TEMA Test Engine

09.12.2010

Contents

1 Introduction

This package contains programs for model analysation, model configuring and composing, log analysation and test engine for running tests. In addition to that some documentation is provided.

1.1 Included Programs

1.1.1 Model Analysation

- tema.actionlist: List all action words in test model
- tema.analysator: Analyse test models
- tema.packagereader: Read various information from model package
- tema.simulate: Simulate model
- tema.xsimulate: Simulate model graphically
- tema.validate: Validate model components

1.1.2 Model Conversion

- tema.ats4appmodel2lsts: Convert Ats4AppModel-model to LSTS-model
- tema.mdm2svg: Convert MDM-Model to svg
- tema.model2dot: Generate graphviz models from TEMA-models.
- tema.model2lsts: Convert any TEMA-model to LSTS-model

1.1.3 Model Configuring,Composing and Executing

- tema.composemodel: Compose test model that can be run with tema.testengine
- tema.generatetestconf: Configure model so it can be composed with tema.composemodel
- tema.testengine: Execute test model
- tema.runmodelpackage: Helper program that uses composemodel,generatetestconf and testengine.

1.1.4 Special tools for model composing

These tools are used by other tools and generally are not indended for use by regular user.

- tema.generatetaskswitcher: Generate task switcher for models
- tema.gt: Tool for graph transformations in LSTSes
- tema.renamerules: Tool for renaming rules in rules-files
- tema.rextendedrules: Convert extended rules file with regular expressions to extended rules file
- tema.specialiser: Specialise model

1.1.5 Log Analysation

- tema.log2srt: Generate subtitle file from TEMA test log
- tema.logreader: General information about TEMA test log
- tema.plotter: Convert information generated with tema.logreader to gnuplot format
- tema.sequencer: Generate action sequence from TEMA test log. Action sequence can be given to TEMA test engine as parameter

1.1.6 Other

- tema.filterexpand: Filter test model
- tema.variablemodelcreator: Generate variable models that can be imported in ModelDesigner

1.2 Included Documentation

- Docs/testengine-for-developers: Short introduction for TEMA Test Engine internals
- Docs/ssprotocol: Specification of protocol used for communication between Test Engine and Client Adapters.
- INSTALL: Installation instructions
- README: General user guide
- TemaLib/tema/ats4appmodel/readme: Instructions for using Ats4AppModel-model to LSTS-model converter

2 Models

2.1 Action Machines

Action machines describe user actions on the level that is independent of user interface details. Each action is called action word.

2.2 Refinement Machines

Refinement machines convert the high level user actions (action words) to concrete events in the user interface using keywords.

There is at least one refinement machine for every action machine for every device.

2.3 Localisation

TEMA includes support for localisation, that is, replacing specially tagged strings in action names (esp. keywords) by strings of selected language.

Localisation data is stored in files, each of which contain a table. The header row specifies which languages will be given in the following columns. The first column contains identifiers that will be translated to the specified language in the localisation.

The localisation tables are read in CSV format produced by Excel (fields are separated by semicolons ';'). Example:

```
identifier;en;fi;se
Contacts;Contacts;Kontaktit;Kontakter
Calendar;Calendar;Kalenteri;Kalender
```

To see, how localisation tables are used in the test execution, see Section.

2.4 Data

TEMA tools support using test data. The data can be filenames, contents for text messages, first and last name, phone numbers, addresses, etc.

2.4.1 Defining data

Test data is defined in separate data files. Each line in a data file defines one variable, its optional structure and all possible values. The variable name and the structure are separated from the values with a colon.

For example, "txtfilename" variable that can have four different values is defined as follows:

```
txtfilename: ["abcxyzääö.txt", "a.txt", ".txt", "a"]
```

It is possible to define structured data values. For example, a variable named “contact” can include fields “name” and “number”. This allows using the data so that a name is associated to a single phone number.

In the following, “contact” variable contains two names and their phone numbers:

```
contact(name, number): [("Alice","5554"), ("Bob","55553")]
```

The depth of structures is not limited. In the next example, “name” field in contacts is divided to “first” and “last” fields, but the phone number is without substructure:

```
contact(name(first, last), number): \
     [(("Alice","Allosaurus"),"5554"), \
     (("Bob","Builder"),"5553")]
```

2.4.2 Using data

Data is be manipulated and printed in the transitions of test models. If the label of a transition contains statements between “$(” and “)$”, the statements are evaluted when the transition is executed in the test model. This happens to all transitions, despite the type of the label (keywords, action words and comments). The statements are written in Python.

If the evaluation results in data in OUT variable, the corresponding $(...)$ in the transition label is replaced by the value of OUT. Otherwise, the $(...)$ is replaced by an empty string.

Functions “first”, “next” and “any” can be used to change the value of a variable.

For example, given the data in the above example, execution of transitions

```
start_awEditContact
  -- trying to find $(any(contact); OUT=contact.name.last)$
  ~kw_VerifyText '$(OUT=contact.name.first)$'
  kw_PressHardKey <East>
  kw_VerifyText '$(OUT=contact.name.first)$'
  kw_SelectMenu 'Edit'
  kw_VerifyText '$(OUT=contact.number)$'
  kw_PressHardKey <SoftRight>
end_awEditContact
```

may have resulted in the following execution in the point of view of a test adapter

```
start_awEditContact
  -- trying to find Builder
  ~kw_VerifyText 'Bob'
  kw_PressHardKey <East>
  kw_VerifyText 'Bob'
  kw_SelectMenu 'Edit'
  kw_VerifyText '5553'
  kw_PressHardKey <SoftRight>
end_awEditContact
```

2.5 Building a model

First input file *testconfiguration.conf* needs to be created for **tema.generatetestconf**. Following example file configures model for two target devices with Messaging-application in Android-emulator target and Telephony-application in N95-target. **Tema.packagereader** can be used to determine required information from model.

```
[targets: type]
Emulator: Emulator21
Frank: N95

[targets: actionmachines[]]
Emulator: Messaging%20-%20Create%20MMS.lsts, \
          Messaging%20-%20Create%20SMS.lsts, \
          Messaging%20-%20Inbox.lsts, \
          Messaging%20-%20Main.lsts, \
          Messaging%20-%20Messages%20Interface.lsts, \
          Messaging%20-%20Messages.lsts, \
          Messaging%20-%20Receiver.lsts, \
          Messaging%20-%20Sender.lsts, \
          Messaging%20-%20Startup.lsts
Frank: Telephony%20-%20Call%20Ender.lsts, \
       Telephony%20-%20Call%20Receiver.lsts, \
       Telephony%20-%20Call.lsts, \
       Telephony%20-%20Caller.lsts, \
       Telephony%20-%20Dialer.lsts

[data: names[]]
datatables: Frank.td, Emulator.td, Text%20Messages.td, \
            Multimedia%20Messages.td, \
            Message%20Constants.td, \
            Telephony%20Constants.td

localizationtables: Emulator21.csv, N95.csv
```

Next execute **tema.generatetestconf**

```
$ mkdir NG_Models
$ cd NG_Models
$ unzip ../NG_Models.zip
$ cd ..
$ tema.generatetestconf NG_Models \
                        testconfiguration.conf \
                        runnable_model
```

To build a model, go to the model directory *runnable_model* and run command **tema.composemodel**.

```
$ cd runnable_model
$ tema.composemodel
```

This results in *combined-rules.ext* file. The file contains the necessary low level information on the model components and their synchronisation in their interleaved execution (the parallel composition rules). The model behaviour is calculated on-the-fly during the test run based on this file.

Individual target models in configured model can also be composed.:

```
$ cd runnable_model/Emulator
$ tema.composemodel
```

This results in *rules.ext* file. Composing single target model can be useful for example for model validation purposes.

Easier alternative is to use command **tema.runmodelpackage**. Runmodelpackage automates previous steps and makes it easier to use models. Following command

```
$ tema.runmodelpackage NG_Models.zip \
       --devices="Frank Emulator" \
       --applications="Frank:Telephony;Emulator:Messaging" \
       --targetdir=runnable_model \
       --norun
```

creates runnable target in directory **runnable_model** with devices *Frank* and *Emulator* but does not start testengine.

2.6 Validating a model

You can simulate the execution of the test model step-by-step with command

```
$ tema.simulate combined-rules.ext
```

or use an advanced simulator with a GUI with command

```
$ tema.xsimulate combined-rules.ext
```

Testengine and *guiguidance* can be used to graphically execute models. It is possible to execute them on real SUT

```
$ tema.testengine --model:parallellstsmodel:combined-rules.ext \
                  --guidance=guiguidance ...
```

or on graphically simulated SUT with *guiadapter*

```
$ tema.testengine --model:parallellstsmodel:combined-rules.ext \
                  --guidance=guiguidance \
                  --adapter=guiadapter ...
```

or on simulated SUT with *testadapter*

```
$ tema.testengine --model:parallellstsmodel:combined-rules.ext \
                  --guidance=guiguidance --adapter=testadapter \
                  --adapter-args=model:combined-rules.ext \
                  --testdata=nodata ...
```

Model components can be validated with command **tema.validate**. For example:

```
$ tema.validate VoiceRecorder.lsts
```

Test models can be analysed with command **tema.analysator**. That gives some estimations about size of test models. Because analysing test models takes long time, it is best to analyse single application at a time. Multiple test models can be combined together in analysator and estimation of their size is given. For example:

```
$ tema.runmodelpackage NG_Models.zip --devices=Emulator \
                       --applications=Emulator:Messaging \
                       --targetdir=MessagingOnly \
                       --norun
$ cd MessagingOnly/Emulator
$ tema.composemodel
$ cd ../..
$
$ tema.runmodelpackage NG_Models.zip --devices=Emulator \
                       --applications=Emulator:Contacts2 \
                       --targetdir=Contacts2Only \
                       --norun
$ cd Contacts2Only/Emulator
$ tema.composemodel
$ cd ../..
$
$  tema.analysator multi MessagingOnly/Emulator/rules.ext \
                         Contacts2Only/Emulator/rules.ext
```

creates two test configurations *MessagingOnly* and *Contacts2Only* with one target named *Emulator*. Models are then composed and after that analysator is executed to estimate total size of test model that contains both *Contacts2* and *Messaging* application.

Simulate, xsimulate and validate can be used directly with any *lsts* (action machine or refinement machine), *mdm* or *ext* (parallel composition rules) file.

To list the action words and keywords appearing in the model, run command

```
$ tema.actionlist rules.ext
```

2.7 Converting Models

There are few converters included in package that can be used to convert models between various formats.

- *tema.mdm2svg* can be used to convert MDM-models to svg graphics
- *tema.model2dot* can be used to convert any TEMA-model to graphviz (http://www.graphviz.org/) format. For example to convert LSTS-model to colored svg, following commands might be used:

```
$ tema.model2dot Messaging%20-%20Create%20SMS.lsts --colored | \
                 dot -Tsvg -o Messaging%20-%20Create%20SMS.svg
```

Model2dot can also be used to convert small test models. In example test model with one application is converted to svg image. Different layout algorithm from graphviz is used, because test model has much more states and transitions than single model components.

```
$ tema.model2dot --colored --compact --no-actions \
                 --no-stateprops rules.ext \
                 | sfdp  -Tsvg  -o rules.svg
```

- *tema.model2lsts* can be used to convert any TEMA-model to LSTS.
- *tema.ats4appmodel2lsts* can be used to convert ATS4AppModel-models (http://ats4appmodel.sourceforg) to LSTS-models.

Some problems with different character encodings might be encountered during model conversion. **iconv** can be used to convert between different character encodings in Unix.

```
$ .. | iconv --from-code iso-8859-1 --to-code utf-8 | ..
```

3 Test execution

3.1 Test Engine

You can execute the test engine with **tema.testengine**. For example:

```
$ tema.testengine \
      --model=parallellstsmodel:rules.ext \
      --coveragereq="actions .*fullscreen.*" \
      --
initmodels="lstsmodel:boot.lsts,parallellstsmodel:initrules.ext" \
      --testdata='file:calendardata.td,file:gallerydata.td' \
      --guidance=gameguidance \
      --guidance-args="lookahead:15,randomseed:42" \
      --adapter=socketserveradapter \
      --adapter-args="port:9090" \
      --actionpp=localspp \
      --actionpp-args='file:phones.csv,lang:en' \
      --logger=fdlogger \
      --logger-args='targetfd:stdout,exclude:ParallelLstsModel' \
      --stop-after=1h30m55s
```

The --model and --coveragereq arguments are obligatory, other arguments have some default values. Model argument specifies which test model should be used. Coverage requirement sets the stopping condition for the test run, that is, when the test has passed. In more detail, the arguments are:

3.1.1 --model (obligatory)

The test execution tool can run tests based on

- a single LSTS file (for example, an LSTS drawn with TVT LSTS Editor)
- a single MDM file (ModelDesigner Model)
- TVT parallel composition extended rules file (**rules.ext** that is generated when tema.composemodel is run in a model directory)
- a test log

if an LSTS file contains the model, use:

```
--model=lstsmodel:name-of-lsts-file
```

if an MDM file contains the model, use:

```
--model=mdmmodel:name-of-mdm-file
```

If the model is described in a parallel composition rules file, use:

```
--model=parallellstsmodel:name-of-rules-file
```

If you want to execute exactly the same keywords as in an earlier test run, use:

```
--model=tracelogmodel:name-of-log-file
```

When using **tracelogmodel**, you must have "Adapter:" rows in the log. Do not exclude adapter module from the logger if you want to be able to repeat the test this way.

3.1.2 --coverage and --coveragereq (obligatory)

```
--coverage=clparser \
--coveragereq="req1 and|or|then req2..."
```

There are five coverage module implementations in the TEMA package: *altercoverage*, *clparser*, *findnewcoverage*, *trace_cov* and *dummycoverage*. The arguments of different modules vary. To list the valid arguments of the module, try *--coveragereq-args=help*.

- *dummycoverage* gives always zero percentage for coverage. This can be used for test runs that should run without stopping condition.
- *trace_cov* can be used with action sequences that are generated with tema.sequencer from test log.
- *findnewcoverage* tries to find new things in the model.
- *altercoverage* tries to maximise switches between applications.
- *clparser* implements coverage language.

  If you want to cover every high level action (action word) that appears in the model, use:

  ```
  --coveragereq="actions end_aw.*"
  ```

  If you want to test viewing an image in full screen mode and after that zooming in and out, use:

  ```
  --coveragereq="action .*awFullScreen then (action .*aw-
  ZoomIn and action
    .*awZoomOut)"
  ```

  To find out which actions appear in the test model, run

  ```
  tema.actionlist rules.ext
  ```

  in the model directory.

  Coverage language is described in more detail in Section Coverage language.

3.1.3 --initmodels

```
--initmodels=modeltype1:modelfile1,modeltype2:modelfile2,...
```

The purpose of initialization models is to drive the SUT to a state that is assumed in the test model. This may require booting the SUT, creating or deleting some files in the SUT, and changing settings of applications in the SUT, for instance.

Initialization models are models that end up to a deadlock, that is, a state which is not the source state of any transition. When a deadlock state is reached, the initialization with that model has been ended. Initialization models are executed one by one in the order they are listed. Model type can be any of supported model types, for example, *lstsmodel*, *parallellstsmodel*, *mdmmodel* or *tracelogmodel*. When all initialization models have been successfully executed, the main test starts. An error detected in the initialization prevents also starting the main test.

3.1.4 --guidance and --guidance-args

```
--guidance=guidance-module
--guidance-args=arg1:val1,arg2:val2,...
```

There are nine guidance module implementations in the TEMA package: *gameguidance*, *gameguidance-t*, *greedyguidance*, *randomguidance*, *guiguidance*, *tabuguidance*, *sharedtabuguidance* , *weightguidance* and *oneafteranotherguidance*. The arguments of different modules vary. To list the valid arguments of the module, try *--guidance-args=help*.

- *randomguidance* chooses randomly (parameter *randomseed*) one of the transitions leaving the current state.
- *gameguidance* calculates every path of the required length (parameter *lookahead*) beginning from the current state. It chooses randomly (parameter *randomseed*) one of the paths which give the greatest increase in the coverage with the smallest number of steps. The calculated paths are then used in at least next *rerouteafter* (parameter) steps.
- *gameguidance-t* calculates in background (in another thread) paths beginning from the current state. The maximum length of the paths is limited by parameter *maxdepth*. Next transition is chosen randomly (parameter *randomseed*) among the paths which give the greatest increase in the coverage. This algorithm uses efficiently the waiting time caused by a slow SUT or test interface.
- *greedyguidance* is a breadth first searching algorithm that returns the shortest path improving coverage. If one of the search limits is reached, a random path is selected. There are two possible limiting parameters: *max_states* gives the upper bound to the number of states expanded in a single search, and *max_second* limits the time a single search can last. (Greedy guidance is known as *wormguidance* in some context.)
- *guiguidance* is a manual guidance. Path is manually selected through graphical user interface. *Guiadapter* can be used with *guiguidance* to fully simulate execution graphically.
- *tabuguidance* tries to find actions/states/transitions that are not in a tabulist.

- *sharedtabuguidance* is a guidance that shares a tabulist between processes
- *weightguidance*
- *oneafteranotherguidance* is a special guidance that executes multiple guidances.

3.1.5 --adapter and --adapter-args

```
--adapter=adapter-module
--adapter-args=arg1:val1,arg2:val2,...
```

There are four adapter module implementations in the TEMA package: *socketserveradapter*, *multiplexingsocketserveradapter*, *testadapter* and *guiadapter*. The arguments of different modules vary. To list the valid arguments of the module, try *--adapter-args=help*.

- *Socketserveradapter* forwards keywords to and receives the results of their execution from a TCP/IP port.

  The most important argument is *port*, which is used for specifying which TCP/IP port the adapter listens to (the default port is 9090). To listen to TCP/IP connections in port 3333, use arguments:

  ```
  --adapter-args=port:3333
  ```

  Details of the communication is described in Socket Server Adapter Communication Protocol *socketserveradapter.pdf*

- *multiplexingsocketserveradapter* is a multiplexing version of socketserveradapter. Multiplexingsocketserveradapter allows more than one client adapters to connect to TCP/IP port.

  The most important argument is *clients*, which is used for specifying how many clients will be used. To connect with two clients in port 9090, use arguments:

  ```
  --adapter-args=clients:2,port:9090
  ```

- *Testadapter* can be used to simulate SUT. Most important argument for testadapter is *model*. Model specifies which test model we are testing. Second import argument is delay, which specifies how many seconds adapter waits between keyword executions.

  ```
  --adapter-args=model:rules.ext,delay:0.5
  ```

- *Guiadapter* can be used in conjunction with *guiguidance* to interactively execute model.

3.1.6 --logger and --logger-args

```
--logger=logger-module
--logger-args=arg1:val1,arg2:val2,...
```

There are two logger module implementations in the TEMA package: *fdlogger* and *nologger*. To see valid arguments, run *--logger-args=help*.

- *fdlogger* has following valid arguments:
  - *targetfile:name-of-log-file* writes log entries to the given file.
  - *targetgzipfile:name-of-log-file* writes log entries to the given gzip compressed file.
  - *targetfd:stdout|stderr|number-of-file-descriptor* writes log entries to the specified file descriptor.
  - *exclude:module* ignores log entries from the given module. Module names appear in the test log in the second column (right after the time).
- *nologger* can be used to disable all logging.

Log entries can be written to several targets. For example, you can save the test log to file testlog1.txt and also display it (print it to the standard output) with arguments:

```
--logger-args=targetfd:stdout,targetfile:testlog1.txt
```

3.1.7 --testdata

```
--testdata='file:data1.td,file:data2.td'
```

Testengine supports testdata. Testdata are in files which can be given to testengine with *file* parameter. Testdata can be disabled with parameter *nodata*.

3.1.8 --actionpp and --actionpp-args

```
--actionpp=action-postprocessor
--actionpp-args=arg1:val1,arg2:val2,...
```

Keywords are sent to the test adapter through "action postprocessors".

Tema package contains one action postprocessor: localisation postprocessor *localspp*. You can give it localisation tables in CSV format with *file* argument and specify the default language with *lang* argument. For example:

```
--actionpp=localspp \
--actionpp-args='file:widgets.csv,file:apps.csv,lang:fi'
```

Localspp also supports device specific localisation. In device specific localisation each localisation table is bound to one device. For example:

```
--actionpp=localspp \
--actionpp-args='file:Device.td:widgets.csv,lang:Device1.td:fi, \
                 file:Device2.td:apps.csv,lang:Device2.td:en'
```

3.1.9 --stop-after

```
--stop-after=duration
--stop-after=1h30m5s
```

Stop after arguments sets the maximum length of the test run. It can be given as hours, minutes, seconds, or a combination of them where the largest time units are given before the smaller ones. All numbers must be integers.

3.1.10 --verify-states

```
--verify-states=1
```

All encountered state verifications can be executed with parameter value *1*. Default value is *0* which means no special treatment to state verifications.

3.2 Repeating a test run

There are three ways to repeat a test run.

3.2.1 Deterministic guidance algorithms

Test runs guided by *gameguidance* and *randomguidance* can be easily repeated by using the same the random seed (parameter *randomseed*) in the test runs, assuming that the SUT works deterministically.

If the SUT behaves differently from the previous test run, the new test run may end up executing a trace that is very different from the previous one.

Note, that repeating a test run guided by *gameguidance-t* is more difficult, because if the delays and the processor load are not exactly the same, the search depth may vary. This implies variation in the chosen transitions too.

3.2.2 Coverage requirement based on executed actions

Form a new coverage requirement by connecting the chosen actions in the test log with "then" operator. Use *gameguidance* as a guidance algorithm.

This is the most robust way to try to execute the same (or not too different) trace as in the previous test run. Even if SUT behaves differently, the test guidance tries to guide the run to very similar trace as quickly as possible.

Sequence of executed action words (and keywords) can be created with *tema.sequencer*:

```
$ tema.sequencer --aw-only < testlog.log > testlog.sequence
$ tema.testengine --coverage=trace_cov --coveragereq= \
                  --coveragereq-args=file:testlog.sequence ...
```

3.2.3 Test log is a model

With *tracelogmodel* you can execute exactly the same keywords as in the previous test run. The test log of the previous run is used as a model in the new run.

If the SUT behaves differently, the test run stops immediately.

```
$ tema.testengine --model=tracelogmodel:testlog.log
```

3.3 Log Analysation

After test, *tema.log2srt*, *tema.logreader*, *tema.plotter* and *tema.sequencer* can be used to analyse test logs.

- tema.log2srt generates subtitle file from TEMA test log. First parameter *delay* tells delay between test video and test log. Second parameter *clock skew* tells clock skew between computer that recorded video and computer that executed TEMA testengine. Clock skew can be removed by using Network Time Protocol (NTP) on both computers.
- tema.logreader prints general information about TEMA test log. It is also possible extract information in either **gnuplot** or csv-format.
- tema.plotter selects information to display with gnuplot from *tema.logreader --gnuplot* output format. For example

  ```
  $ tema logreader testlog.log --gnuplot --datarate=10.0s | \
    tema plotter -x s -y kw --term='svg' | gnuplot > keywordsPerSec.svg
  ```

  creates svg-picture with information about keyword execution compared with time.
- tema.sequencer generates action sequences from TEMA test log. Action sequence can be given to TEMA test engine as parameter

3.4 Coverage language

Syntax of the language is the following:

```
cov-req  ::= elem-req | cov-req ("and" | "or" | "then") cov-req
elem-req ::= ("action" | "actions") regexp
```

where *regexp* is an regular expression matching names of actions in the test model. Parenthesis can be used around coverage requirements.

In elementary requirements (*elem-req*) with *action* you express that you want to cover at least one of the actions matching the regular expression. With *actions* you require that every action matching the regular expression should be covered.