Pattern Based GUI Testing Modeling Environment

Tiago Monteiro
Departamento de Engenharia Informática
Faculdade de Engenharia, Universidade do Porto
Porto, Portugal
tiago.silva.monteiro@fe.up.pt

Ana C. R. Paiva
Departamento de Engenharia Informática
Faculdade de Engenharia, Universidade do Porto
Porto, Portugal
apaiva@fe.up.pt

***Abstract*—This paper presents a modeling environment (ME) developed for a Domain Specific Language (PARADIGM) which aims to support the construction of models to be used in the context of Model Based GUI Testing (MBGT). It starts by briefly presenting PARADIGM which aims to increase the level of abstraction of the models and promote reuse in order to diminish the effort in building models for MBGT. Afterwards, it describes the architecture of the ME, how the constraints of the language are enforced within the ME to ensure the consistency of the models built, the test case configuration of the model elements, the test case generation algorithm and how the ME can be extended/adapted to include additional features.**

Keywords*— *GUI modeling; DSL; Model based testing; GUI testing.

I. INTRODUCTION

Contrarily to general purpose languages, such as C and Java, a domain-specific language (DSL) is “tailored to a specific application domain” [1].

This paper presents a modeling environment for a DSL (PARADIGM) tailored to the context of Model Based GUI Testing (MBGT). The main goal of this language is to increase the level of abstraction of the GUI models, promote reuse and reduce the effort in building models to MBGT [2]. There are other languages for the same context, such as EFG [3] and VAN4GUIM [4], but we believe they do not foment the same level of reuse.

This paper is structured as follows: section II presents briefly the PARADIGM language; section III presents the analysis performed to choose a framework to support the development of the modeling environment; section IV describes the modeling environment functionalities and how they can be extended; finally, section V presents conclusions and future work.

II. PARADIGM LANGUAGE

The PARADIGM language (Fig. 1) is comprised by elements and connectors. The elements can be *Init* (to mark the beginning of a GUI model), *End* (to mark the termination of a GUI model), *Structural* (to allow structure the GUI model in different levels of abstraction) and *Behavioral* (to describe the behavior to test).

As models become larger, coping with their growing complexity forces the use of structuring techniques such as different hierarchical levels that allow use one entire model *A* inside another model *B* abstracting the details of *A* when within *B*. It is like what happens in programming languages, such as C and Java, with constructs such as modules. *Form* is a structural element that may be used for that purpose. A *Form* is a model (or sub-model) with an *Init* and an *End* elements.

Group is also a structural element but it does not have *Init* and *End* and, moreover, it has a Boolean attribute named *AnyOrder* that, when true, means that all elements inside the *Group* may be executed in an arbitrary order.

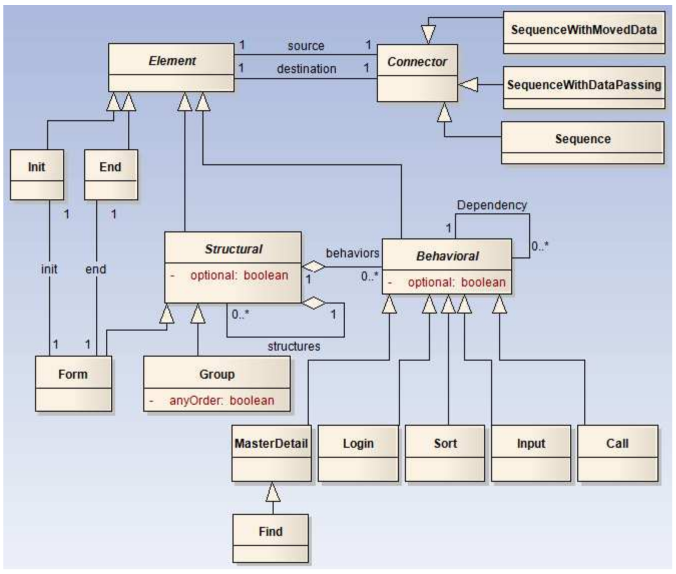


Fig. 1. PARADIGM language model

GUIs have recurrent behavior that is common and produce similar results (called UI patterns). A pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice [5]. We took the notion of UI patterns into the context of GUI testing which led us to the concept of UI Test Patterns. Those test patterns are the *Behavioral* elements within PARADIGM which define test strategies for testing UI patterns. Test strategies are a set of configurations (to test the associated behavior) and have to be mapped to real controls present in the GUI to be tested to allow the subsequent execution of the generated test cases.

Finally, notice that, except for *Init* and *End*, all the other elements have a Boolean attribute called *Optional* that, when true, means that it is possible to bypass the corresponding behavior in order to achieve a specific goal.

The elements can be combined through connectors. There are three different kinds of connectors: *Sequence*, *SequenceWithDataPassing* and *SequenceWithMovedData*.

Two elements connected by *Sequence* (A *Sequence* B) means that interaction (with the modeled GUI) according to element *B* can only be performed after interaction according to *A* finishes. Connectors *SequenceWithDataPassing* and *SequenceWithMovedData* are similar to *Sequence*, but, additionally, the first connector also means that element *B* receives data from element *A* and the second connector means that element *A* transfers data to element *B* (in this particular case, element *A* loses data and element *B* gets data).

Besides connectors, there is also a relation called *Dependency* to model the case when the destination element properties depend on the properties of a set of source elements.

The GUI models constructed with PARADIGM must follow some rules in order to be considered well-formed. These rules are imposed by the modeling environment developed and will be listed in section IV.B.

III. Framework Analysis

The PARADIGM modeling environment aims to provide support for constructing well-formed GUI models, configure those models with test input data, generate test cases from those models and execute them on a real GUI.

Nowadays, there are several frameworks that easy the process to build a DSL and corresponding modeling environment. The choice of which framework to use in the present work was performed by evaluating several features of a set of available frameworks. Those characteristics were: possibility to extend or create new functionalities; possibility to define properties to configure language elements; possibility to define rules to ensure model integrity; possibility to integrate the tool with other development environments; possibility to save the created models in XML and being an active project. The frameworks evaluated according to these characteristics were: Eclipse Graphical Modeling Framework [6][7]; StarUML [8], Open Modelsphere [9][10] and ArgoUML [11]. This analysis was conducted considering the documentation freely available and hands-on experiments.

StartUML was discarded because, as far as we know, it is a discontinuous project. ArgoUML seems to be in an incipient state (version 0.34) and may be subject of updates that can cause problems for those developing on top of it. Open Modelsphere forces to change the core of the application to reach the objective of creating a new notation which does not seem a good solution for our purposes.

For these reasons, Eclipse Graphical Modeling Framework seemed to be the best option. This framework allows for easy creation of a fully-featured modeling environment, is an active project and presents a rather active community of support.

A. Eclipse Graphical Modeling Framework

Launched in 2006, Eclipse Graphical Modeling Framework (GMF) [6] (Fig. 2) is used to create graphical editors for modeling languages.

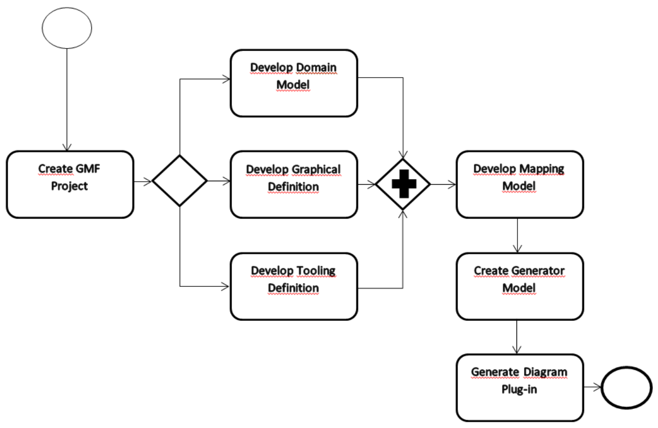


Fig. 2. DSL creation process in Eclipse Graphical Modeling Framework

This process of constructing a DSL based on GMF comprises the definition of the Domain Model (to define the elements, connections and their properties); the definition of the graphical representation of each element of the DSL (within the Graphical Definition Model); and the definition of the menus, the actions and toolbars within the Tooling Definition. Afterwards, one needs to develop the Mapping Model, whose purpose is to specify the relationships among the elements in each of the previous models, linking each domain element with a graphical representation and proper tooling. After setting some generation properties, it is possible to build the modeling environment as a plug-in for Eclipse.

IV. PARADIGM Modeling Environment

The PARADIGM Modeling Environment (Fig. 3) allows the creation of models with all elements and connectors present in the PARADIGM language. On the right-side is the palette with all the elements and connectors of the language. On the left is the modeling area and below the properties tab.

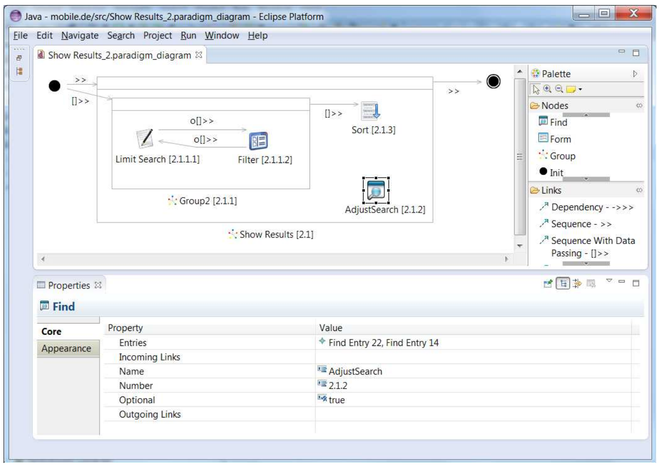
A. Elements and connectors

As stated before, the current version of PARADIGM Modeling Environment presents all elements and connectors described in the PARADIGM language. However, in the future, it may be necessary to add elements (or connectors) to extend/adapt the language. For instance, imagine that we want to add the *Call* element to the language. For that purpose, the programmer has to edit the models referred in III.A.

1. **Create the new element in the Domain model**, name it (*Call*) and state that its *ESuper Type* is Behavioral (because *Call* descends from *Behavioral*). Add attributes to hold the configurations (*entries*) and to hold the mapping between the element of the model and the controls in the GUI (*callMapping*) which implement the behavior described. This mapping is useful during test case execution [12].

2. **Create a graphical design for the added element** in the Graphical Definition Model. If the graphical representation of an element is a figure, create a rectangle in the graphical definition model; and add a class extending *ImageFigure* in the plugin figures with the path to the image in the constructor (for example, “images/call.png”).
3. **Create the tool for the added element** in the modeling environment. Add a new *Creation Tool* with the name of the element (in this example *Call*) in the Tooling Definition Model.
4. **Create a new *Top Node Reference*** (or a *Link Mapping* in case of connector) in the Mapping Model and specify the relationship among domain, graphical and tooling elements.

After this, one just needs to generate the modeling environment code all over again.



Property	Value
Entries	Find Entry 22, Find Entry 14
Incoming Links	
Name	AdjustSearch
Number	2.1.2
Optional	true
Outgoing Links	

Fig. 3. Modeling Environment

B. Rules

The environment enforces a set of rules in order to guarantee that the models are well-formed. These rules are implemented in OCL [13] as follows:

Each model (or *Form*) can have only one *Init* and one *End*

```
not (self.nodes->select(oclIsTypeOf(Init))->size() > 1)
not (self.nodes->select(oclIsTypeOf(End))->size() > 1)
```

Init cannot be the destination of a connector and *End* cannot be the source of a connector

```
not self.destination.oclIsTypeOf(Init)
not self.source.oclIsTypeOf(End)
```

Init and *End* cannot be directly connected by a connector

```
self.source.oclIsTypeOf(Init) implies
   not self.destination.oclIsTypeOf(End)
```

An element cannot be connected to itself

```
self.source <> self.destination
```

Two elements cannot be connected (twice) by two different connectors of the same type

```
self.relations->forAll(c1, c2 |
   c1<>c2 and c1.source = c2.source implies
        c1.destination<>c2.destination
```

One element cannot belong to two different group elements

```
Group.allInstances()->forAll(g1, g2 | g1<>g2 implies
   (g1.nodes->intersection(g2.nodes) = g1.nodes) or
   (g1.nodes->intersection(g2.nodes) = g2.nodes) or
   ((g1.nodes->intersection(g2.nodes))->size() = 0))
```

The modeling environment allows adding new rules. For that, it is necessary to:

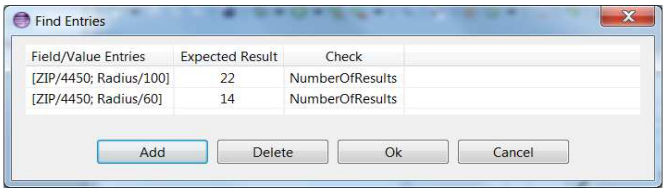
1. Within the Mapping Model, add a new *Audit Rule* in the *Audit Container*, name it, define the message to show when the rule is broken, set its severity (“Info”, “Warning” or “Error”) and define live mode of the rule (if the rule is verified while modeling; or if the rule is only verified when the user specifically asks for model validation).
2. Inside the *Audit Rule*, add a *Constraint* with the rule implementation (in OCL or Java), and add the context of this rule (it may be a domain element, or domain attribute).

C. Test Case Configuration

After constructing a GUI model describing the functionality to test, it is possible to generate test cases. For that, the tester needs configure each behavioral element of the model in order to provide test data and specify the checks to be performed during test execution.

To demonstrate this functionality, consider the model in Fig. 3 and particularly node “AdjustSearch [2.1.2]”. This is a *Find* element that will test if the GUI is capable of finding the correct answer for a given input.

ME allows the tester to define a particular input for a particular field, the expected result and also the check to perform. Fig. 4 sums up the test configuration for this element. In this case, the test will check if the cardinality of the results’ set obtained by the search is 22 (in the first line) and 14 (in the second line).



Field/Value Entries	Expected Result	Check
[ZIP/4450; Radius/100]	22	NumberOfResults
[ZIP/4450; Radius/60]	14	NumberOfResults

Fig. 4. “Find Entries” dialog

When a new element is added to the supported language (in this case PARADIGM), the programmer has also to define a test configuration. For that, he has to use the ability of Eclipse to be extended by components (plug-ins). To build such plug-in, the programmer must:

1. Extend the class *CorePlugin* and implement the method run.
2. Create a new class that extends *PropertyDescriptor* and call the necessary dialog to interact with for the test configuration purposes.
3. Create a JAR file with the project created and the project *pluginLoader* and place it in the *lib* folder in the *Paradigm.edit.ui* project.

D. Path Generation

The test case generation is performed in two steps. Firstly, test paths are generated by a recursive algorithm. In general, the algorithm calculates all the possible paths traversing the model starting in element *Init* until reaching the *End* element, both mandatory elements within a model. Secondly, the concrete test cases are generated from those paths. An approach to do so can be found in [14].

Nevertheless, it is important to highlight the special treatment of some elements during path generation. When a specific path has an optional element (the *Optional* attribute is set to true) the generator introduces paths without the presence of that optional element. For instance, consider a path A-B-C in which B is optional. In this case, the path generator will add the path A-C to the set of possible paths. *Forms* and *Groups* also deserve special treatment. In case of a model structured into levels of abstraction (with *Forms* within it), the generator calculates all test paths of that *Form*, stores them and continues generating the test paths for the model. Afterwards, the paths generated for a *Form* are inserted in the test paths generated for the model substituting the *Form* element by such paths. In case of Groups, it generates all paths for the Group generating also all possible combinations of these paths if the *AnyOrder* attribute is true for that Group. As what happens with *Forms*, it stores these paths, so they can be later introduced in their proper place.

After test paths are generated, the concrete test cases are built replacing each behavioral element within each path by its specific test strategy. If wanted, it is also possible to add other test case generation algorithms to the modeling environment. To use a different algorithm, the programmer must create a new plug-in project. In the XML file of the project, he has to create a new command, a new command handler and a context menu entry. Afterwards, implement the class referenced in the command handler and as an extension of *AbstractHandler*. The execute method of such class must implement the desired test case generation algorithm.

E. Test case execution

Right now, the ME just provides an execution module for web applications. However, it allows the addition of new modules to execute the test cases generated over different platforms, such as, desktop. The execution module works on top of Selenium [15] and Sikuli [16]. That allows us to identify GUI objects either by their properties or by their bitmap. During configuration time, the tester needs to point out the objects (to capture their properties and their bitmap) in order to act upon them during test case execution time.

V. CONCLUSIONS

This paper presented a modeling environment for the PARADIGM language to be used in the context of MBGT. A demonstration video can be found in *www.fe.up.pt/~apaiva/tools/paradigmME.wmv*. The goal of this ME is to support modeling GUIs, test case configuration, test case generation and test case execution. In addition, the ME provides extension points that allow adding new elements to the supported language and adding new test case generation algorithms.

In spite of some experiment realized in laboratory that make us confident in the usability and usefulness of the ME, we intent to realize a set of experiments in industry to evaluate the real benefit of the overall approach to increase the quality of GUIs. The result of these experiments will be useful, not only to improve the ME functionalities, but also to improve the PARADIGM language providing, for instance, new behavioral elements that contribute to increase the level of abstraction of the constructed models.

ACKNOWLEDGMENT

This work is financed by the ERDF – European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT – Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-020554.

REFERENCES

[1] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Computing Surveys*, vol. 37, no. 4, pp. 316–344, Dec. 2005.

[2] M. Cunha, A. C. R. Paiva, H. S. Ferreira, and R. Abreu, "PETTool: A pattern-based GUI testing tool," in *Software Technology and Engineering (ICSTE), 2010 2nd International Conference on*, 2010, vol. 1, pp. V1–202.

[3] A. M. Memon, M. Lou Soffa, and M. E. Pollack, "Coverage criteria for GUI testing," *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering ESECFSE9*, vol. 26, no. 5, p. 256, 2001.

[4] R. M. L. M. Moreira and A. C. R. Paiva, "Visual Abstract Notation for Gui Modelling and Testing - VAN4GUIM," in *ICSOFT (SE/MUSE/GSDCA)*, 2008, pp. 104–111.

[5] C. Alexander, S. Ishikawa, and M. Silverstein, *A Pattern Language: Towns, Buildings, Construction*, vol. 2, no. 0. Oxford University Press, 1977, p. 1171.

[6] R. C. Gronback, *ECLIPSE MODELING PROJECT - A Domain-Specific Language Toolkit*. Addison-Wesley, 2009.

[7] A. Shatalin and A. Tikhomirov, "Graphical modeling framework architecture overview," in *Eclipse Modeling Symposium*, 2006.

[8] M. Lee, H. Kim, J. Kim, and J. Lee, *StarUML 5 . 0 - Developer Guide*. 2005.

[9] Grandite, "Open Modelsphere - User Guide," 2009. [Online]. Available: http://www.modelsphere.org/help/User_Guide.html.

[10] Grandite, *Open ModelSphere 3.0 - Developer Guide*, no. September. 2008.

[11] A. Ramirez, L. Tolke, M. Wulp, J. Benett, K. Odutola, A. Rueckert, and P. Vanpeperstraete, *ArgoUML User Manual A tutorial and reference description*. 2011.

[12] A. R. Paiva, J. P. Faria, and R. A. M. Vidal, "Specification-Based Testing of User Interfaces," in *Interactive Systems. Design, Specification, and Verification*, vol. 2844, J. Jorge, N. Jardim Nunes, and J. e Cunha, Eds. Springer Berlin Heidelberg, 2003, pp. 139–153.

[13] "OCL specification," 2006. [Online]. Available: http://www.omg.org/cgi-bin/doc?formal/06-05-01.

[14] C. D. Nguyen, A. Marchetto, and P. Tonella, "Combining model-based and combinatorial testing for effective test case generation," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, 2012, pp. 100–110.

[15] "Selenium." [Online]. Available: http://seleniumhq.org/.

[16] "Sikuli." [Online]. Available: http://www.sikuli.org/.