Trace Log to Sequence Diagram Conversion

Browsing through the code traces is an exercise in frustration. Applications produce huge logs that take hours to analyze. The Python scripts presented here let you visualize the traces as sequence diagrams. Interactions between objects are presented visually.

The traces to be included in the visualization are selected by specifying the regular expressions.

Step 1: Sample Trace to Sequence Diagram Conversion

Let's get started by converting a sample trace into a sequence diagram. We start with downloading a few tools:

1. Download and install the free Community Edition of EventStudio System Designer.
2. Download and install the latest version of Python 3.X
3. Download and extract the latest release of the trace to sequence diagram Python scripts
4. Open a command line prompt in the directory where the Python scripts were unzipped.
5. Navigate to the directory where to the Python scripts for this project were downloaded.
6. Convert an included sample trace file into a sequence diagram by typing the following on the command prompt:

```
python trace2sequence.py -i sample_trace.txt
```

7. You should see the following output on the command prompt:

```
EventStudio Community Edition.
Generated sequence-diagram
Generated context-diagram
```

8. Click on the following generated diagrams in the `document` directory:
    - `sequence-diagram.pdf` - A sequence diagram showing object level interactions
    - `context-diagram.pdf` - A context diagram of the object interactions.

Step 2: Customize Regular Expressions to Map Traces to FDL

By now you would have seen a sequence diagram generated from the sample trace output. If you can modify your traces to match the trace output in sample_trace.txt you can skip this step.

In most cases however, there will be changes needed in `config.py` to map your trace format to the FDL input needed by EventStudio.

The `config.py` file lets you configure the trace to sequence diagram (FDL) mapping for:

- Remarks that are shown on the right side of a sequence diagram
- Message interactions
- Method invoke and return
- Object creation and deletion
- Action taken by an object
- State transition
- Timer start, stop and expiry
- Resource allocation and freeing

We will be mapping traces to FDL statements to generate the sequence diagrams. The mapping will also require a basic understanding of regular expressions. Let's visit these topics before we go any further.

Learning FDL - The Sequence Diagram Markup Language

The FDL (Feature Description Language) will be used to generate sequence diagrams. For a quick overview of FDL, refer to the EventStudio User Manual.

Python Regular Expressions

- The Python website has a good introduction to regular expressions.
- Pythex is great for testing your regular expressions.

Templates and Regular Expressions

Trace Template

We start with defining a regular expression that extracts information from the trace format. A regular expression that extracts trace information from sample_trace.txt is shown below:

```
# The trace messages follow this high level format. The current regular expression
# assumes that all traces are of the format:
#
# [time][generator][file]type body
#
# time          The trace begins with time information in square brackets
#
# generator     Entity generating the trace message. This may be a generic entity
#               name. For C++ methods use the calling objects class name.
#               For a C function use the function name.
#
# file          The next square bracket contains filename, line number
information.
#
# type          Defines the type of a trace. The type here is used to determine
the
#               mapping to an FDL statement. Refer to the traceMapper dictionary.
#               traceMapper maps the type to the trace handler that will parse the
#               trace body and extract information for generating an FDL
statement.
```

```
#
# body          This is the text following the type statement. Parsing of this
text
#               depends upon the type of the trace. This file contains the regular
#               expression definitions for parsing of the body for different
#               trace types.

traceRegex = r'\[(?P<time>.*)\]\s*\[(?P<generator>.*)\]\[(?P<file>.*)\]\s*(?
P<type>\S+)\s+(?P<body>.*)'
```

Statement Templates

If you browse `customize.py`, you will see that the trace extraction regular expressions and the FDL statement generation templates are defined to next to each other. You would rarely need to change the FDL generation templates.

Refer to the example below. invokeRegex regular expression extracts named fields, called, method and params. These fields along with the caller are used in the FDL template.

```
# Regular expression for parsing the function/method entry trace body
invokeMethodRegex = r'(?P<called>\w+)(\.|::)(?P<method>\w+)\s*(\((?
P<params>\w+)\))?'

# FDL mapping template for function/method entry
invokeTemplate = '{caller} invokes {called}.{method}{params}'
```

Optional Step 3: High Level Object Grouping and Bookmarks

Generated sequence diagrams can get complex in two ways:

1. Diagrams are long and extend into several pages
2. Too many interactive objects are present on a sequence diagram

Bookmarks

The first problem can be addressed by using bookmarks. Once you define bookmarks you will be able to navigate quickly to different parts of the sequence diagram. We recommend adding your important messages into the bookmark list. An example is shown below:

```
# Add messages that need to be bookmarked in the PDF file. This is useful
# as it lets to quickly navigate through the sequence diagram output of
# a trace. PDF quick navigation bookmarks will be added whenever the messages
# listed below are seen in the trace message.
bookmarks = frozenset({
    'RandomAccessMessage',
    'RRCConnectionSetupComplete',
    'InitialUEMessage',
```

```
    'ReleaseConnection'
    })
```

Specify Object Groupings

The second issue highlighted above may be addressed by grouping objects. You can optionally specify the parents for objects in the `customize.py` file. EventStudio can use this information to generate a high level sequence diagram (Component Level Interaction Diagram).

An example of object grouping is shown in an excerpt from a sample `customize.py` :

```
# EventStudio can generate a high level sequence diagram that can abstract
# out a set of classes as a high level entity. This abstraction is useful in
# understanding the trace output at a higher level of abstraction.
#
# List the interacting entities along with their parent. For example, the
# tuples below indicate that DSP_01 and DSP_23 belong to the same high level
#  PHY entity. This means EventStudio will generate trace output at two levels:
#
# - A sequence diagram where DSP_01 and DSP_23 show up as separate axis.
# - A high level sequence diagram where PHY axis abstracts out the interactions
#   involving DSP_01 and DSP_23
#
# Just include the parent information for external actors in the system. Object
# parents for internal actors are extracted from the trace contents.
objectParents = OrderedDict([
    # Tuples of object and its parent
    # (entity, parent)
    ('DSP_01','PHY'),
    ('DSP_23','PHY'),
    ('CoreNetwork', 'EPC'),
])
```