ACE

a collaborative editor

Report Implementation Testframework

Berne University of Applied Sciences
School of Engineering and Information Technology

Date:	23.06.2005
Version:	1.0.2
Projectteam:	Mark Bigler (biglm2@hta-bi.bfh.ch) Simon Räss (rasss@hta-bi.bfh.ch) Lukas Zbinden (zbinl@hta-bi.bfh.ch)
Receivers:	Jean-Paul Dubois (doj@hta-bi.bfh.ch) Claude Fuhrer (frc@hta-bi.bfh.ch)
Location:	Subversion Repository

Contents

List of Tables

List of Figures

1 Introduction

Testing distributed systems can be complicated. The order of events in the system are generally non deterministic. To specify specific order of events, sessions are depicted in similar figures as in figure 1 in the literature. The beginning of the arrows represent the generation of an operation and the ending of the arrows represent the reception of an operation.

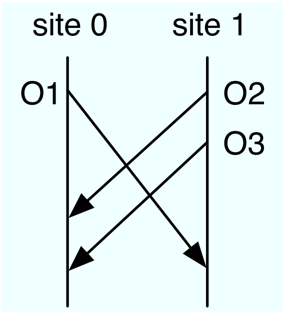

Figure 1: Collaborative editing session

In order to implement a collaborative application, one should be able to specify such scenarios (testcases) with expected results (final states). This is the reason why we decided to implement a testframework.
Effectively the test framework will allow to test the operational transformation algorithm implementation. This is the core part of a collaborative editing application. Other parts of the system will have to be tested differently and are intentionally left out.

1.1 Requirements

The testframework has the following basic requirements:

- specify scenarios as depicted in figure 1
- specify initial and final states
- verify that final states are equal at the end at all sites

Further, the following requirements are given:

- independent of algorithm implementation (testframework uses only algorithm implementation independent classes and interfaces)
- applicable to any algorithm implementation

If not all the aspects of a given algorithm implementation can be tested by the testframework, specific tests have to be written.

2 Architecture

The central classes of the test framework are:

- ScenarioLoader

- ScenarioBuilder
- Node
- NodeVisitor
- Scenario

The `ScenarioLoader` is responsible to load a scenario from a given input stream. The default implementation (`DefaultScenarioLoader`) loads the scenario from an XML file. The `ScenarioLoader` has a single method `loadScenario`. This method accepts a `ScenarioBuilder` and as input an input stream.
The `ScenarioBuilder` is the interface needed to build a scenario. The `ScenarioLoader` uses this abstract interface to inform the builder about the structure of the scenario. The default implementation `DefaultScenarioBuilder` creates a `Scenario` object, which is an object model for a scenario. The default scenario builder represents the scenario as a graph. The nodes of the graph have specific semantic meaning. There are nodes for generation of operations, reception of messages and messages for the start/end of a site's lifecycle.
The `Scenario` object built by the `DefaultScenarioBuilder` contains the nodes topologically sorted. That is, ancestors are visited before their children (note, the graph in figure 1 is a directed acyclic graph).
`Scenario` has a method `accept` that takes a `NodeVisitor` as a parameter. The `NodeVisitor` has methods for all the existing node types. It gets called back for each node in the scenario.
The `ExecuteVisitor` executes a given scenario. That is, it replays the scenario on a given set of objects of an algorithm implementation. The `AlgorithmTestFactory` is used to create the algorithm implementation object for each site. It has also methods for creating documents and timestamps.

2.1 Scenario Graph

To understand the inner workings of the `ch.iserver.ace.test` package, one must know the structure of a scenario. A scenario consists of vertical lines that represent the lifecycle of a site. On these vertical lines, there are certain events. These events can be seen as nodes in the graph. From generation events, there are edges both to the next local event and edges to all remote sites (send events). The reception of a message (represented by a reception node) are another type of node.
The scenario graph has a special property. It is a directed acyclic graph. This special property is a necessary condition to check on a generated scenario object model. It is used to process the graph in correct order (i.e. topological order).

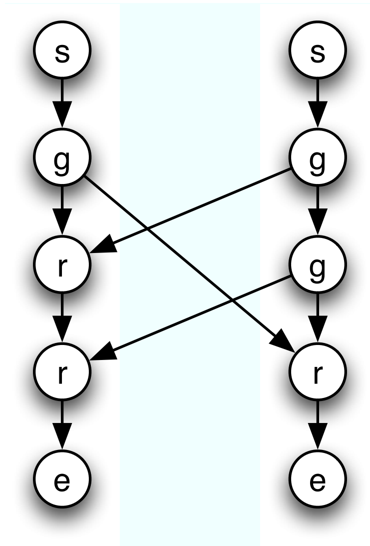

Figure 2: Scenario Graph

So the scenario from figure 1 is transformed into the scenario graph in figure 2. There are four different types of nodes: start (s), generation (g), reception (r) and end nodes (e). The start nodes represent the start of the lifecycle of a site, the generation nodes the generation of an operation, the reception nodes the reception of a remote request and the end nodes the end of a site's lifecycle.

2.2 Node Structure

The nodes of the scenario graph are represented by the `Node` interface. There are four implementations of this interface: `StartNode`, `GenerationNode`, `ReceptionNode` and `EndNode`. The nodes store the successors but not the predecessor. This information is not needed for traversing the nodes in topological order. A distinction is made between local and remote successors in order to facilitate testing of some properties on the graph.

2.3 Properties to Test

There are several properties that have to be checked on the generated scenario object in order to be sure that the scenario is correct. That is, given a correctly working algorithm implementation, the replayed scenario results in converging document states at all sites.

GenerationNode: A `GenerationNode` must have the following properties:

- exactly one predecessor node (local)
- exactly on local successor node
- exactly $n-1$ remote successor nodes
- that is there must be exaclty n successor nodes

ReceptionNode: A `ReceptionNode` must have the following properties:

- exactly two predecessor nodes, one local, one remote
- exactly on successor nodes

StartNode: A `StartNode` has the following properties:

- no predecessor
- exactly one successor

Note that these properties are not explicitely checked by the implementation.

EndNode: An `EndNode` has the following properties:

- exactly one predecessor
- no successor

Note that these properties are not explicitely checked by the implementation.

Other Checks: Beside the basic checks on the nodes, the following checks have to be done:

- no operation is generated and received at the same site
- no operation is received more than once by same site
- no operation is generated twice
- generated graph is a directed acyclic graph

3 How To

This section contains information on both how to create a scenario definition in the standard format (XML file) expected by `DefaultScenarioLoader` and how to use scenario defininitions for tests of concrete algorithm implementations.

3.1 XML Scenario Definition

The default implementation of `ScenarioLoader` loads scenario definitions from an xml file. The XML file conforms to the XSD schema found at `/src/resources/test/scenario.xsd` in the subversion repository. Following is an example of a scenario document:

```
<?xml version="1.0"?>
<scenario initial="ABC" final="AcB">
    <operation id="1" type="ch.iserver.ace.text.InsertOperation">
        <property name="position" value="1"/>
        <property name="text" value="c"/>
    </operation>
    <operation id="2" type="ch.iserver.ace.text.DeleteOperation">
```

```
        <property name="position" value="1"/>
        <property name="text" value="B"/>
    </operation>
    <site id="0">
        <generate ref="1"/>
        <receive ref="2"/>
    </site>
    <site id="1">
        <generate ref="2"/>
        <receive ref="1"/>
    </site>
</scenario>
```

The root element `scenario` defines the initial and final state at all sites. It has two type of child elements, `operation` and `site`. All operations have to be declared. The operation element has an `id` attribute uniquely identifying the operation and a type attribute that specifies what Java object should be instantiated. The `property` child elements allow to set properties on the created operation objects. The `name` attribute specifies the property name and the `value` attribute specifies the value of the property.
The `site` element has an `id` attribute that is used only for informational purpose. A `site` element can contain `generate` and `receive` elements, both of which have a `ref` attribute identifying an operation by identifier. The sequence of `generate` and `receive` elements inside a `site` element represent the sequence of events at one site.
Based on this information in the XML file the scenario graph can be created. The XML schema file itself contains documentation about the structure of XML scenario files.

3.2 Test Algorithm Implementation

In the package `ch.iserver.ace.test` there is a JUnit abstract test case subclass called `AlgorithmTestCase`. This class has a single protected method `execute` that executes a given scenario. It uses the default scenario loader and default scenario builder.
To create a test case for a specific algorithm implementation, you should make the following steps:

1. create subclass of `AlgorithmTestCase`
2. add implementation for methods of `AlgorithmTestFactory` interface. This includes methods for creating the algorithm, initial timestamps and document models.
3. add test methods calling execute with the corresponding scenario as input

The execute methods throws a `VerificationException` if there was an error verifying the final state at all sites. That is, the document states diverged. A `ScenarioException` is thrown if there is an error loading/building the scenario.

4 Outlook

The testframework as it exists today proved that it is generally very useful. However, it has some limitations. Namely, it is not particularly well-suited to testing the *Jupiter* algorithm.

Jupiter does only work correctly in the case of two sites. For more than two sites, a central server is required that forwards requests. It is impossible to specify this server component at the moment. This comes from the fact, that we decided that the testframework must be algorithm implementation independent. Further, it is not possible to test undo/redo operations. This is a shortcoming that should be removed in a future version.

4.1 Planned Improvements

These two points mentioned above are planned to be implemented in order to make the testframework even more useful:

- extend testframework to support testing *Jupiter* algorithm with scenario files
- extend testframework to support undo/redo operations