Model-Based Testing with a General Purpose Keyword-Driven Test Automation Framework

Tuomas Pajunen, Tommi Takala, and Mika Katara
Department of Software Systems
Tampere University of Technology, Finland
mika.katara@tut.fi

***Abstract*—Model-based testing (MBT) is a relatively new approach to software testing that extends test automation from test execution to test design using automatic test generation from models. The effective use of the new approach requires new skills and knowledge, such as test modeling skills, but also good tool support. This paper focuses upon the integration of the TEMA model-based graphical user interface test generator with a keyword-driven test automation framework, Robot Framework. Both of the tools are available as open source. The purpose of the integration was to enable the wide testing library support of Robot Framework to be used in online model-based testing. The main contribution of this paper is to present the integration providing a base for future MBT utilization, but we will also describe a short case study where we experimented with the integration in testing a Java Swing GUI application and discuss early experiences in using the framework in testing Web GUIs.**

I. Introduction

System-level test automation has gone through multiple generations, where each new generation has raised the level of abstraction used in the test design. The state of the art test automation, the keyword-driven testing process, abstracts the implementation of tests behind high-level actions, i.e. keywords [1, pp. 446-464][2]. In GUI testing, keywords typically depict basic user actions, such as pressing keys, typing or reading text. The tests are built as sequences of keywords, and keywords are automatically translated into concrete low-level scripts. The abstraction can be increased further by dividing keywords into different hierarchical levels. For instance, low-level concrete scripts may be abstracted under lower-level keywords, and lower-level keywords under higher-level keywords.

The two main benefits of keyword abstraction are the reduced amount of maintenance work related to tests and the fact that the tests are easier to build and understand. When a detailed implementation of an action is in one script, the maintenance work has to be performed only to that script. Understanding and creating tests do not require programming expertise since keywords are easy to understand, because they are not low-level system commands, but rather general commands familiar from everyday usage. A keyword script also is much shorter than a more traditional test script.

Model-based testing (MBT) is a way of automating test design. It is defined as an approach that uses a model of the system under test (SUT) in testing tasks [3, pp. 825-837]. Model-based testing can be seen as a specification-based test generation approach in which the model is the specification. The term model-based testing is a generic term used for several test generation techniques.

Utting and Legeard [4, pp. 6-8] present four test generation approaches in model-based testing: 1) Generation of test input data from a domain model, 2) Generation of test cases from an environment model, 3) Generation of test cases with oracles from a behavior model, and 4) Generation of test scripts from abstract tests. The first three approaches are all based on test generation from a model. The domain model describes the domains of input data that can be given to the system and the environment model the expected environment, such as operation frequencies, of the SUT. Both models can be used to generate input for the SUT, but either does not include the expected output of the system, thus requiring manual work for verification. The third model, behavioral model, includes oracle information about the expected behavior of the system and can thus be used to detect any irregularities in the output of the SUT automatically. The forth approach does not include models as such, but rather test cases that are described in a high-level of abstraction, without the low-level implementation details. Basically a script defined with high-level keywords could be categorized as the fourth approach. This paper uses the term model-based testing in the third meaning.

The test execution in MBT can be divided into *online* and *off-line* testing. In online testing the tests are generated at the same time as they are executed with an adapter tool. Online testing is good for testing a nondeterministic SUT and for long-running test sessions. [4, p. 376]. Offline testing refers to an approach where the test execution and generation are carried out separately.

The objective in this paper is to describe the integration of an online model-based testing tool, the TEMA toolset, with a keyword-driven test automation framework, Robot Framework [5], both available as open-source. The integration enables the use of the wide library support of Robot Framework to broaden the testing scope of the TEMA toolset. Robot Framework has several libraries for different domains, which the integration enables for model-based testing use, including a library for testing Java Swing GUI applications. A short case study was performed with a Swing application, jEdit, to trial the integration. We also report early experiences in using the framework in testing Web GUIs

The paper is structured as follows: Robot Framework and the TEMA toolset are introduced in Section II and Section III describes their integration. Section IV describes the case study and Section V presents the conclusions.

II. Background

The TEMA toolset is a set of model-based testing tools, which executes tests on the basis of keywords. Robot Framework is a generic keyword-driven test automation framework with several keyword libraries for various domains. In the testing tool integration, the former performs the model-based test generation, whereas the latter introduces the keywords used, and handles the concrete lower-level keyword execution on the SUT.

A. Robot Framework

Robot Framework [5] is a generic keyword-driven test automation framework meant for acceptance level testing. The tool uses keyword abstraction for test design. Keywords are divided into higher-level user keywords and lower-level library keywords. The user keywords are built by creating combinations from the keywords presented in the keyword libraries. Test cases are described with scripts that use the keywords and control structures, such as loops.

The available keywords of Robot Framework are defined in libraries. New libraries can be implemented with Python or Java. There are two types of libraries: standard and external. The standard libraries are distributed with Robot Framework and the external libraries are released in separate packages:

Standard libraries	
BuiltIn	Keywords for generic testing needs, such as variable verification, conversions and delays.
OperatingSystem	Keywords for operating system tasks, such as file system operations and executing commands.
Telnet	Keywords for Telnet connection handling.
Collections	Keywords for handling lists and dictionaries.
String	Advanced string handling keywords.
Dialogs	Keywords for getting user input during test execution.
Screenshot	Keywords for capturing and storing screenshots.
Remote	Proxy between Robot Framework and a test library.

External libraries	
Swing	Keywords for Java Swing GUI handling.
SSH	Keywords for executing commands over an SSH connection.
AutoIt	Keywords for Windows GUI handling with the AutoIt tool.
Database	Keywords for database handling.
Selenium	Keywords for web page handling with the Selenium tool.

Table I presents an example test suite. The first section of the table defines the different keyword libraries that are included to the test, the second one initializes some of the variable used in the test cases, the third one defines the test cases in the suite and the final section defines the high-level user keywords used in the test cases with the keywords defined in the libraries. The test suite files for Robot Framework can be in HTML, plain text, TSV (tab-separated values) or RST (reStructuredText) format.

Robot Framework is a command-line tool and can be launched using either a Python or a Jython interpreter. The core framework itself is Python code, but Jython is required when using libraries built with Java (such as the Swing library). The file that contains the test suite to be executed

Table I: An HTML format test data file including a test case to test jEdit file opening in Robot Framework

Setting	Value
Library	OperatingSystem
Library	SwingLibrary

Variable	Value
${jedit}	org.gjt.sp.jedit.jEdit
${inFile}	\code\testing.cpp
${outTitle}	jEdit - testing.cpp

Test Case	Action	Argument	Argument
File opening	Start Application	${jedit}	
	Open File	${inFile}	
	Check Title	${outTitle}	

Keyword	Action	Argument	Argument
Open File	[Arguments]	${file}	
	Select Window	jEdit	
	Select From Main Menu	File|Open...	
	Select Dialog	File Browser	
	Insert Into Text Field	filename	${file}
	Push Button	Open	
Check Title	[Arguments]	${expTitle}	
	Select Window	jEdit	
	${actTitle}=	Get Selected Window Title	
	Should Be Equal	${actTitle}	${expTitle}

is given as a parameter. When launched, Robot Framework will execute the test cases in order given in the test suite file and generate log files for results in HTML format

B. TEMA Toolset

The TEMA toolset is a set of model-based testing tools designed for domain-specific GUI application testing [6], [7]. The TEMA toolset was originally designed for GUI testing of smartphone applications, such as those for S60 and Android platforms, but can be extended to support different domains. The toolset provides a complete set of model-based testing tools from modeling to test generation.

Figure 1 presents the architecture of the TEMA toolset including Robot Framework integration for test execution. The architecture has been designed to support different kinds of SUTs. The main components of the architecture are the following [7]:

1) Model Design for creating and maintaining models of the SUT.
2) Test Control for setting up and launching tests.
3) Test Generation for generating and executing tests.
4) Keyword Execution for executing keywords on the SUT.

When testing a new GUI application for a domain that the TEMA toolset supports, a major share of the work lies in the modeling. The model of the application is created with the Model Designer tool. When the domain of the application is not supported, a significant amount of work goes into building the implementation for keywords, i.e. implementing a new Keyword Execution component. In the domain-specific approach every new application does not need a new Keyword Execution component. This reduces the amount of work related to the Keyword Execution compared to the application-specific approach.

Model Design: TEMA models describe the functionality of the SUT from the user perspective. They define what inputs can be given to the system and how the system should respond. The models are created using an Eclipse-based graph editor called Model Designer [8].

The model in the TEMA toolset is built using a transition-based notation, Labeled State Transition System (LSTS). The exact definition of LSTS and how it is used in a modeling task in TEMA can be found from [7]. In general, transition-based notations focus on describing transitions between different states. They are mainly graphical node-and-arc notations. The nodes represent the states of the system and the arcs the actions of the system. The modeling formalism used in TEMA allows models to be divided to separate model components for easier test maintenance. The model components are then combined to a complete test model using a specific automatic parallel composition method. In addition to enabling the creation of maintainable models, this scheme facilitates concurrent testing of multiple application or devices.

The model components in TEMA are composed of two-level state machines: higher-level *action machines* and lower-level *refinement machines*. An action machine describes a part of an application, such as an independent view, in a high-level of abstraction. A refinement machine defines the functionality of an action machine for a specific device model in the case of the smartphone domain. With such a division, the same action machines can be used with multiple device models; only the refinement machines need to be changed, which greatly decrease the modeling effort.

The transition of an action machine is an *action word* or a synchronization word. A synchronization word can change the active action machine, based on the parallel composition method. Action words define user actions in a very high level that is not dependent on the exact model of the SUT. As an example of an action word, ”awSendMMS” could specify a sending of a multimedia message in a messaging application. For verifying that the SUT and model states are equivalent, the states of the action machine can contain *state verifications*.

The implementation of an action word is defined in the refinement machine. The transitions of the refinement machine are labeled with keywords that map the action words to concrete GUI operations, such as pressing keys and selecting items from a menu. Keywords can also contain simple statements from the Python programming language (e.g for storing and generating data) and references to separate test data and localization data tables. Test data tables contain all the input that should be modifiable by the tester. The localization table includes strings for different language variants of the SUT. The state verifications are implemented similarly to action words.

Test Control: The Test Control contains a Web GUI, which can be used to define and execute model-based tests. Tests are designed by defining so called ”test configurations” for different modes of test generation. A test configuration can be a smoke test, a bug hunting test, a basic coverage test or a use case test. The smoke test goes through the model transitions randomly for a specified period of time. The bug hunting test goes through the transitions of the model using some algorithm, such as random heuristic, without specific objectives and will continue until a bug has been found. The basic coverage test goes through those elements in the model that the modeler has pointed out as especially important (key functionality of the application). The use case test is based on a sequence defined using action words and logical operators (AND, OR, and THEN).

Test Generation: The Test Generation transforms the test configurations into keywords and controls the test execution logic using the online approach. During the test execution, the executed keywords are selected on the basis of the test configuration, the model, the test generation algorithm and the results of previously executed keywords.

In practice, the online approach is utilized by defining

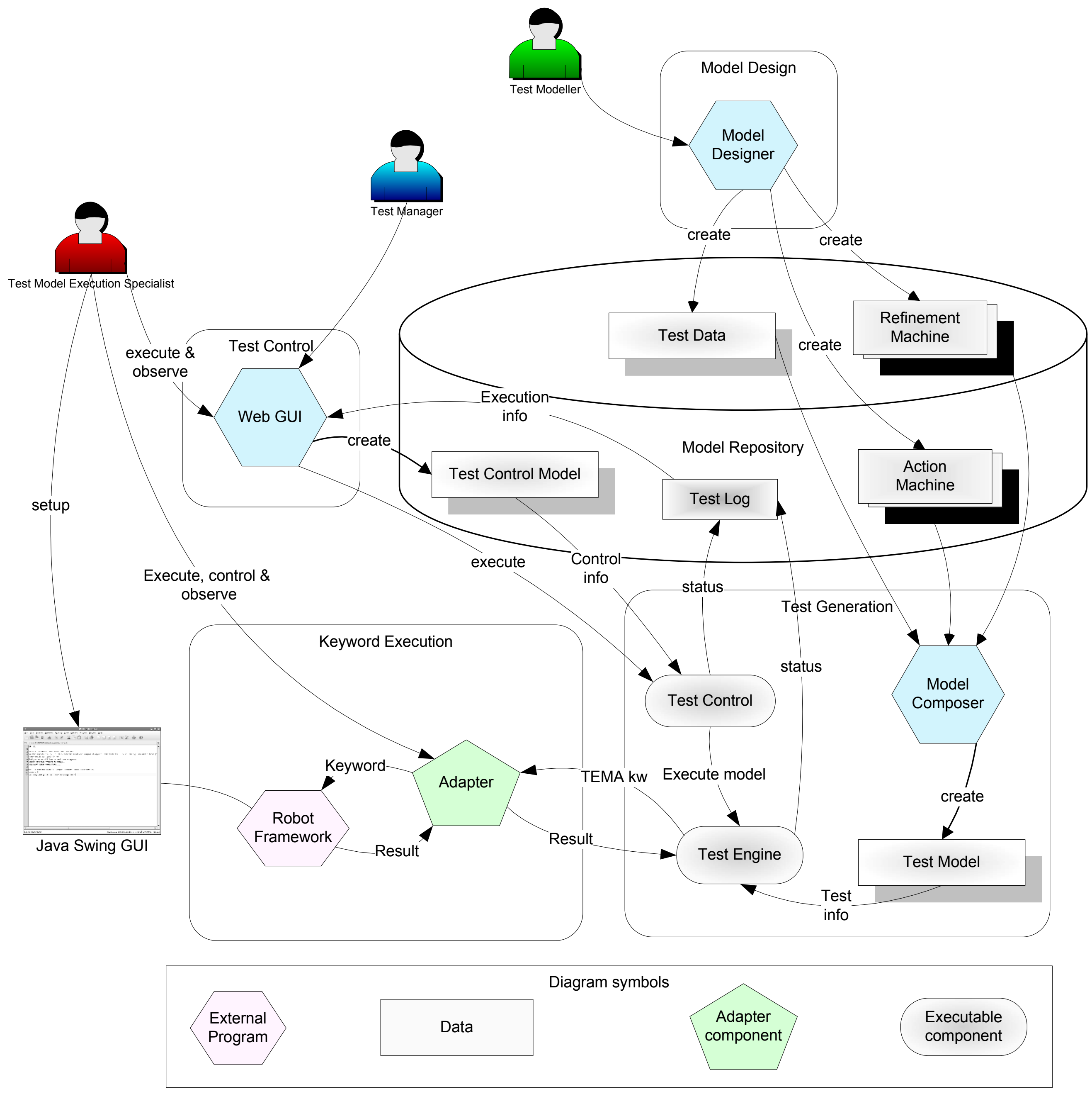


Figure 1: TEMA architecture

branches to the models in both action machine and refinement machine level. In refinement machines, two branches can be defined for a keyword based on its result: true or false. The same branching can be used in the action machines, but the correct branch is chosen based on the successful execution of an action word.

Keyword Execution: After the test generation has provided the keywords, the Keyword Execution component of the toolset performs the concrete low-level test execution on the SUT. The Keyword Execution usually has two subcomponents: the adapter and the external test tool. The external test tool translates the keywords into low-level operations, which are executed on the SUT. The adapter handles the communication between the Test Engine and the External test tool: it creates a connection, requests keywords and responses with keyword execution results. In the case study presented in this paper, Robot Framework works as the external test tool.

III. Tool Integration

This section first presents an overview of the integration between the TEMA toolset and Robot Framework. Second, we discuss OnlineLibrary, which is an online testing library that was implemented for Robot Framework to enable the online testing approach. Thirdly, we present how the communication is handled between the OnlineLibrary and the TEMA Test Engine with an Adapter unit. Finally, the available keywords and the syntax restrictions are summarized.

A. Overview

One integration has already been implemented between the TEMA toolset and Robot Framework [9]. The integration carries out the test execution and generation on the basis of an off-line approach, and thus cannot utilize the online features of TEMA. Our goal was to build an integration that supports the online approach, allowing more flexibility in the tests, such as preparing to changing SUT responses and enabling long-period testing.

In the online approach of the TEMA toolset the tests are generated during the test execution, which requires an interactive keyword execution tool. Such a tool carries out the keyword execution in short cycles. A cycle consists of reading input data for the keyword, parsing the data, executing the corresponding keyword and reporting the keyword execution result. By default, Robot Framework is based on running static test cases and carries out the test execution in different phases: reading and parsing all the test data, executing all the keywords in the test cases and finally reporting the results of the cases. Therefore, Robot Framework had to be modified for the online testing to support short keyword execution cycles.

A library called "OnlineLibrary" was implemented for this task. Figure 2 shows the overview of how the OnlineLibrary is used to receive keywords interactively from the TEMA toolset. In place of the Swing Library any library or multiple libraries of Robot Framework can be used. The OnlineLibrary uses the internal structures of Robot Framework to parse and execute the received keyword and then passes the keyword execution result to the TEMA toolset. In contrast to the usual test execution of Robot Framework, the keywords are received from the OnlineLibrary and not from the internal test suite structure.

B. OnlineLibrary

Creating a new library fits well into the architecture of Robot Framework and was considered to be the most suitable approach to achieve the online execution. The OnlineLibrary can interactively execute keywords from library and resource files. The library uses a socket connection to receive the keywords. Connection parameters (host, port) are defined when the library is imported. The OnlineLibrary is imported in the same way as any other library in Robot Framework. The keywords are also used like any other keywords of Robot Framework, but when a **Run Online** keyword is executed, the following keywords are received from the online connection until a **Stop Online** keyword is executed.

When the online keyword execution is started with **Run Online**, the OnlineLibrary waits for a line of keyword data from the online connection. When the keyword data is received, the data is split into keyword elements using a separator string. The keyword elements are used to create a Keyword object with the internal Keyword Factory methods of Robot Framework. The Keyword object is then executed in Robot Framework. When the execution is completed, the OnlineLibrary sends the keyword execution result (pass or fail) to the online connection. Then the OnlineLibrary is ready to handle the next keyword from the online connection. When the **Stop Online** keyword is received, the online keyword execution ends and Robot Framework returns back to normal keyword execution. In addition, the library contains the **Set Parameter Separator** keyword that defines the separator string which is used for splitting the keyword data into keyword elements.

C. Adapter unit

A separate Adapter unit needed to be implemented for the communication between Robot Framework and the TEMA toolset. The Adapter unit is a modification of the TEMA adaptation made for Linux GUI applications [10]. It handles the keyword and the execution result exchange between the OnlineLibrary and the Test Engine. Figure 3 illustrates the structure of the Adapter unit and the connections to the Test Engine and the OnlineLibrary.

The common execution cycle of the Adapter unit consists of three main steps: requesting a keyword, executing the keyword and sending the execution results. The cycle continues until there are no more keywords for execution or when the SUT falls out of sync with the models.

D. Keywords

All the keywords of the imported resource files and the libraries of Robot Framework can be used in testing with a few exceptions. The OnlineLibrary does not support the `:PARALLEL`, `:FOR`, `...`, and `[setting]` control structures, but they can be used to define user keywords. TEMA keywords have only boolean return values indicating the success of executing a keyword, whereas Robot framework keywords can return, e.g. string values. Therefore, such keywords can only be used in defining user keywords, not in the TEMA models. Typically Robot Framework has a keyword for fetching some data from a GUI object and another keyword for verifying the data. These two keywords can be combined to a single user verification keyword that can then be used in the TEMA models.

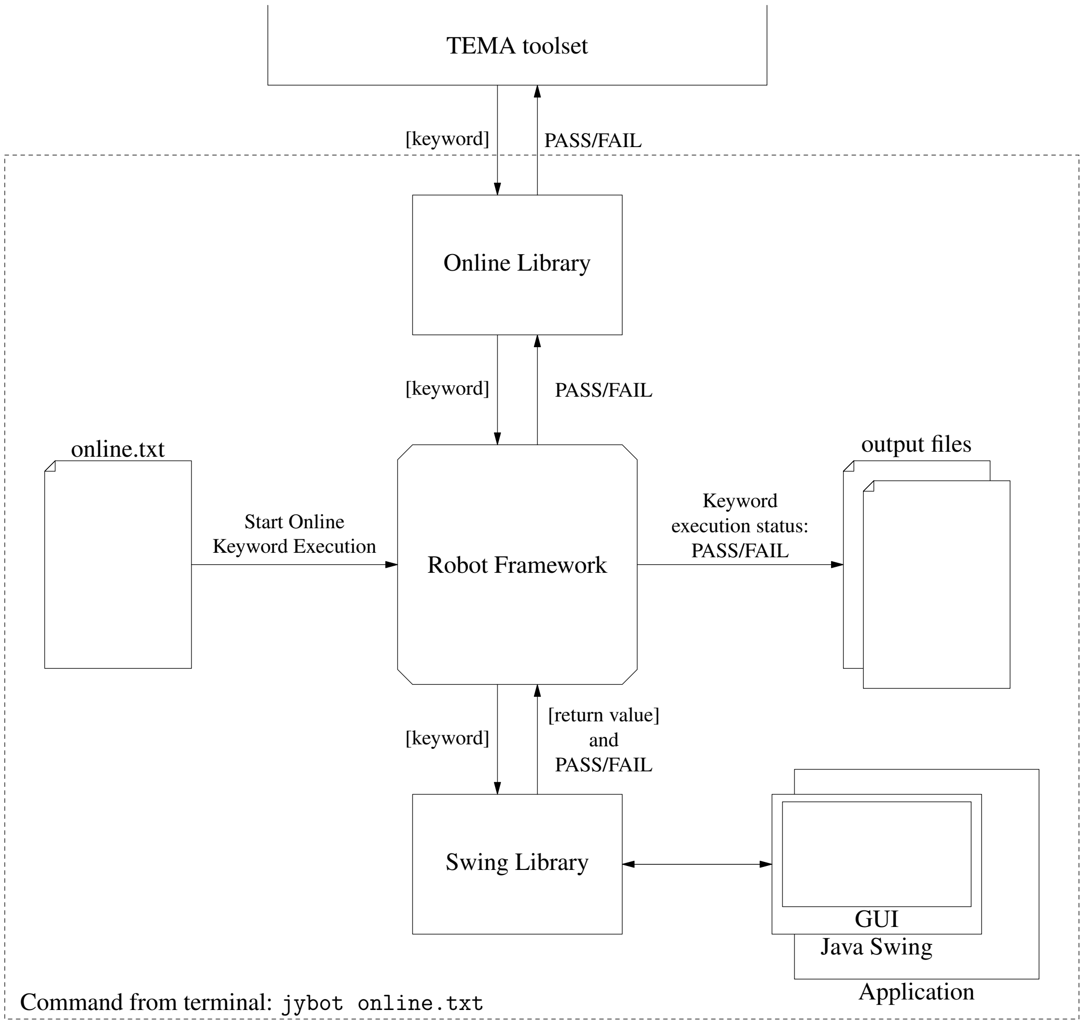


Figure 2: Robot Framework online integration

IV. Case Study: Testing Java Swing GUI

This section introduces a case study where the integrated tools were used in model-based testing of a Java application, jEdit. It should be noted that a complete testing of jEdit exceeds the scope of this paper. Hence, only a small portion of jEdit was modeled and tested. The purpose of this small scale testing was to find out what kinds of issues arise out of the model-based testing of a text editor application using the new tool integration. A significant effort was used investigating how jEdit works, figuring out how it can be tested, learning modeling, building the model, and solving various issues that emerged during the testing. These issues include e.g. how to access a particular GUI component, what should be verified, or why a piece of testing data blocks the test execution.

Swing Library

The Swing library of Robot Framework was used for testing of jEdit. The library is implemented with Java and uses Abbot [11] and Jemmy [12] for handling the Swing GUI events. The library has keywords for keyboard events, application launching and handling various GUI components, such as checkboxes, buttons and text fields. The keywords are documented in [13].

Many of the keywords need an identifier parameter to specify the targeted GUI component. A GUI component can be identified by index or by name. A software developer can set the name of the component with the setName method of the Component base class of Swing. If the developer has not given a name for the component, one can try to access the component with an index number. Another workaround involves sending keyboard events, for example tabular key event for changing the focused component. However, the test maintenance is easier, when the component name is used for the identification. For finding out object identifiers and to explore the GUI of an Swing application, the Swing Explorer [14] can be used.

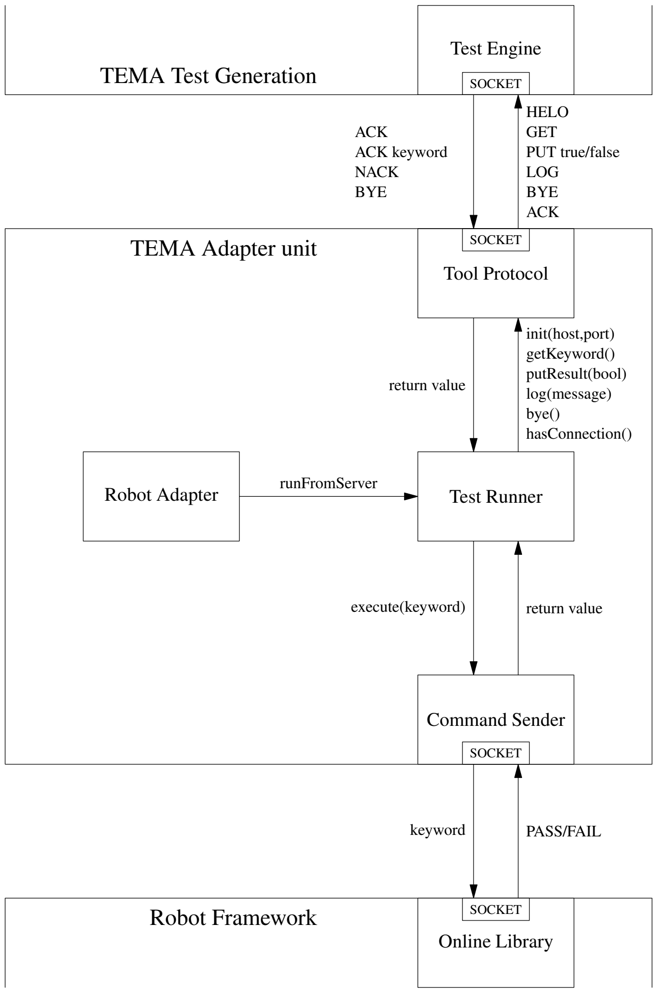


Figure 3: Adapter between Test Generation and Robot Framework

jEdit Overview

jEdit [15] is a versatile open-source programmer's editor written in Java. As a Java application, it can be used in various operating systems. It has an extensible plug-in architecture with a number of plug-ins available. jEdit provides a good many features that programmers might need for editing, such as auto indentation, syntax highlighting, text folding and syntax support for several programming languages.

The jEdit GUI is made up of two main components: the main menu and the text editing area. Various shortcuts and additional information elements can be added to the GUI area and some are visible by default. The main menu contains ten sub-menus: File, Edit, Search, Markers, Folding, View, Utilities, Macros, Plugins and Help. Every menu has as an average of more than ten operations and the execution of an operation often requires additional input values. In total, jEdit provides a very large set of functionality.

Modeling jEdit

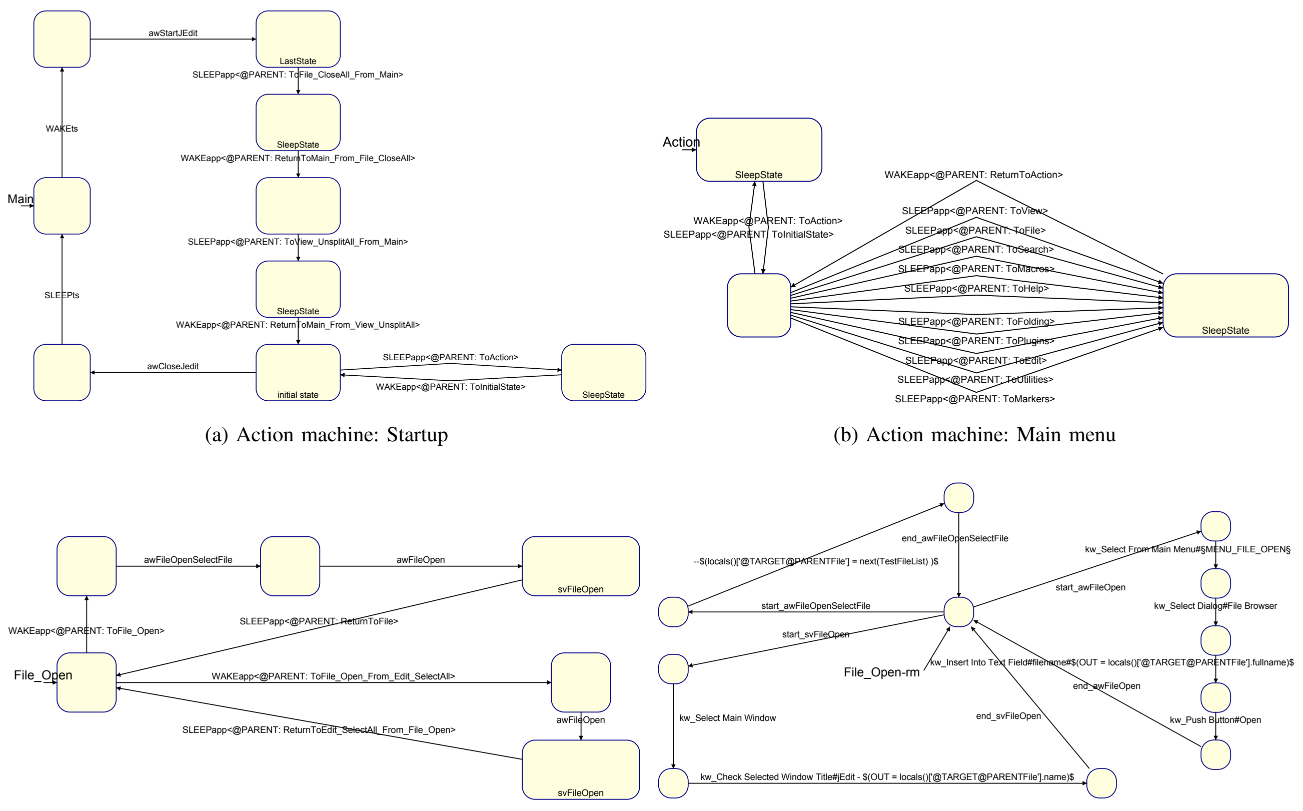
The starting point for the model is the launching of the application. The start-up is modeled with a single action machine, which contains action words to start and close jEdit and state verifications to confirm the initial state of the SUT (Figure 4a). In addition, the start-up machine contains a synchronization word to move the execution to other action machines that model the rest of the functionality of the application. The transitions beginning with ”SLEEPapp” or ”WAKEapp” are the synchronization words used by the parallel composition. Once a ”SLEEPapp” transition is executed, the parallel composition will move the execution to other action machine that has the corresponding ”WAKEapp”. As an example, the ”SLEEPapp<@PARENT: ToAction>” in the Startup model is executed synchronously with the ”WAKEapp<@PARENT: ToAction>” found from the main menu action machine (Figure 4b).

From a modeling perspective, when jEdit has been started, more than a hundred operations are available. Accessing all these operations from the same action machine would make the action machine large and hard to maintain. Therefore, the operations are divided into different action machines. The action machines can be divided into three hierarchical levels: main menu, menu and operation.

From the start-up action machine the execution moves to the main menu action machine (Figure 4b). The main menu action machine does not contain any functionality of its own, but it contains synchronizations for all the menu action machines that are available from the main menu. There are ten menu action machines that can be reached: File, Edit, Search, Markers, Folding, View, Utilities, Macros, Plugins and Help. In addition, the execution can move back to the start-up.

Similar to the main menu action machine, a menu action machine can only synchronize operation action machines and the main menu action machine. A menu action machine can activate the operations which are under the corresponding menu. For instance, a File menu action machine can activate action machines for the operations New File, Open, Save, Save As, Load, Close, Print, etc. The resulting action machine is very similar to Figure 4b.

An operation action machine contains action words and state verifications for the corresponding operation. An operation action machine can activate the menu action machine which activated the operation. An operation action machine can also activate other operations to perform some sub-operation for the original operation. Figure 4c presents an example of an action machine for the File Open operation. In this model only the operation level action machines and the start-up action machine have action words and state verifications, and non-empty refinement machines. Figure 4d shows an example of a refinement machine for the File Open operation. The transitions beginning with ”aw” in the action


awStartJEdit
LastState
SLEEPapp<@PARENT: ToFile_CloseAll_From_Main>
SleepState
WAKEapp<@PARENT: ReturnToMain_From_File_CloseAll>
WAKEts
Main
SLEEPapp<@PARENT: ToView_UnsplitAll_From_Main>
SleepState
WAKEapp<@PARENT: ReturnToMain_From_View_UnsplitAll>
SLEEPts
awCloseJedit
initial state
SLEEPapp<@PARENT: ToAction>
WAKEapp<@PARENT: ToInitialState>
SleepState
Action
SleepState
WAKEapp<@PARENT: ReturnToAction>
WAKEapp<@PARENT: ToAction>
SLEEPapp<@PARENT: ToInitialState>
SLEEPapp<@PARENT: ToView>
SLEEPapp<@PARENT: ToFile>
SLEEPapp<@PARENT: ToSearch>
SLEEPapp<@PARENT: ToMacros>
SLEEPapp<@PARENT: ToHelp>
SLEEPapp<@PARENT: ToFolding>
SLEEPapp<@PARENT: ToPlugins>
SLEEPapp<@PARENT: ToEdit>
SLEEPapp<@PARENT: ToUtilities>
SLEEPapp<@PARENT: ToMarkers>
SleepState
awFileOpenSelectFile
awFileOpen
svFileOpen
SLEEPapp<@PARENT: ReturnToFile>
WAKEapp<@PARENT: ToFile_Open>
File_Open
WAKEapp<@PARENT: ToFile_Open_From_Edit_SelectAll>
awFileOpen
SLEEPapp<@PARENT: ReturnToEdit_SelectAll_From_File_Open>
svFileOpen
end_awFileOpenSelectFile
--$(locals()['@TARGET@PARENTFile'] = next(TestFileList))$
kw_Select From Main Menu#$MENU_FILE_OPEN$
start_awFileOpen
kw_Select Dialog#File Browser
start_awFileOpenSelectFile
start_svFileOpen
File_Open-rm
kw_Insert Into Text Field#filename#$(OUT = locals()['@TARGET@PARENTFile'].fullname)$
end_awFileOpen
kw_Select Main Window
end_svFileOpen
kw_Push Button#Open
kw_Check Selected Window Title#jEdit - $(OUT = locals()['@TARGET@PARENTFile'].name)$

(a) Action machine: Startup

(b) Action machine: Main menu

(c) Action machine: File open

(d) Refinement machine: File open

Figure 4: jEdit models

machines are the action words and their implementations are given using keywords in the refinement machines between ”start_aw” and ”end_aw” transitions.

Seven operations of jEdit were modeled. These were four File menu operations New, New in Mode, Open and Close All and two Edit menu operations Copy and Select All and one View menu operation Unsplit All. Testing jEdit only with these operations is far from sufficient, but it reveals issues that are related to model-based testing of Java GUI applications and editor applications. It is also worth mentioning that the way jEdit was subdivided and modeled here is not the only approach, others just as good could exist.

Modeling and Testing Issues

When testing a text or data intensive application, such as the editor application jEdit, the lack of text handling support complicates the modeling. When the character support is restricted or there is no support for multi row string handling, the complexity of the model increases, because text handling logic needs to be added to the model itself.

The modeling of jEdit gave indications that a versatile and flexible model requires that various details about the state of testing are stored in the model. For instance, state information like:

- Name of the file which is currently in the text editing area.
- Current content of the text editing area.
- Names of the files that are open.
- Names of the modified files.
- Content of the clipboard.
- Marked lines.

In TEMA, Python statements can be used to store this kind state information into data variables.

In the implementation of action words and state verifications it was noted that, generally, the action words are easy to implement with keywords. Performing a user action is usually straightforward if the interface design has been successful. However, state verifications can be more complicated to perform, as the application was not built to support automatic testing.

Because no specifications were available during modeling, it was sometimes difficult to find the information about how the application should behave. In the case of some operations, it is quite clear what should happen, whereas other operations can be interpreted in many ways. This was not a problem in this case study, for the modeler can decide how the SUT should work. In general, this is a more important issue, which ought to be verified from requirements or from a person with the specification responsibility. If modeling is done as early as at the beginning of a project, raising unclear issues of this kind may help the requirements specification process.

A basic comparison issue needs to be solved for state verifications. But how much shoud be verified? An operation may cause various changes in the SUT. For example, when a file is opened, the title of the window changes, the content of the file is visible in the text editing area, and file-related settings, such as encoding, are changed. Thus, it should be specified what a state verification does when a file is opened; all the verifications or only some of them. In the case study, when a file was opened, it was enough to check that the window had the correct title. The content of a file can be checked in connection with some other operation, e.g. content reading. Thus, if the tester wants to make a test of opening a file and checking the content, both of these operations need to be done as well as the verifications related to them. This seems to be a logical solution for modeling. If the model is designed just like a normal test case, it may end up being awkward. The same verifications would recur in various action words and the model would lose some of its maintainability and readability.

Accessing custom GUI components directly is not possible in many cases. New application-specific keywords might be implemented to enable direct access to custom components. There are two workarounds for the accessing problem: sending keyboard events, or using other operations for partial access. The biggest problem in testing jEdit was accessing the text editing area, which is not a standard Swing component. However, a workaround was found. Sometimes an operation does not have a suitable keyword in the library keywords. For example, several views can be opened in jEdit, but there is no keyword to verify that several views are visible.

Testing Web GUIs with Selenium Library

Using the Selenium Library in MBT to test web applications is currently an ongoing task, but we discuss some of the experiences that have been gained to give another viewpoint using the tool integration. Selenium Library is based on the popular web application testing tool Selenium [16] and defines a keyword interface library for Robot Framework.

Currently, we have created a small model for the Amazon.com online store. The model contains functionality for searching shopping items defined in test data, adding items to the shopping cart, removing items from the cart and verifying that the cart is updated correctly.

Selenium Library is used exactly the same way as the Swing library in the jEdit case. The library is specified in the test data file (Table I) and if necessary, new custom keywords are defined. For our models, the existing set of keywords was sufficient. The HTML elements that can be accessed using Selenium Library keywords are defined using the key attributes of elements such as id, name, href or text content. In addition, XPath or DOM can be used to specify any arbitrary element.

The dynamic nature of modern Web GUI causes some problems in automating their testing. Creating keywords that can e.g. click a specific element in the view can be problematic if the element has no unique attributes (such as the id-attribute) or these attributes change every time a page is generated. In these situations, one might need to use detailed XPath-clauses that are prone to change. The problems are similar with the Java case when objects have no specific identifiers.

In Amazon.com, most objects did have an identifier, but some objects were only accessible using XPath. The syntax of XPath caused a small problem as it uses the special reserved characters of the action names used in the TEMA models. The problem was solved by moving the XPath-clauses from the action names to the test data.

V. Conclusions

The tool integration we developed worked in testing of a Java Swing application. The implementation required one new library for Robot Framework and modifications to adapter scripts. However, we think that the amount of implementation work was rather small when compared to the benefits of the integration. The OnlineLibrary makes online model-based testing possible with Robot Framework and provides an access to all testing libraries of Robot Framework for users of the TEMA toolset.

While OnlineLibrary makes it possible to use all the libraries of Robot Framework, more investigation is needed in the future to verify that the libraries work well with the TEMA toolset. In principle, there should be no reasons why they would not work, but some implementation details may have unexpected effects that require some adaptation. For example, the Java Swing library causes problems when closing a tested application, as this will shutdown the whole Java Virtual Machine where Robot Framework is also running.

The current integration seems stable and the keyword execution works smoothly. Robot Framework responds fast to the keywords which it can execute successfully (e.g. a test with 22 keyword took 68 seconds). The keywords for GUI actions have a time limit, which is used to allow delays in GUI responses. The GUI-related keywords that fail are often repeatedly performed until the time limit expires. The failing GUI keywords slow down the keyword execution, but the time limit can be optimized to fit specific situations.

More cases studies involving other libraries of Robot Framework, including those for non-GUI testing, remain as future work.

Acknowledgment

Partial funding from Tekes, Nokia, Ixonos, Symbio, Cybercom Plenware, F-Secure, Qentinel, Prove Expertise, as well as the Academy of Finland (grant number 121012), is gratefully acknowledged. We appreciate help from Lotta Liikkanen, who conducted the Selenium Library case study, and the rest of the TEMA team. Anonymous reviewers provided helpful comments.

References

[1] M. Fewster and D. Graham, *Software Test Automation: Effective use of test execution tools*. Addison–Wesley, 1999.

[2] H. Buwalda, "Action figures," STQE Magazine, March/April 2003, pp. 42–47.

[3] I. K. El-Far and J. A. Whittaker, *Model-Based Software Testing* in J. J. Marciniak (ed.), *Encyclopedia of Software Engineering*. Wiley, 2002.

[4] M. Utting and B. Legeard, *Practical Model-Based Testing – A Tools Approach*. Morgan Kaufmann, 2007.

[5] Robot Framework homepage, http://code.google.com/p/robotframework/. Cited Mar. 2011.

[6] TEMA Toolset homepage, http://tema.cs.tut.fi. Cited Mar. 2011.

[7] A. Jääskeläinen, M. Katara, A. Kervinen, M. Maunumaa, T. Pääkkönen, T. Takala, and H. Virtanen, "Automatic GUI test generation for smart phone applications - an evaluation," in *Proc. of the Software Engineering in Practice track of the 31st International Conference on Software Engineering (ICSE 2009)*. IEEE Computer Society, 2009, pp. 112–122, companion volume.

[8] A. Jääskeläinen, "A domain-specific tool for creation and management of test models," Master's thesis, Tampere University of Technology, Jan. 2008.

[9] M. Mokko, "Combatibility of open sourced model-based test automation for software testing (in Finnish)," Master's thesis, University of Oulu, 2010.

[10] A. Jääskeläinen, T. Takala, and M. Katara, "Model-based GUI testing of smartphone applications: Case S60 and Linux," in *Model-Based Testing for Embedded Systems*. CRC Press, to appear.

[11] Abbot Java GUI Test Framework, http://abbot.sourceforge.net/. Cited Mar. 2011.

[12] Jemmy homepage, https://jemmy.dev.java.net/. Cited Mar. 2011.

[13] Swing library keywords, http://robotframework-swinglibrary.googlecode.com/svn/tags/swinglibrary-1.1/doc/swinglibrary-1.1-doc.html. Cited Mar. 2011.

[14] Swing Explorer homepage, https://swingexplorer.dev.java.net/. Cited Mar. 2011.

[15] jEdit homepage, http://www.jedit.org/. Cited Mar. 2011.

[16] Selenium homepage, http://seleniumhq.org/. Cited Mar. 2011.