK)
    MessageHandler.Send returns (true)
  Call.HandleInvite returns
```

A nested method invocation along with returns statements is shown above.

```
Call invokes Call.ProcessMessage(msg="My Invite", "bool send")
  Call.ProcessMessage returns
```

An object invokes its own method.

5.5 Object Creation and Deletion

5.5.1 Create statement

This statement is used to create a dynamic object. The object that creates the dynamic object may or may not be specified. A dynamic object create statement can be put only inside the feature {} block in the FDL file. Use a dynamic declaration statement to declare a dynamic object. Use the delete statement when a dynamic object needs to be deleted. A dynamic object created in a feature must be deleted before the feature ends.

The output corresponding to a create statement is shown as a dotted arrow from the creator entity to the created entity. Create is printed on the top of the arrow. And, the instance axis for the dynamic object starts just from the create arrow.

5.5.1.1 EXAMPLES

```
dynamic: call in frontend
feature "Call Setup" {
      msg1: phone -> call_mgr
      call_mgr creates call
      dialtone: call -> phone
      ...
      call_over: call -> call_mgr
      call_mgr deletes call
}
```

Here, a dynamic object call is declared inside the frontend component. The Call Setup feature shows that the call_mgr object creates call object. Once the call is over, call_mgr deletes the call object

```
dynamic: call in frontend
feature "Call Setup" {
      msg1: phone -> call_mgr
      call_mgr creates(call_id, phone_no) call
      dialtone: call -> phone
      ...
      call_over: call -> call_mgr
      call_mgr deletes call
}
```

A create statement with parameters is shown here. The output shows the parameter list below the create opcode in the sequence diagram, component interaction diagram, module interaction diagram, subsystem interaction diagram or system interaction diagram.

```
dynamic: HSS | MRFC in "IMS CN"
feature "IMS Call Setup" {
   create HSS
      "I-CSCF", HSS action "Identify S-CSCF"
   delete HSS

   create MRFC
      "S-CSCF", MRFC action "Setup Conference"
   delete MRFC
}
```

Anonymous create and delete are used here to share a single axis between HSS and MRFC.

Note: When a lightweight header is used, the create box for the dynamic object in the sequence diagram is drawn only when the dynamic object is created. If image based light weight header is selected, the dynamic object image is drawn only when the dynamic object is created. This applies to regular dynamic and reusable dynamic objects.

5.5.2 Delete statement

This statement is used to delete a dynamic object. The object that deletes the dynamic object may or may not be specified. The delete statement also allows a self-delete. A dynamic object delete statement can be put only inside the feature {} block in the FDL file. Use a dynamic declaration statement to declare a dynamic object. Use the create statement when a dynamic object needs to be created. A dynamic object created in a feature must be deleted before the feature ends.

The output corresponding to a delete statement is shown as a big X mark. An arrow from the deleting object to the deleted object precedes the X mark if it is not the case of self-delete.

5.5.2.1 EXAMPLES

```
call_over: call -> call_mgr
call_mgr deletes call
```

Here, on receipt of the call_over message from call object, the call_mgr object deletes the call object.

```
release_call: rout_mgr -> call
call deletes call
```

Here, on receipt of the release_call message from the rout_mgr, the call object deletes itself.

```
dynamic: HSS | MRFC in "IMS CN"
feature "IMS Call Setup" {
   create HSS
      "I-CSCF", HSS action "Identify S-CSCF"
   delete HSS

   create MRFC
      "S-CSCF", MRFC action "Setup Conference"
   delete MRFC
}
```

Anonymous create and delete are used here to share a single axis between HSS and MRFC.

5.6 Timers

5.6.1 Timer Start (One Shot and Periodic) statement

This statement is used for starting a timer by an eternal or dynamic object. The timer name and the object starting the timer must be specified.

Use the timer stop statement for stopping a timer. Use timer timeout statement for showing the timeout for a timer. EventStudio issues a warning if it finds that a timer has been left running at feature end. For dynamic objects, EventStudio issues a warning if timers are left running at object deletion.

EventStudio supports one time and periodic timers. Only one timeout is permitted for "one time" timers. Periodic timers are allowed multiple timeouts.

5.6.1.1 EXAMPLES

```
feature "Call Setup" {
      offhook: phone -> call_mgr
      call_mgr creates call
      call starts await_first_digit_timer
      digits: phone -> call
      call stops await_first_digit_timer
      ...
}
```

Here, the call object starts the await_first_digit_timer (one time) timer to await first digit dialed by the subscriber.

```
feature "Call Setup" {
      offhook: phone -> call_mgr
      call_mgr creates call
      call starts periodic send_billing_pulse_timer
      ...
      timeout send_billing_pulse_timer
      ...
      timeout send_billing_pulse_timer
      ...
      timeout send_billing_pulse_timer
}
```

The call object starts the send_billing_pulse_timer periodic timer. Multiple timeouts are depicted for the timer.

5.6.2 Timer Restart statement

This statement is used for restarting an already existing timer started by any eternal or dynamic component. The timer name and the object stopping the timer must be specified. A timer restart statement can be put only inside the feature {} block in the FDL file.

5.6.2.1 EXAMPLE

```
feature "Call Setup" {
      offhook: phone -> call_mgr
      call_mgr creates call
      call starts await_digit_timer
      digits: phone -> call
      call restarts await_digit_timer
      ...
      timeout await_digit_timer
}
```

Here, the await_digit_timer has been restarted on receipt of digits from the phone.

5.6.3 Timer Stop statement

This statement is used for stopping an existing timer started by an eternal or dynamic object. EventStudio issues a warning if it finds that a timer has been left running at feature end. For dynamic objects, EventStudio issues a warning if timers are left running at object deletion.

5.6.3.1 EXAMPLES

```
feature "Call Setup" {
      offhook: phone -> call_mgr
      call_mgr creates call
      call starts await_first_digit_timer
      digits: phone -> call
      call stops await_first_digit_timer
      ...
}
```

The call object stops the await_first_digit_timer timer on receipt of the first digit dialed by the subscriber.

5.6.4 Timeout statement

This statement is used to show the timeout of an existing timer started by an eternal or dynamic object. Use the timer start statement for starting a timer.

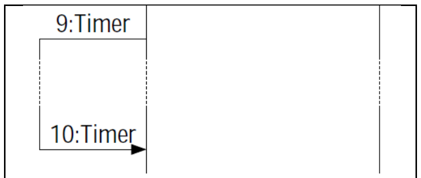
5.6.4.1 EXAMPLES

```
feature "Call Setup" {
      offhook: phone -> call_mgr
      call_mgr creates call
      call starts await_first_digit_timer
      |* The calling subscriber does not dial any digit *|
      timeout await_first_digit_timer
      ...
}
```

Here, the calling subscriber does not dial any digit and the await_first_digit_timer timeout happens.

5.6.5 Time Passage statement

Passage of time can be depicted with three dots in the FDL. This is shown as dotted axis.



```
A starts Timer
...
timeout Timer
```

5.7 Actions

5.7.1 Action statement

This statement is used to model a specific action taken by a single or a group of entities.

The statement takes two forms:

- **Single:** Action involving a single object, component, module, subsystem or system is specified.
- **Multiple:** Action involving multiple objects, components, modules, subsystems or systems is specified.

The output of an action statement is shown as a box with the action indicated inside it. The action statement font, color and line width may be changed by prefixing it with a style statement.

The "action" statement also supports hyperlinks in PDF based sequence diagrams. The hyperlinks may be used to link to files on the Internet or your local drive.

5.7.1.1 EXAMPLES

```
call_mgr action "Obtain subscriber profile"
```

The call_mgr object is shown to perform the action of obtaining the subscriber profile.

```
audit action "Check data consistency"
```

An audit object is shown to perform the action of checking data consistency.

```
[act_style] client, server action "Establish TCP Connection"
```

Client and the server take a joint action “Establish TCP Connection”. The statement also associates a style with the action statement. The style specifies the color, font, line width etc. for this action.

```
bank, broker, customer action "Close the Loan"
```

The bank, the broker and the customer work together to close the loan.

```
a, b action "SIP Call" <http://rfc.net/rfc3261.html>
```

Here the action box is also a hyper link pointing to the SIP RFC.

```
bsc1, bsc2 action "Handover" <file://c:\Doc\handover_diagram.pdf>
```

The above statement links to a sequence diagram describing the handover in detail.

```
system: Earth
subsystem: "North America" in Earth, Asia in Earth
module: "United States" in "North America", Canada in "North America"
module: India in Asia
component: California in "United States", Nevada in "United States"
component: Quebec in Canada, Rajasthan in India
eternal: "Los Angeles" in California, "San Francisco" in California
eternal: "Las Vegas" in Nevada
eternal: Montreal in Quebec, Jaipur in Rajasthan

feature "Bid for Events" {
    /* Component level action:
       - Represented by a box covering Los Angeles, San Francisco and
         Las Vegas axes.
       - Included in component and lower level interaction diagram. */
    California, Nevada action "West Coast Bid for Olympics"

    /* System level action:
       - Represented by a box covering all objects under Earth.
       - Included in system level and lower interaction diagrams. */
    Earth action "Earth Summit"
}
```

5.7.2 Action Begin statement

This statement is used to model an action started by a single entity or a group of entities.

The statement takes the following forms:

- **Single:** Action involving a single object, component, module, subsystem or system is specified.
- **Multiple:** Action involving multiple objects, components, modules, subsystems or systems is specified.

The action statement font, color and line width may be changed by prefixing it with a style statement.

The "action begin" statement also supports hyperlinks in PDF sequence diagrams. The hyperlinks may be used to link to files on the Internet or your local drive.

5.7.2.1 EXAMPLES

```
call begins action "Feeding Ring back tone"
...
call ends action "Feeding Ring back tone"
```

Here, the call object is shown to initiate the action of feeding ring back tone. Later, the call object ends feeding the ring back tone.

```
[my_style] client, server begin action "Security Association"
...
[my_style] client, server end action "Security Association"
```

The client and server objects jointly begin the action of setting up a "Security Association". Later, the client and the server objects jointly end the "Security Association". The statement also associates a style with the action statement. The style specifies the color, font, line width etc. for this action.

5.7.3 Action End statement

This statement is used to model an action completed by a single entity or a group of entities.

The statement takes the following forms:

- **Single:** Action involving a single object, component, module, subsystem or system is specified.
- **Multiple:** Action involving multiple objects, components, modules, subsystems or systems is specified.

The action is indicated inside the box. The action statement font, color and line width may be changed by prefixing it with a style statement.

The "action end" statement also supports hyperlinks in PDF sequence diagrams. The hyperlinks may be used to link to files on the Internet or your local drive.

5.7.3.1 EXAMPLES

```
call begins action "Feeding Ring back tone"
...
call ends action "Feeding Ring back tone" <http://c:\dialtone.pdf>
```

The call object is shown to initiate the action of feeding ring back tone. Later, the call object ends feeding the ring back tone. The "ends action" statement is hyperlinked to a file on the local drive.

```
[my_style] client, server begin action "Security Association"
...
[my_style] client, server end action "Security Association"
```

The client and server objects jointly begin the action of setting up a "Security Association". Later, the client and the server objects jointly end the "Security Association". The statement also associates a style with the action statement. The style specifies the color, font, line width etc. for this action.

5.8 Resource Allocations

5.8.1 Resource Allocate statement

This statement is used for allocating a resource by a single or a group of entities. The entities allocating the resource and the resource name must be specified. Use a resource free statement for deallocating a resource. EventStudio issues a warning when it finds that resources have not been deallocated at feature end. For dynamic objects, EventStudio issues a warning if the resources allocated by the object have not been freed before the object deletion.

The output of a resource allocate statement is shown as a box with "allocate" as the title and the resource name inside it.

The statement takes two forms:

- **Single:** Resource allocation involving a single object, component, module, subsystem or system is specified.
- **Multiple:** Resource allocation involving multiple objects, components, modules, subsystems or systems is specified.

5.8.1.1 EXAMPLES

```
call_mgr allocates "Outgoing trunk"
```

The call manager object allocates an outgoing trunk resource for a call.

```
call_mgr allocates "Outgoing trunk" <http://c:\referencedoc.pdf>
```

The resource allocate statement also supports hyperlinks in PDF based sequence diagrams. The hyperlinks may be used to link to files on the Internet or your local drive.

```
[my_style] client, server allocate "Encryption Coprocessor"
...
[my_style] client, server free "Encryption Coprocessor"
```

The client and server objects jointly allocate an "Encryption Coprocessor". Later, the client and the server objects jointly free the "Encryption Coprocessor". The example also shows a style specification that can be prefixed to the resource allocate and free statements. The "action end" statement also supports hyperlinks in PDF sequence diagrams. The hyperlinks may be used to link to files on the Internet or your local drive.

```
system: Earth
subsystem: "North America" in Earth, Asia in Earth
module: "United States" in "North America", Canada in "North America"
module: India in Asia
component: California in "United States", Nevada in "United States"
component: Quebec in Canada, Rajasthan in India
eternal: "Los Angeles" in California, "San Francisco" in California
eternal: "Las Vegas" in Nevada
eternal: Montreal in Quebec, Jaipur in Rajasthan

feature "Bid for Events"{
    /* Component level resource allocate:
      - Represented by a box with allocate as title and resource name inside it.
      The box covers Los Angeles, San Francisco and Las Vegas axes.
       - Included in component and lower level interaction diagram. */
    California, Nevada allocate "Stadium for Olympics in West Coast"

    ...
    California, Nevada free "Stadium for Olympics in West Coast"

    Earth allocates "Earth's Satellite"
    ...
    Earth frees "Earth's Satellite"
}
```

5.8.2 Resource Free statement

This statement is used for de-allocating a resource by a single or a group of entities. The resource must have been allocated earlier by the resource allocate statement. The entities deallocating the resource and the resource name must be specified in the statement. EventStudio issues a warning when it finds that resources have not been deallocated at feature end. For dynamic objects, EventStudio issues a warning if the resources allocated by the object have not been freed before the object deletion.

Like resource allocate statement, resource free statement also supports user definable hyperlinks.

The statement takes two forms:

- **Single:** Resource free involving a single object, component, module, subsystem or system is specified.
- **Multiple:** Resource free involving multiple objects, components, modules, subsystems or systems is specified.

5.8.2.1 EXAMPLES

```
[my_style] call_mgr frees "Outgoing trunk"
```

The call manager object frees the outgoing trunk resource allocated earlier for a call. The example also shows a style specification that can be prefixed to the resource allocate and free statements.

```
client, server allocate "Encryption Coprocessor"
...
client, server free "Encryption Coprocessor"
```

The client and server objects jointly allocate an "Encryption Coprocessor". Later, the client and the server objects jointly free the "Encryption Coprocessor".

```
system: Earth
subsystem: "North America" in Earth, Asia in Earth
module: "United States" in "North America", Canada in "North America"
module: India in Asia
component: California in "United States", Nevada in "United States"
component: Quebec in Canada, Rajasthan in India
eternal: "Los Angeles" in California, "San Francisco" in California
eternal: "Las Vegas" in Nevada
eternal: Montreal in Quebec, Jaipur in Rajasthan

feature "Bid for Events" {
    California, Nevada allocate "Stadium for Olympics in West Coast"

    ...
    California, Nevada free "Stadium for Olympics in West Coast"

    Earth allocates "Earth's Satellite"
    ...
    /* System level resource free:
       - Represented by a box with title free and resource name inside it
covering all objects under Earth.
       - Included in system level and lower interaction diagrams. */
    Earth frees "Earth's Satellite"
}
```

5.9 State Transitions

5.9.1 State Change statement

This statement is used to represent state transitions of state machines. State for a single or a group of entities may be specified. The name of the entities and the name of the state to which transition is taking place, must be specified. A state change statement can be put only inside the feature {} block in the FDL file. This statement is used during detailed design phase.

The output of a state change statement is shown as a hexagonal box with the name of state indicated inside it.

Note: The strings used in this statement should not contain any leading or trailing blanks. Also, words within the string should not be separated by more than one blank.

The statement takes two forms:

- **Single:** State change involving a single object, component, module, subsystem or system is specified.

- **Multiple:** State change involving multiple objects, components, modules, subsystems or systems is specified.

5.9.1.1 EXAMPLES

`call state = "Collecting digits"`
The state of the call object state machine has been specified to change to collecting digits.

`caller, called state = "Conversation"`
The caller and called subscribers have jointly entered the "Conversation" state.

`[my_style] "car dealership" state = "Awaiting Shipment"`
Here a style prefix is used to control the layout of a state statement.

`[my_style] customer, banker state = "Loan Closed"`
Here a style prefix is used to control the layout of a joint state statement.

`California, Nevada state = "Developed"`
Here, the state of the California and Nevada state machines has been specified to change to developed.

5.10 Sequence Groupings

5.10.1 Sequence Block Statement

A sequence block is used to group multiple statements. A sequence block encloses multiple statements between the sequence {}. Sequences are identified by their title. The title is also displayed in the PDF bookmarks. Sequence blocks may be nested.

A sequence block can be displayed in two modes, expanded or summarized. A sequence block is represented as a rectangle around the enclosed statements in the expanded mode. Nested sequences are represented with nested rectangles. In the summarized mode, the sequence block is represented as a rectangle with just the title; no interactions within the sequence block are shown.

The real value of the sequence statement is that it lets you control the level of detail in the document by choosing how to display a sequence. EventStudio supports the following options:

For Sequence expansion option...	...add to document object in project.scn.json
Don't expand	`"expandSequence": "do-not-expand",`
Expand first time only	`"expandSequence": "expand-first-time-only"`
Always expand	`"expandSequence": "expand-always"`

When configured in the “expand-first-time-only”, a sequence block is automatically minimized if the sequence block has already been included in a previous scenario in the generated diagram. This feature helps minimize the size of a sequence diagram when many scenarios are included in a single diagram.

You may use this statement to define:

- A while loop that contains a sequence of statements that are repeated to accomplish a task.
- Additional level of detail that may be abstracted out in a high-level diagram.

5.10.1.1 EXAMPLES

```
sequence "PDP Context Activation" {
    "PDP Context Activation Request" : UT -> "Core Network"
    "Core Network" action "Establish Quality of Service"
    "PDP Context Activation Response": UT <- "Core Network"
}
```

The PDP context activation is modeled as a sequence. When expanded, the statements in this sequence block are rendered inside a rectangle that covers the entities participating in the sequence.

```
#define GET_DIGIT(_call, _dsp) sequence {"Get Digit";\
    "Request Digit"  : _call -> _dsp;\
    "Collected Digit": _call <- _dsp;\
    "More Digits"    : _call -> _dsp;\
    |* _call needs more digits. *|;\
    }
```

Sequence blocks and macros make a good fit. The GET_DIGIT macro contains the "Get Digit" sequence block.

```
sequence "Update Database" {
    sequence "Connect to Database"{
        "Courier Agent" action "Locate database"
        "Courier Agent" action "Connect to database"
    }
    "Courier Agent" action "Query the database"
    "Courier Agent" action "Update the database"
    "Backup Entry": "Courier Agent" -> "Backup Database Server"
}
```

Sequences can be nested as shown here. EventStudio automatically determines the bounding rectangle for each sequence.

5.10.2 Preconditions Block

Preconditions in use case may be enclosed in a preconditions block. Any statement permitted in a sequence block may be used in a preconditions block.

```
preconditions {
   "Courier Agent" state = "Awaiting User Input"
   "Backup Database Server" action "Initialize"
}
```

5.10.3 Postconditions Block

Post-conditions in use case may be enclosed in a postconditions block. Any statement permitted in a sequence block may be used in a postconditions block.

```
postconditions {
   "Courier Agent" state = "Completed user transaction"
   "Backup Database Server" action "Commit Transaction"
}
```

5.10.4 Loop Block

Looping may be modeled using the loop block. Any statement permitted in a sequence block may be used in a loop block. Loop blocks may be nested.

```
loop "Iterate over all rows" {
   loop "Iterate over all columns" {
      GameController invokes ChessBoard.Reset(row, column)
  }
}
```

5.11 Remarks

5.11.1 Remark statement

This statement allows the user to explain the feature flow. Single line and multi-line remarks are supported:

- Single line remarks begin with % and end
- Multi-line remarks are enclosed within **|*** and ***|**.

The remarks are displayed on the right-side remark column of a sequence diagram. FDL associates remarks with the previous FDL statement.

5.11.1.1 EXAMPLES

```
routing_request(digits): call_mgr -> rout_mgr

% Call manager sends the routing request to routing manager.

routing_response(destination_equipment_num): rout_mgr -> call_mgr
```

```
routing_request(digits): call_mgr -> rout_mgr

|* Call manager sends the routing request to routing manager.
This message carries the called party digits *|

routing_response(destination_equipment_num): rout_mgr -> call_mgr
```

5.11.2 Block Remark statement

This statement allows the user to explain the feature flow. FDL supports block remarks enclosed within |= and =|. FDL supports multi-line block remark statement. Block remarks are used to show milestones in the execution of a feature. They do not associate with any FDL statement. It should be noted that a block remark statement could be put only inside feature {} block in the FDL file.

Block remarks are printed along the complete width of the PDF output page. Block remarks support the following formatting options:

- The text can be divided between paragraphs by simply leaving a blank line between the two paragraphs.
- Block remark text alignment can be changed in theme declaration (5.13.4). Supported alignment options are:
  - Left Align
  - Center Align
  - Right Align
  - Justify
- Verbatim (Line breaks are preserved)

5.11.2.1 EXAMPLES

```
routing_response: rout_mgr -> call_mgr
ringing: call_mgr -> trm_call

|= Call routing complete, terminating subscriber being rung, answer
awaited =|
```

Here, the block remark statement is not associated with any message statement. It indicates the milestone that the call is in ringing phase.

```
|= This scenario describes call setup when the call is originated
by a mobile subscriber. The document describes a detailed flow
involving the base station, MSC, HLR and VLR.

Note that this document corresponds to Release 05.00.77. Changes
are needed to support the latest release. =|

channel_setup: SubscriberTerminal -> BaseStation
AGCH: BaseStation -> SubscriberTerminal
```

This block remark statement has been used to introduce the scenarios. Note how the text has been divided into two paragraphs. The output scenario diagrams will preserve the division into paragraphs.

5.11.3 Comment statement

This statement is used to provide programmer documentation. FDL supports standard C-type comments enclosed within /* and */. FDL supports multi-line comment statement.

EventStudio ignores the standard C-type comments and no output is produced corresponding to the comment.

5.11.3.1 EXAMPLES

```
/* V5.2 originating
   call setup */

feature "Call Setup" {
      msg1: phone -> call_mgr
      msg2: call_mgr -> core
}
```

C-style multi-line comments are supported.

```
// V5.2 originating call setup

feature "Call Setup" {
      msg1: phone -> call_mgr
      msg2: call_mgr -> core
}
```

C++ style single line comments are supported. Note that these comments should not be used in macro definitions.

5.11.4 Heading statement

A large sequence diagram may be subdivided into sections using the heading statement.

- The headings are displayed as a block remark.
- A bookmark to the headings is added in the left pane of a PDF sequence diagram file.
- Message numbering is reset when headings are encountered.

5.11.4.1 EXAMPLES

The syntax for headings is shown in the following example:

```
heading "Conversation Mode"
```

5.11.5 Issue statement

This statement is used to indicate unresolved issues that may exist. Another use may be to provide review comments. An issue statement can be put only inside the feature {} block in the FDL file.

The output corresponding to an issue statement is shown in bold in the right-side "remarks" column. Also, whenever a review document command is given, all the issue statements in the FDL file are reported in the output window.

```
5.11.5.1    EXAMPLE
issue "Call metering procedure to be defined"
```

5.12 Preprocessor

FDL supports a powerful preprocessor that is like the C preprocessor. All preprocessor directives begin with a hash (#). The following preprocessor directives are supported:

```
#include "stdinc.fdl"
```

Include an FDL file located in the current directory. Paths relative to the current directory may be used.

```
#include <theme.fdl>
```

Include an FDL file from the include path list. They include path list contains:

- Standard include directory in the EventStudio extension folder.
- Project include directory in the project directory.

```
#define msg_ack(msg, a, b)   "msg": a -> b;\
                             |* msg sent from a to b *|;\
                             "msg ack": b -> a;\
                             |* b acknowledges msg *|;
```

- Macros may be defined with multiple parameters.
- Multiple statements may be defined in a single line by separating them with a semicolon (;)
- Macros may be split into multiple lines by using a semicolon-backslash-enter (;\) pattern.
- Macro parameter substitution is allowed in macro text, strings, remarks and block remarks.
- Macros may be nested.

```
#define tone_feed(d, t, a)   |= Feed d#t =|;\
                             a action "Feeding d t";
```

Token pasting operator hash (#) allows two macro parameters to be "pasted" together.

```
#ifdef LTE_RELEASE_10
#else
#endif
```

Conditional compilation is supported with the #ifdef-#else-#endif statement. Symbols controlling the conditional compilation may be defined

using a: • #define, or • Symbols defined in the scenario or documents (Chapter 3).

`#ifdef LTE_RELEASE_10`
`#endif`
 	
Conditional compilation is supported with the #ifdef--#endif statement. Symbols controlling the conditional compilation may be defined using a:
 	
• #define, or
• Symbols defined in the scenario or documents (Chapter 3).

5.13 Styles, Themes and Document Layout

5.13.1 Style declaration

Style specifications allow you to specify the formatting for message statements. The style declaration allows you to specify:

- The color as red, blue and green components.
- Font and font size for the message name text
- Font and font size for the parameter text
- Line pattern for the message arrow
- Line width for the message arrow

The ranges for individual parameters are specified below. Note that not all values need to be specified in a style declaration. Message defaults are used for unspecified values.

`color`	Specifies the color to be used. The color is specified as a set of three floating point numbers between 0.0 and 1.0. The first one denotes the red component, the second specifies the green component and the last one specifies the blue component. • Millions of colors can be defined in terms of the individual color component values. **Color** appendix may be used to select from predefined colors.
`bgcolor`	Specifies the background color. The bgcolor attribute is supported in: • Axis boxes in light weight header • Begin/end action (single and joint) • Allocate/free resource (single and joint) • Action (single and joint)

`textcolor`	Specifies the text color in all statements. Color appendix may be used to select from predefined colors.
`paramcolor`	Specifies the parameter color in message and object interaction statements. Color appendix may be used to select from predefined colors.
`font`	Specifies the font of the message name. The following font strings are supported. • `"Arial"` • `"Arial Narrow"` • `"Courier New"` • `"Times New Roman"` • `"Arial-Bold"` • `"Arial Narrow-Bold"` • `"Courier New-Bold"` • `"Times New Roman-Bold"`
`fontsize`	Specifies the size of the message name font in points.
`linewidth`	Specifies the width of the message arrow in points.
`linepattern`	Selects the line pattern for the message arrow. The options are: • `"---"` (Normal line) • `"- -"` (Dashed line) • `"..."` (Dotted line)
`paramfont`	Specifies the font for message parameters. The font strings are same as the message name font specification.
`paramfontsize`	Specifies the size of the message parameter font in points.

`image`	Windows only Specifies a JPEG image associated with the style. The image specification is used when styles are applied to object, component and module declarations. The image specified in the style is used in the column heading if `sequenceDiagramHeader` attribute in the active theme is set to `"single-tier-light-weight-with-images"`. The image selection in the header is based on the following rules: | Eternal/Dynamic | Component | Module Images | Images Drawn in |

Object Images	**Images**		**the Header**
At least one eternal or dynamic object uses a style with image.	N/A	N/A	The eternal/dynamic objects that have associated images are drawn with the image.
No eternal or dynamic object has a style with an image.	At least one parent component has a style with an image.	N/A	Component image is used. The image is centered between all eternal and dynamic objects within the component.
No eternal or dynamic object has a style with an image.	No component has a style with an image.	At least one module has a style with an image.	Module level images are used. The images are centered between all entities contained in a module.

`match`	Specifies a regular expression that is used to automatically apply a style to statements that match the regular expression. When a match attribute is present in a style, the style gets automatically gets applied to statements that match the specified regular expression. For details about regular expressions, refer to the appendix on **Regular Expression**

5.13.1.1 EXAMPLE

```
style sip: color="1.0,0.0,0.0", font="Times New Roman-Bold", fontsize="15",
   linewidth="4", linepattern="- -", paramfont="Courier New", paramfontsize="4"
style server: color="0.0,1.0,0.0", font="Times New Roman-Bold", image="server.jpg"
style mobile: color="0.0,0.0,1.0", font="Times New Roman-Bold", image="mobile.jpg"

module: m
component: p in m
eternal: [mobile] e in p, [server] f in p
style udp_tag_style: color=RED, bgcolor=ROYALBLUE, font="Arial Narrow"

feature "Testing" {
    [sip] invite(caller, called): e-udp -> f-udp
    two_way_path: e <-> f
}
```

In the above example, we have explicitly applied the style that we want to apply to a statement. EventStudio supports two ways to implicitly specify the styles:

- Default Styles
- Regular Expression Style Filters

The above example also illustrates the styling as applied to

Message Tagging. EventStudio applies the udp_tag_style to the definition in the invite message.

5.13.2 Default Styles

Specifying a style with every statement can become tedious. EventStudio allows you to specify the default style to be used when no style has been specified.

Simply define styles with the following names. EventStudio will automatically apply these styles to the appropriate statements and declarations. (Note that an explicit style specification will override the default style specification.)

Default Style Name	Application
`default_invokes_style`	Default style for invokes statements
`default_returns_style`	Default style of a return from an invokes-statement.
`default_message_style`	Default style for unidirectional and bidirectional message interactions.
`default_cascaded_message_style`	Default style for cascaded messages. If this style is specified, it applies to statements of the form: • `Msg: a -> b -> c` • `Msg: a <-b <- c`
`default_tag_style`	Default style for message tags.
`default_multicast_style`	Default style for multicast.
`default_create_style`	Default style for object creation statement.
`default_delete_style`	Default style for object deletion statement.
`default_remark_style`	Default style for remarks.
`default_block_remark_style`	Default style for block remarks.

`default_timer_style`	Default style for timer operations.
`default_state_style`	Default style for state transitions.
`default_action_style`	Default style for: • action • Begin action • End action
`default_allocate_style`	Default style for resource allocation statement.
`default_free_style`	Default style for resource free statement.
`default_system_style`	Default axis style used in system level interactions diagrams.
`default_subsystem_style`	Default axis style used in subsystem level interactions diagrams.
`default_module_style`	Default axis style used in module level interactions diagrams.
`default_component_style`	Default axis style used in component level interactions diagrams.
`default_eternal_style`	Default axis style used for eternal objects.
`default_dynamic_style`	Default axis style used for dynamic objects.
`default_heading_style`	Default style used in the heading statement.

5.13.2.1 EXAMPLE

```
style default_message_style: color="1.0,0.0,0.0", font="Times New Roman-Bold",
   fontsize="15", linewidth="4", linepattern="- -", paramfont="Courier New",
   paramfontsize="4"

module: m
component: p in m
eternal: e in p, f in p

feature "Testing" {
    /* default_message_style is applied to the following message statements. */
    invite(caller, called): e -> f
    two_way_path: e <-> f
}
```

5.13.3 Regular Expression Style Filters

You can use the match attribute in a style declaration to specify a regular expression. All statements that match the regular expression will automatically be assigned the style defining the match attribute. (For more information about regular expression matching, refer to the Regular Expression appendix).

5.13.3.1 EXAMPLE

```
/* request_style is automatically applied to messages that end with the "Request". */
style request_style: color="1.0,0.0,0.0", font="Times New Roman-Bold",
   fontsize="15", linewidth="1", paramfont="Courier New",
   paramfontsize="4", match="Request$"

/* response_style is automatically applied to messages that end with the "Response". */
style response_style: color="0.0,1.0,0.0", font="Times New Roman-Bold",
   fontsize="15", linewidth="1", linepattern="- -", paramfont="Courier New",
   paramfontsize="4", match="Response$"

module: m
component: p in m
eternal: e in p, f in p

feature "Testing" {
    /* The request_style is applied to the following statement because the message
       opcode matches the style regular expression (i.e. ends with Request).*/
    ResourceAllocationRequest: e -> f

    /* The response_style is applied to the following statement because the message
       opcode matches the style regular expression. (i.e. ends with Response)*/
    ResourceAllocationResponse: e <- f
}
```

5.13.4 Theme declaration

Themes provide an overall control over the document layout.

- A theme declaration consists of a theme name followed by a sequence of attribute-value pairs. The attribute-value pairs might be specified across multiple lines. The syntax is:

```
theme MyTheme: blockRemarkFormatting="right-align",
               paperSize="letter"
theme LeftAlign: blockRemarkFormatting="left-align"
```

- Initial theme for a statement is specified as a modifier to the feature statement. The syntax is:

```
{MyTheme} feature "a feature" {

}
```

- The theme declaration supports the attribute-value pairs shown in the table below.

Attribute	Possible values
messageParameterFormatting	"multiple-parameters-per-line-center-align" "single-parameter-per-line-center-align" "single-parameter-per-line-left-align"
sequenceDiagramHeader	"multi-tier-header" "single-tier-light-weight" "single-tier-light-weight-with-images"
displayParentInLightWeightHeader	"true" "false"
blockRemarkFormatting	"left-align" "center-align" "right-align"

	"justify" "verbatim"
assignSequenceNumbersInSequenceDiagrams	"true" "false"
remarkType	"callout" - Display remarks as callouts "column" - Display remarks on the right side
paperSize	"letter" "legal" "B" "C" "A4" "A3" "B4" "<width-in-inch> inch <height-in-inch> inch" or "<width-in-mm> mm <height-in-mm> mm" [4]
minimizeEMFImageWidth	"true" "false"
minimizeEMFImageLength	"true" "false"
sequenceDiagramLayout	"auto-choose-between-portrait-landscape" "landscape-only" "portrait-only"
stateShape	"curved-rectangle" "hexagon"
pageMarginPDF	"small" "medium" "large"
pageMarginEMF	"small" "medium" "large"
headingAndTitleFont	"Arial" "Arial Narrow" "Courier New" "Times New Roman"
sequenceDiagramMessageNameFont	"Arial" "Arial Narrow" "Courier New" "Times New Roman"
sequenceDiagramMessageNameFontSize	"1" to "35"
collaborationDiagramMessageNameFont	"Arial" "Arial Narrow" "Courier New" "Times New Roman"
collaborationDiagramMessageNameFontSize	"1" to "35"
messageParameterFont	"Arial" "Arial Narrow" "Courier New" "Times New Roman"
messageParameterFontSize	"1" to "35"
actionBoxFont	"Arial" "Arial Narrow" "Courier New" "Times New Roman"
actionBoxFontSize	"1" to "35"
remarkFont	"Arial" "Arial Narrow" "Courier New" "Times New Roman"
remarkFontSize	"1" to "35"
blockRemarkFont	"Arial" "Arial Narrow" "Courier New" "Times New Roman"
blockRemarkFontSize	"1" to "35"

5.13.5 Page Break statement

This statement inserts a page break in PDF and EMF files. Use this statement whenever you wish to force the PDF or EMF output to start from a new page.

Note: This statement applies to only PDF and EMF files. It is ignored when generating HTML output.

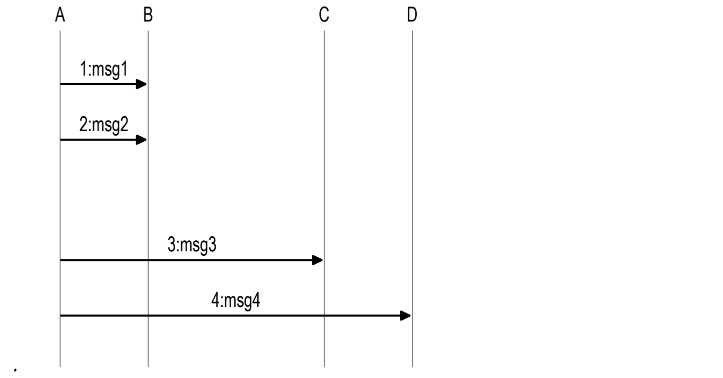
[4] e.g. **`paperSize="5.5 inch 6.5 inch"`** or **`paperSize="60 mm 70 mm"`**.

5.13.5.1 EXAMPLE

```
...
pagebreak
|= Call Release Processing. =|
```

Here, a page break has been forced at the beginning of a new phase of a call.

5.13.6 Horizontal and Vertical Separator statements



```
eternal: A in p, B in p, separator in p, C in p, D in
p
feature "f"
msg1: A -> B
msg2: A -> B
separator
msg3: A -> C
msg4: A -> D
```

You can insert a blank column by declaring a separator. An example is shown in the figure on the left. Space has been left between objects B and C.

A blank line can be inserted by just adding a separator between two statements. Refer to the example on the left.

6 REGULAR EXPRESSION REFERENCE

6.1 Regular Expression Examples

Regular Expression Example	Meaning
Docking	Matches all messages containing "Docking" anywhere in the message, e.g. ReadyFor*Docking*, *Docking*Command
^Docking	Matches all messages beginning with "Docking", e.g. *Docking*Command, *Docking*Rejected
\cundocking	Matches all messages containing "undocking". Case is ignored in the comparison, e.g. *Undocking*Allowed, tawait_*undocking*_finished
Response$	Matches all messages ending with "Response", e.g. OrbitDealloc*Response*, EndDocking*Response*
{Request}|{Response}$	All message ending with "Request" or "Response" e.g. Orbit*Request*, OrbitAlloc*Response*
^.r	Matches all messages with "r" as the second character, e.g. O*r*bitAllocRequest, P*r*oceedForUndocking
^[DU]	Matches messages beginning with D or U. e.g. *D*ockingRequest, *U*ndockingRequest
^[^DU]	Matches messages not beginning with D or U. e.g. *R*eadyForDocking, *O*rbitRequest
<Act	Matches message containing words beginning with Act. e.g. *Act*ivateSession, "Demand *Act*ion"
ion>	Matches messages containing words ending with ion. e.g. "Demand Act*ion*", "Spread *ion* message"
^.[0-9]	Matches messages that have a digit in the second character. e.g. M*1*Request, "Q2 Report"
\s	Matches message that contain white spaces. e.g. "Demand Action", "Invoke Call Request"

^{Setup}|{Release}_([a-z]#)_([0-9]#)$	Matches messages that begin with "Setup" or "Release" and contains lower case word followed by a number. e.g. *Setup_call_53*, *Release_message_23*, *Setup_request_1*
^{Error}(_0x[0-9A-F]#)#$	Matches messages starting with Error and followed by hex numbers. e.g. Error_0xAA_0x01_0x555, Error_0xAB

6.2 Regular Expression Rules

Pattern	Meaning
^	Match the beginning of line
$	Match the end of line
.	Match any character
[ ]	Match characters in set
[^]	Match characters not in set
?	Match previous pattern 0 or 1 times (greedy)
|	Match previous or next pattern
@	Match previous pattern 0 or more times (non-greedy)
#	Match previous pattern 1 or more times (non-greedy)
*	Match previous pattern 0 or more times (greedy)
+	Match previous pattern 1 or more times (greedy)
{ }	Group characters to form one pattern
( )	Group and remember
\	Quote next character (only of not a-z)
<	Match beginning of a word
>	Match end of a word
\x##	Match character with ASCII code ## (hex)

\N###	Match ASCII code ### (dec)
\o###	Match ASCII code
\a	Match \a
\r	Match 0x13 (cr)
\b	Match \b
\t	Match 0x09 (tab)
\f	Match \f
\v	Match \v
\n	Match 0x10 (lf)
\e	Match escape (^E)
\s	Match whitespace (cr/lf/tab/space)
\S	Match nonwhitespace (!\S)
\w	Match word character
\W	Match non-word character
\d	Match digit character
\D	Match non-digit character
\U	Match uppercase
\L	Match lowercase
\C	Match case sensitively from here on
\c	Match case ignore from here on

7 COLOR REFERENCE

BLACK	"0.0,0.0,0.0"	⬛
DIMGRAY	"0.41,0.41,0.41"	⬛
DIMGREY	"0.41,0.41,0.41"	⬛
GRAY	"0.50,0.50,0.50"	⬛
GREY	"0.50,0.50,0.50"	⬛
DARKGREY	"0.66,0.66,0.66"	⬜
DARKGRAY	"0.66,0.66,0.66"	⬜
SILVER	"0.75,0.75,0.75"	⬜
LIGHTGRAY	"0.83,0.83,0.83"	⬜
LIGHTGREY	"0.83,0.83,0.83"	⬜
GAINSBORO	"0.86,0.86,0.86"	⬜
WHITESMOKE	"0.96,0.96,0.96"	⬜
WHITE	"1.00,1.00,1.00"	⬜
ROSYBROWN	"0.74,0.56,0.56"	🟫
INDIANRED	"0.80,0.36,0.36"	🟥
BROWN	"0.65,0.16,0.16"	🟥
FIREBRICK	"0.70,0.13,0.13"	🟥
LIGHTCORAL	"0.94,0.50,0.50"	🟥
MAROON	"0.50,0.0,0.0"	🟥
DARKRED	"0.55,0.0,0.0"	🟥
RED	"1.00,0.0,0.0"	🟥
SNOW	"1.00,0.98,0.98"	⬜
SALMON	"0.98,0.50,0.45"	🟥
MISTYROSE	"1.00,0.89,0.88"	⬜
TOMATO	"1.00,0.39,0.28"	🟥
DARKSALMON	"0.91,0.59,0.48"	🟧
ORANGERED	"1.00,0.27,0.0"	🟧
CORAL	"1.00,0.50,0.31"	🟧
LIGHTSALMON	"1.00,0.63,0.48"	🟧
SIENNA	"0.63,0.32,0.18"	🟫
CHOCOLATE	"0.82,0.41,0.12"	🟫
SADDLEBROWN	"0.55,0.27,0.7"	🟫
SEASHELL	"1.00,0.96,0.93"	⬜

SANDYBROWN	"0.96,0.64,0.38"	🟧
PEACHPUFF	"1.00,0.85,0.73"	🟧
PERU	"0.80,0.52,0.25"	🟫
LINEN	"0.98,0.94,0.90"	⬜
DARKORANGE	"1.00,0.55,0.0"	🟧
BISQUE	"1.00,0.89,0.77"	🟧
TAN	"0.82,0.71,0.55"	🟫
BURLYWOOD	"0.87,0.72,0.53"	🟫
ANTIQUEWHITE	"0.98,0.92,0.84"	⬜
NAVAJOWHITE	"1.00,0.87,0.68"	🟧
BLANCHEDALMOND	"1.00,0.92,0.80"	⬜
PAPAYAWHIP	"1.00,0.94,0.84"	⬜
MOCCASIN	"1.00,0.89,0.71"	🟧
WHEAT	"0.96,0.87,0.70"	🟫
OLDLACE	"0.99,0.96,0.90"	⬜
ORANGE	"1.00,0.65,0.0"	🟧
FLORALWHITE	"1.00,0.98,0.94"	⬜
GOLDENROD	"0.85,0.65,0.13"	🟨
DARKGOLDENROD	"0.72,0.53,0.4"	🟫
CORNSILK	"1.00,0.97,0.86"	⬜
GOLD	"1.00,0.84,0.0"	🟨
KHAKI	"0.94,0.90,0.55"	🟨
LEMONCHIFFON	"1.00,0.98,0.80"	⬜
PALEGOLDENROD	"0.93,0.91,0.67"	🟨
DARKKHAKI	"0.74,0.72,0.42"	🟫
BEIGE	"0.96,0.96,0.86"	⬜
LIGHTGOLDENRODYELLOW	"0.98,0.98,0.82"	⬜
OLIVE	"0.50,0.50,0.0"	🟫
YELLOW	"1.00,1.00,0.0"	🟨
LIGHTYELLOW	"1.00,1.00,0.88"	⬜
IVORY	"1.00,1.00,0.94"	⬜
OLIVEDRAB	"0.42,0.56,0.14"	🟩
YELLOWGREEN	"0.60,0.80,0.20"	🟩
DARKOLIVEGREEN	"0.33,0.42,0.18"	🟩
GREENYELLOW	"0.68,1.00,0.18"	🟩
LAWNGREEN	"0.49,0.99,0.0"	🟩

CHARTREUSE	"0.50,1.00,0.0"	🟩
DARKSEAGREEN	"0.56,0.74,0.56"	🟩
FORESTGREEN	"0.13,0.55,0.13"	🟩
LIMEGREEN	"0.20,0.80,0.20"	🟩
LIGHTGREEN	"0.56,0.93,0.56"	🟩
PALEGREEN	"0.60,0.98,0.60"	🟩
DARKGREEN	"0.0,0.39,0.0"	🟩
GREEN	"0.0,0.50,0.0"	🟩
LIME	"0.0,1.00,0.0"	🟩
HONEYDEW	"0.94,1.00,0.94"	⬜
SEAGREEN	"0.18,0.55,0.34"	🟩
MEDIUMSEAGREEN	"0.24,0.70,0.44"	🟩
SPRINGGREEN	"0.0,1.00,0.50"	🟩
MINTCREAM	"0.96,1.00,0.98"	⬜
MEDIUMSPRINGGREEN	"0.0,0.98,0.60"	🟩
MEDIUMAQUAMARINE	"0.40,0.80,0.67"	🟩
AQUAMARINE	"0.50,1.00,0.83"	🟩
TURQUOISE	"0.25,0.88,0.82"	🟦
LIGHTSEAGREEN	"0.13,0.70,0.67"	🟦
MEDIUMTURQUOISE	"0.28,0.82,0.80"	🟦
DARKSLATEGRAY	"0.18,0.31,0.31"	⬛
DARKSLATEGREY	"0.18,0.31,0.31"	⬛
PALETURQUOISE	"0.69,0.93,0.93"	🟦
TEAL	"0.0,0.50,0.50"	🟦
DARKCYAN	"0.0,0.55,0.55"	🟦
AQUA	"0.0,1.00,1.00"	🟦
CYAN	"0.0,1.00,1.00"	🟦
LIGHTCYAN	"0.88,1.00,1.00"	⬜
AZURE	"0.94,1.00,1.00"	⬜
DARKTURQUOISE	"0.0,0.81,0.82"	🟦
CADETBLUE	"0.37,0.62,0.63"	🟦
POWDERBLUE	"0.69,0.88,0.90"	🟦
LIGHTBLUE	"0.68,0.85,0.90"	🟦
DEEPSKYBLUE	"0.0,0.75,1.00"	🟦
SKYBLUE	"0.53,0.81,0.92"	🟦
LIGHTSKYBLUE	"0.53,0.81,0.98"	🟦

STEELBLUE	"0.27,0.51,0.71"	🟦
ALICEBLUE	"0.94,0.97,1.00"	⬜
SLATEGREY	"0.44,0.50,0.56"	⬛
SLATEGRAY	"0.44,0.50,0.56"	⬛
LIGHTSLATEGREY	"0.47,0.53,0.60"	⬛
LIGHTSLATEGRAY	"0.47,0.53,0.60"	⬛
DODGERBLUE	"0.12,0.56,1.00"	🟦
LIGHTSTEELBLUE	"0.69,0.77,0.87"	🟦
CORNFLOWERBLUE	"0.39,0.58,0.93"	🟦
ROYALBLUE	"0.25,0.41,0.88"	🟦
MIDNIGHTBLUE	"0.10,0.10,0.44"	🟦
LAVENDER	"0.90,0.90,0.98"	⬜
NAVY	"0.0,0.0,0.50"	🟦
DARKBLUE	"0.0,0.0,0.55"	🟦
MEDIUMBLUE	"0.0,0.0,0.80"	🟦
BLUE	"0.0,0.0,1.00"	🟦
GHOSTWHITE	"0.97,0.97,1.00"	⬜
DARKSLATEBLUE	"0.28,0.24,0.55"	🟪
SLATEBLUE	"0.42,0.35,0.80"	🟪
MEDIUMSLATEBLUE	"0.48,0.41,0.93"	🟪
MEDIUMPURPLE	"0.58,0.44,0.86"	🟪
BLUEVIOLET	"0.54,0.17,0.89"	🟪
INDIGO	"0.29,0.0,0.51"	🟪
DARKORCHID	"0.60,0.20,0.80"	🟪
DARKVIOLET	"0.58,0.0,0.83"	🟪
MEDIUMORCHID	"0.73,0.33,0.83"	🟪
THISTLE	"0.85,0.75,0.85"	🟪
PLUM	"0.87,0.63,0.87"	🟪
VIOLET	"0.93,0.51,0.93"	🟪
PURPLE	"0.50,0.0,0.50"	🟪
DARKMAGENTA	"0.55,0.0,0.55"	🟪
FUCHSIA	"1.00,0.0,1.00"	🟪
MAGENTA	"1.00,0.0,1.00"	🟪
ORCHID	"0.85,0.44,0.84"	🟪
MEDIUMVIOLETRED	"0.78,0.08,0.52"	🟥
DEEPPINK	"1.00,0.08,0.58"	🟥

HOTPINK	"1.00,0.41,0.71"	🟥
PALEVIOLETRED	"0.86,0.44,0.58"	🟥
LAVENDERBLUSH	"1.00,0.94,0.96"	⬜
CRIMSON	"0.86,0.08,0.24"	🟥
PINK	"1.00,0.75,0.80"	🟥
LIGHTPINK	"1.00,0.71,0.76"	🟥

8 DEPRECATED

This section documents FDL syntax that has been deprecated. This syntax is supported for backward compatibility.

8.1 Case statement

This statement is used to specify multiple scenarios in the same FDL file. A case statement is enclosed within a case-endcase block. The legs for the case are included within the case-endcase block. Use the leg statement to define each scenario option. Nesting of case statements is also supported. A scenario determines the flow of a feature under certain conditions. To define a scenario, use the Scenario Wizard and select a leg from each case statement that is encountered in the feature flow. A case statement can be put only inside the feature {} block in the FDL file.

There is no output corresponding to the case statement. The leg selected for defining the scenario is displayed in bold in the right-side "remarks" column in a sequence diagram.

8.1.1 Examples

```
case
    leg "Switchover fails":
        ...
    leg "Switchover succeeds":
        ...
    leg "Switchover aborted":
        ...
endcase
```

Here, a simple case statement is shown where three legs have been specified. As a part of scenario definition, when Scenario Wizard encounters this case statement, it prompts the user to select a leg from the three options.

```
case
  leg "Partial dialing":
         ...
  leg "Complete dialing":
    ...
    case
      leg "Routing failure":
        ...
      leg "Routing successful":
        ...
        case
          leg "Outgoing trunk congestion":
            ...
          leg "Called party busy":
            ...
          leg "No answer":
            ...
          leg "Successful call":
            ...
        endcase
    endcase
endcase
```

Here, a nested case statement is shown. As a part of scenario definition, when Scenario Wizard encounters this case statement, it prompts the user to select a leg from the outermost case statement. If the user selects "Partial dialing", the scenario definition would complete. If, however, the user selects

"Complete dialing", Scenario Wizard will prompt the user to select another leg from the next nested case. If the user selects "Routing successful", the user is again prompted to select a leg from the innermost case statement. If the user selects "Successful call", the scenario definition for a successful call would complete.

8.1.2 Relation with If-Else-Endif statement

- Any case leg traversed in the case-endcase may be used in an if-else-endif or if-endif statement.
- Case-endcase statements may be enclosed within an "if" or "else" part of an if-else-endif or if-endif statements.

8.2 Leg statement

This statement is used to specify legs within a case-endcase block. The legs for the case are included within the case-endcase block. The leg statement defines each scenario option. A scenario determines the flow of a feature under certain conditions. To define a scenario, use the Scenario Wizard and select a leg from each case statement that is encountered in the feature flow. A leg statement can be put only inside the case-endcase block in the FDL.

The leg selected for defining the scenario is displayed in bold in the right-side "remarks" column in a sequence diagram. The leg selections are also available as bookmarks in PDF sequence diagrams. When a PDF sequence diagram is open, the leg selection bookmarks are shown on the left side of the screen. Clicking on the bookmarks will take you directly to the leg selection.

Note: The strings used in this statement should not contain any leading or trailing blanks.

8.2.1 Examples

```
case
    leg "Switchover fails":
        ...
    leg "Switchover succeeds":
        ...
    leg "Switchover aborted":
        ...
endcase
```

Here, a simple case statement is shown where three legs have been specified. As a part of scenario definition, when Scenario Wizard encounters this case statement, it prompts the user to select a leg from the three options.

```
case
  leg "Partial dialing":
    ...
  leg "Complete dialing":
    ...
    case
      leg "Routing failure":
        ...
      leg "Routing successful":
        ...
        case
          leg "Outgoing trunk congestion":
            ...
          leg "Called party busy":
            ...
          leg "No answer":
            ...
          leg "Successful call":
            ...
        endcase
    endcase
endcase
```

Here, a nested case statement is shown. As a part of scenario definition, when Scenario Wizard encounters this case statement, it prompts the user to select a leg from the outermost case statement. If the user selects "Partial dialing", the scenario definition would complete. If, however, the user selects "Complete dialing", Scenario Wizard will prompt the user to select another leg from the next nested case. If the user selects "Routing successful", the user is again prompted to select a leg from the innermost case statement. If the user selects "Successful call", the scenario definition for a successful call would complete.

8.3 If-Else-Endif statement

This statement is used to choose different feature flows from previously encountered leg statements. The string for the if-statement must have been defined as a leg of a previous case statement. When EventStudio encounters the If-Else-Endif statement, it transfers control to the "If block" if the leg corresponding to the "if string" has been chosen in a previous case statement. The control is transferred to the "Else block" if the leg corresponding to the "if string" has not been chosen. Look at the example given below to completely understand how this statement is used.

There is no output produced corresponding to this statement.

8.3.1 Examples

```
case
    leg "DTMF dialing":
        ...
    leg "Pulse dialing":
        ...
endcase
...
if "DTMF dialing"
    call frees "DTMF Receiver"
    call action "Increment DTMF counter"
else
    call action "Increment Pulse counter"
endif
```

Here, a case statement defines the legs "DTMF dialing" and "Pulse dialing". Later, in the feature flow if statement checks if "DTMF dialing" leg was chosen. If "DTMF dialing" leg was taken, control is transferred to the "if block". Otherwise the control is transferred to the "else block". It may be noted that EventStudio remembers the leg selections and user is not prompted for an input when an "if statement" is encountered.

```
case
    leg "Priority Delivery":
        ...
    leg "Standard Delivery":
        ...
endcase
...

case
    leg "Customer Present":
        ...
    leg "Customer Not Present":
        ...
endcase
...

if "Customer Not Present"
    if "Priority Delivery"
       driver action "Call the customer to reschedule delivery"
    else
       driver action "Leave the package at the door"
    endif
else
    call action "Deliver the package"
endif
```

The above example shows how an if-else-endif can be nested within another if-else-endif statement. If-else-endif and if-else statements can be nested to specify the handling for combination of leg statements.

8.4 If-Endif statement

This statement is used to choose different feature flows from previously encountered leg statements. The string for the "if statement" must have been defined as a leg of a previously defined case statement. When EventStudio encounters the If-Endif statement, it transfers control to the "If block" if the leg corresponding to the "if string" has been chosen in a previous case statement. Look at the example given below to completely understand how this statement is used.

There is no output produced corresponding to this statement.

8.4.1 Example

```
case
    leg "DTMF dialing":
        ...
    leg "Pulse dialing":
        ...
endcase
...
if "DTMF dialing"
    call frees "DTMF Receiver"
endif
```

Here, a case statement defines the legs "DTMF dialing" and "Pulse dialing". Later, in the feature flow if statement checks if "DTMF dialing" leg was chosen. If "DTMF dialing" case leg was taken, control is transferred to the "if block". Otherwise the control is transferred to the statement following "endif". It may be noted that EventStudio remembers the leg selections and user is not prompted for an input when an "if statement" is encountered.

```
case
    leg "Priority Delivery":
        ...
    leg "Standard Delivery":
        ...
endcase
...

case
    leg "Customer Present":
        ...
    leg "Customer Not Present":
        ...
endcase
...

if "Customer Not Present"
    if "Priority Delivery"
       driver action "Call the customer to reschedule delivery"
    else
       driver action "Leave the package at the door"
    endif
endif
```

The above example shows how an if-else-endif can be nested within an if-endif statement. If-else-endif and if-else statements can be nested to specify the handling for combination of leg statements.

8.5 Goto and Label statements

The Goto statement allows you to jump forward to a label defined at a later point in the FDL. Use this statement to exit the diagrams from error scenarios.

8.5.1 Example

```
case
   leg "No digits dialed":
     goto exit

   leg "All digits dialed":
endcase

case
   leg "Call routing successful":
   leg "Call routing failed":
      goto exit
endcase

label exit:
  |= call has ended. =|
```

A "goto" to the exit label is being used to end error scenarios in a call setup.

8.6 Remark and Block Remark Delimiters

Remark type	EventStudio 6 delimiters	EventStudio 7+ delimiters
Remarks	(* multi line remark *)	|*multi line remark *|
	(* single line remark *)	% single line remark
Block remarks	[* block remark *]	|= block remark =|

8.7 Verbose Statement Syntax

Statement	EventStudio 6 syntax	EventStudio 7+ syntax
Single entity action	obj takes action "My action"	obj action "My action"
Multiple entity action	obj1, obj2 take action "Joint action"	obj1, obj2 action "Joint action"

Feature	feature "My feature"
	
	endfeature
		feature "My feature" {
		
		}

Sequence	sequence "My sequence"
	
	endsequence
		sequence "My Sequence" {
		
		}