A GUI Modeling DSL for Pattern-Based GUI Testing
PARADIGM

Rodrigo M. L. M. Moreira[1] and Ana C. R. Paiva[1]
[1]*INESC TEC and Department of Informatics Engineering*
Faculty of Engineering of the University of Porto, Porto, Portugal
{pro08007, apaiva}@fe.up.pt

Keywords: GUI Modeling, GUI Testing, Domain-Specific Languages, PARADIGM, Pattern-Based GUI Testing, Graphical Modeling DSL.

Abstract: Today's software feature user interface (UI) patterns. Those patterns describe generic solutions for common recurrent problems. However, to the best of our knowledge, there is no specific testing methodology that is particularly suited for testing those patterns providing generic testing solutions that can be reused after minor configurations in order to test slightly different implementations. Pattern-Based Graphical User Interface Testing (PBGT) is a recent methodology that aims at systematizing and automating the GUI testing process, by sampling the input space using "UI Test Patterns" that express generic solutions to test common recurrent GUI's behaviour. This paper describes the development process of PARADIGM, a domain specific language (DSL) to be used in the context of PBGT and empirically evaluates PARADIGM to assess its diminished modeling efforts, usefulness, graphical power, and acceptability.

1 INTRODUCTION

Nowadays the majority of software applications feature a Graphical User Interface (GUI). They are the system's entry point, upon which users communicate with the underlying software. GUIs have become quite popular and represent a big role on the success of the software. GUIs stand as the first contact between the system and its users. Users' opinion is extremely important as it is their decision to either use such software or not. If the user finds errors in the GUI, he will lose trust on the software. Therefore, it becomes important to test GUIs for their functional correctness.

GUIs often feature UI Patterns (Moreira et al., 2013) [1]. UI Patterns are recurring solutions to solve common GUI design problems. Until quite recently, no approach existed to test UI Patterns for their functional correctness. Our prior work (Cunha et al., 2010; Moreira et al., 2013) introduces a new methodology called Pattern-Based GUI Testing method (PBGT), that aims at systematizing and automating the GUI testing process. In addition, PBGT develops the notion of UI Test Pattern, which provides a reusable and configurable test strategy, in order to test a GUI that was implemented using a set of UI Patterns.

This paper describes PARADIGM, a DSL to be used in the context of PBGT. It describes the activities followed during its development, according to best practices and a set of guidelines, and presents an empirical evaluation to assess its diminished modeling effort, usefulness, graphical power, and acceptability. Examples of the PBGT approach and the usage of PARADIGM to model real applications, can be found in (Moreira et al., 2013).

Modeling GUIs for testing offers a number of challenges. The need to express specific domain features, along with the desire to express them using paradigms familiar to the domain experts, leads to the demand for domain specific languages. However, DSLs are, in general, difficult to design, implement and maintain. In addition, developing DSLs from scratch represent a laborous manual task. Moreover, when/if a DSL grows in complexity and size, its design becomes more error-prone.

Alloy (Jackson, 2011) is an intuitive formal modeling language, with a syntax that is simple easy to read and write. It is supported by the Alloy Analyzer tool, a friendly SAT (satisfability) based tool that enables automatic model V&V (verification and validation). In the context of our work, Alloy is used to support the development of a DSL from scratch. From

[1]http://ui-patterns.com/, http://pttrns.com/

the Alloy model of the DSL, it is possible to generate instances (that should represent possible valid GUI test models) and analyze those instances in order to define and tune language constraints. This is an iterative process that ends when the constraints necessary to ensure the construction of well-formed PARADIGM test models are found.

The main contributions of this current paper include:

– a detailed description concerning the process (according to best practices) that was followed to create a new DSL (PARADIGM) for the PBGT domain;

– a novel iterative approach using Alloy in DSL development;

– an empirical case study to assess the effort required to model GUIs with PARADIGM;

– evaluation of PARADIGM usage at an industrial environment.

2 BACKGROUND

This section presents existing GUI Modeling approaches and tools/frameworks that support the development of graphical editors.

2.1 GUI Modeling

Spec# (Microsoft, 2013) is a model-based specification language developed by Microsoft Research. It extends the existing object-oriented .NET programming language C# with specification constructs like pre-conditions, post-conditions, invariants, and non-null types. GUIs can be modelled with Spec# describing actions performed by the end-user and the expected outcome of each user action (Paiva et al., 2003). Hence, GUI Spec# models are the input to Spec Explorer (Microsoft, 2012) with GUI Mapping Tool (Paiva et al., 2005), which is a model-based testing tool, that generates and executes test cases automatically. However, the creation of GUI models with Spec# is very demanding and time consuming.

The Visual Abstract Notation for GUI Modeling and Testing (VAN4GUIM) (Moreira and Paiva, 2008) is an approach that tries to diminish this effort, by hiding formalism details used in Model-Based Testing (MBT) approaches. Furthermore, it aims at promoting the use of MBT in industrial environments by providing a graphical front-end to assist GUI modeling, and thus making the crafting of models more appealing to testers rather than the textual notation. This visual notation is based on Canonical Abstract Prototyping (Constantine, 2003) and ConcurTaskTrees - CTT (Paternò et al., 1997). It consists of five UML profiles: *Containers*, who are responsible to hold/group UI elements; *User Actions*, that are tools to act upon containers; *Hybrids* that result in combination between elements to model user actions who act upon specific containers; *CTT connectors* to describe relationship among elements and; *Window Manager* to describe windows behavior. GUIs modeled with VAN4GUIM are then translated automatically to Spec#, according to a set of translation rules. However, the model does not feature all GUIs behavior. Additional behavior needs to be included manually in the generated Spec#. After the Spec# model is completed it will be used as input to Spec Explorer with GUI Mapping Tool.

Event Flow Graphs (EFG) are a popular approach that allow to create a GUI model by capturing the flow of events, featuring all possible event interactions in the UI (Memon et al., 2001). GUI models are generated via a tool called GUI Ripping Tool. However, to the best of our knowledge, no tool exists to model an EFG by hand.

2.2 Tools for Language Implementation

Numerous tools have the capability to generate graphical editors from a metamodel of a DSL language. Such tools differ on the: (i) implementation to attain concrete graphical syntaxes; (ii) implementation complexity; and (iii) maintenance efforts.

Microsoft Domain-Specific Language Tools (Cook et al., 2007) allow to create a representation of the model of a language, to specify their relations, to describe the semantics of the language and to define the DSL graphical representation. Furthermore, Microsoft DSL Tools offer a solid and seamless implementation, nonetheless depending on the implementation goals, the fact of being platform reliant, can be considered as a restriction.

The Eclipse platform contains a set of modeling frameworks, such as Eclipse Modeling Framework (EMF) (Steinberg et al., 2009), Graphical Editing Framework (GEF) (Rubel et al., 2011) and Graphical Modeling Framework (GMF) (Gronback and Boldt, 2013). EMF is a modeling framework that facilitates building tools to support modeling languages. GEF provides the graphical support needed for building a model/language editor on the top of the EMF framework. In addition, GMF is a more advanced and generative framework for developing graphical editors for providing support for DSL development, by leveraging EMF and GEF. These frameworks are popular, straightforward to use and maintain. Moreover, they have a strong community support.

StarUML (StarUML, 2005) is an open source project having in mind flexibility, extensibility, and fast development. StarUML can be extended to provide further functionality over the tool by the development of new modules. The tool acts as modeling platform to provide functionality for various platform technologies and external tools. This modeling software offers great extensibility, but it requires a deep knowledge over the core extension mechanisms of the tool. Furthermore, this tool only runs on win32 systems and its development has stagnated.

Open ModelSphere (Grandite, 2008) is a platform independent modeling tool. This tool is supported by an experienced community and be freely modified as desired. The downside is the complexity to cope with the addition of new language elements. In addition, it also requires a deep knowledge over the ModelSphere layered model.

3 DSL IMPLEMENTATION

In general, DSLs are high-level languages exclusively tailored to specific tasks. They are designed to make their users effective in a specific domain. They are important because they represent a more natural, robust, precise, and maintainable way of capturing the essence of a given problem, rather than merely being expressed in a general-purpose language.

The PBGT methodology requires a GUI model to be crafted specifically for this domain. Thus, a language is required to assist in the creation of models particularly tailored for PBGT. With the latter in mind, several DSLs for GUI modeling have been evaluated, and have been described in the previous section. However, to the best of our knowledge, there is no DSL built from the right beginning of its development with reusability concerns, providing UI Test Patterns that can be adapted for testing different implementations after a small configuration step. This language promotes reusability either by reusing existing elements or extending them to be reused by others. In addition, our DSL allows building a model describing the test goals instead of describing the expected behavior.

PARADIGM is a DSL that has been created, from scratch, for PBGT. The main goals of this language are:

– provide higher level of abstraction models;

– provide generic test strategies for testing GUIs;

– provide language flexibility and promote reuse;

– provide extension mechanisms to cope with possible new UI Patterns trends;

– simple to use and maintain;

– and reduce the effort in building models.

PARADIGM development was based on a "tailored DSL engineering process" (Strembeck and Zdun, 2009), as well as guidance from Martin Fowler's book "Domain Specific Languages" (Fowler, 2010) and "Design Guidelines for Domain Specific Languages" (Karsai et al., 2009). These references define a set of best practices to follow during the development of a DSL.

A DSL development is an iterative process. This process is comprised by 5 major activities (Strembeck and Zdun, 2009): (1) Definition of the *Language Core Model* with the language elements and their relations; (2) Add constraints (if needed) to restrict the language funcionalities (*Language Restrictions*); (3) Specify the *Syntax* of the language to describe how the elements are represented; (4) Describe the dynamic *Behavior* of the elements of the language, i.e., how such elements perform and interact; (5) Integrate the DSL within a tool to support the construction of models (*Platform Integration*).

3.1 Language Core Model

A language core model captures all relevant domain abstractions and specifies relations among them. Abstractions refer to concepts within the context of PBGT. In PARADIGM, these abstractions are behavioral elements (UI Test Patterns), structural elements and language connectors.

A DSL model written in UML (Figure 1) can be defined as a metamodel or a UML profile. We opted for a metamodel approach since (Brucker and Doser, 2007): (i) the defined model does not need to be transferred to other domains; (ii) we want to have a more abstract syntax; (iii) it provides easier semantic definition; (iv) and the domain is well-defined due to a unique precise set of concepts (UI Test Patterns, mainly).

3.1.1 Elements

An *Element* is an abstract entity that represents the concepts within PBGT domain. A model written in PARADIGM starts with the *Init* element and finishes with the *End*. Both *Init* and *End* are specializations of the abstract entity *Element*. Models can be structured in different levels of abstraction. For instance, as models grow it is possible to use structuring techniques, to handle different hierarchical levels,

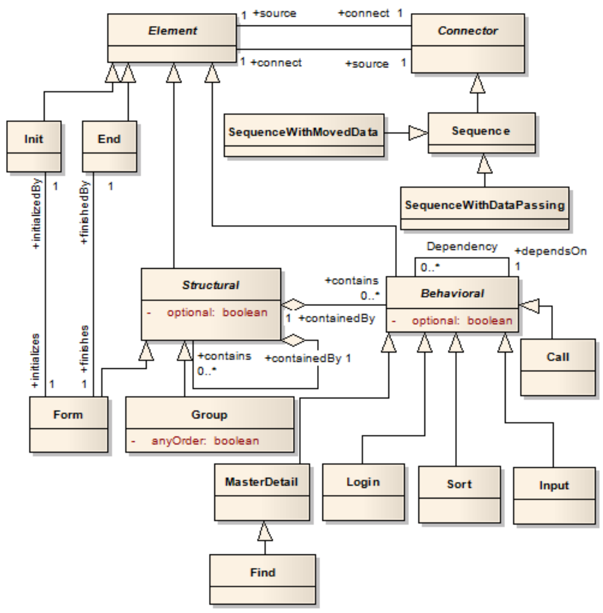


Figure 1: PARADIGM Language Metamodel.

so that a *model A* can "live" inside a *model B*. *Structural Form* elements were created specifically for this purpose. A *Form* (*Structural* element) embodies a model (or sub-model), with an *Init* and *End* elements. *Groups* are also structural elements but they are used to gather elements that may be executed in any order. *Behavioral* elements represent the UI Test Patterns that define strategies for testing the UI Patterns.

3.1.2 Connectors

ConcurTaskTrees (CTT) (Paternò et al., 1997) are a popular, expressive and powerful graphical notation for task representation. It defines a set of temporal operators to combine tasks. The connectors within the PARADIGM language are inspired by CTT.

PARADIGM defines three connectors: "*Sequence*"; "*SequenceWithDataPassing*"; and "*SequenceWithMovedData*". The "*Sequence*" connector indicates that the target element cannot start before the source element has completed. The "*SequenceWithDataPassing*" connector has the same behavior as "*Sequence*" and, additionally, indicates that the target element receives data from the source element. "*SequenceWithMovedData*" has a similar meaning to the "*SequenceWithDataPassing*" connector, however, the source element transfers data to the target, so the source loses the data that was transferred. In addition, there is another kind of relation among elements – "*Dependency*" – indicating that the target element depends on the properties of a set of source elements, for instance, when it is the result of a calculation.

3.1.3 Language Constraints

The model of the PARADIGM's language contains additional constraints that cannot be expressed directly in the UML language model. During DSL development some initials thoughts regarding language constraints tend to evolve in time and need to be enhanced as the language model increases and progresses. To assist in defining the constraints to be imposed on the language, we have decided to follow an iterative development process by building the language model in Alloy as will be explained in the sequel.

3.1.4 PARADIGM Model and Restrictions in Alloy

Alloy is a lightweight formal language that supports structural and behavioral modeling. It is suitable for early stages of software development, which allows discovering the correct software abstractions. Alloy is based on relational logic that combines the quantifiers of first order logic with the operators of the relational calculus. In addition, Alloy allows expressing complex structural and behavioral constraints. One of the advantages of Alloy is its simple but powerful notation.

An Alloy model contains a set of imports, signatures, relations, facts, predicates, functions, assertions and commands. A signature introduces a typed set of atoms and may have fields. Facts are constraints on relations that always hold. Predicates define reusable constraints. Functions define reusable expressions. Commands allow performing "check" and "run" analysis. Where "run" checks model consistency by looking for valid instances and; "check" verifies the assertion looking for counter examples.

The PARADIGM UML metamodel (Figure 1) was translated to Alloy in order to generate instances (that should represent possible valid GUI test models) and, afterwards, analyze them to think about the properties that such models should have and tune the language constraints. This is an iterative process that ends when the instances obtained from the Alloy model correspond to valid well-formed PARADIGM models. This is when the correct set of constraints is found.

As such, the starting point of this process was to define the structure of PARADIGM models as they may be structured in different levels of abstraction. From this Alloy model (Figure 2), it is possible to generate instances (e.g., Figure 3) (with Alloy Analyzer Tool) for a scope of at most 11 elements of each signature but exactly 2 Forms and 1 Group. We have defined an extra signature (comparing with the

UML metamodel), *Model*, a specialization of *Form* to describe the upper level of the overall model and for simplifying the specification. Figure 3 shows a GUI test model with 2 hierarchical levels (*Model* and *Form*). The *Model* has one *Init* (Init1), one *End* (End1), one *Behavior* or *UI Test Pattern* (Behavior1), a *Form* and a *Group*. The *Form* has one *Init* (Init0), one *End* (End0) and a *UI Test Pattern* (Behavior0). The Group has two *UI Test Patterns* (Behavior2 and Behavior3). This instance allowed us to check if the model specification was correct (by featuring the designed restrictions).

```
open util/boolean

abstract sig Element{}
sig Init extends Element{}
sig End extends Element{}
abstract sig Behaviour extends Element{}
abstract sig Structural extends Element{
 innerStructs: set Structural,
 innerBehaviour: some Behaviour
}
sig Group extends Structural {
}{#innerBehaviour> 1}
sig Form extends Structural {
 init: one Init,
 end : one End,
}
one sig Model extends Form {}

fun Parent[e:Element] : Structural{
 innerStructs.e + innerBehaviour.e + init.e + end.e
}

fact {
 all e:Element-Model | one Parent[e]
 no Parent[Model]
     -- every Form and Group should be inside
     -- (achievable from) Model
 all s:Structural-Model | s in Model.^innerStructs
}
```

Figure 2: Alloy model featuring PARADIGM hierarchical levels.

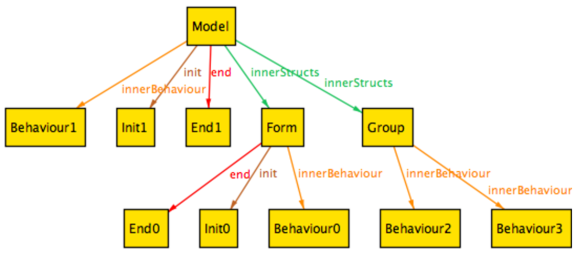


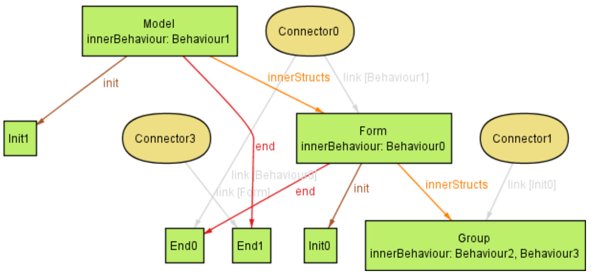
Figure 3: Instance of the Alloy model (from Figure 2) regarding PARADIGM hierarchical levels.

Afterwards, we evolved the model in order to include the description of the language connectors and constraints (Figure 4) that ensure that each connector links only two elements that are within the same Structural element (Model, Form and Group). Before each statement there is a description of its meaning starting with "– –". An example of an instance generated from this model is illustrated in Figure 5.

The refined language constraints for PARADIGM are:

```
fact {
    -- there is a path from all elements inside a form until
    -- the End element of that form
  all f:Form | f.end+f.innerBehaviour+f.innerStructs
       in f.init.^(Connector.link)
    -- there is a path from the Init element inside a
    -- Form to all elements of that form
   all f:Form | f.init+f.innerBehaviour+f.innerStructs
       in f.end.^(~(Connector.link))
}
sig Connector{
 link: Element  -> lone Element
} {  -- no connectors from one element to the same element
    no link.Element & link[Element]
     -- the source of one connector cannot be End
    no link.Element & End
    -- the destination of a connector cannot be Init
    no link[Element] & Init
    -- there are no links to/from Model
    no (link[Element]+link.Element) & Model
    -- no link from Init directly to End
    lone (link[Element] + link.Element) & (Init + End)
    -- no connector between elements within different Structural elements
    no e: link[Element] | Parent[link.e] != Parent[e]
   }
```

Figure 4: Alloy model featuring PARADIGM connectors.



Connector.link

```
{Behaviour$0->End$0,
Behaviour$1->Form$0,
Form$0->Behaviour$1,
Form$0->End$1,
Group$0->Behaviour$0,
Init$0->Group$0,
Init$1->Behaviour$1}
```

Figure 5: Instance of the Alloy model (from Figure 4) regarding PARADIGM connectors.

LC1: A *Connector* cannot connect an element to itself;

LC2: A *Connector* cannot have *Init* as destination neither *End* as source;

LC3: An *Init* element cannot connect directly to an *End* element;

LC4: Two elements cannot be connected more than once by connectors of the same type;

LC5: Two Elements can only be connected if they belong to the same Structural Element (Model; Form; Group);

LC6: Elements inside a *Form* (but not inside *Groups* of that *Form*) cannot be loose, i.e., for all elements within a *Form*, there is at least one path from the *Init* to the *End* that traverses that element.

3.2 Behavior Specification

The DSL behavior specification, also noted as dynamic semantics, stands as a part of the language model and defines the behavior concerning the usage of a given DSL language element. PARADIGM's behavioral semantics establishes the instructions on how to use the language and defines the interactions among language elements.

To some extent, the DSL language constraints already define the behavior of the PARADIGM language. Nevertheless, these are not sufficient to define the behavioral varieties of the PARADIGM elements. So, it is important to also define the possible configuration of the UI Test Patterns during the modeling phase to allow testing generic behavior over slightly different implementations and with different input data.

3.2.1 UI Test Patterns

UI Patterns represent common functionalities that can be implemented in different ways. Nevertheless, what is important to notice is the common behavior that is reflected in these generic implementations. A UI Test Pattern aims at testing those UI Patterns accross its several slightly different implementations. Master/Detail, Find, Sort, Input and Call are UI Test Patterns.

Definition 1. *UI Test Pattern defines a test strategy which is formally defined by a set of tuples with the form:*

$$< Goal, V, A, C, P >, where$$

Goal *is the id of the test;* ***V*** *is a set of pairs* $\{$*[variable, inputData]*$\}$ *relating test input data with the variables involved in the test;* ***A*** *is the sequence of actions to perform during test case execution;* ***C*** *is the set of possible checks to perform during test case execution, for example, "check if it remains in the same page";* ***P*** *is a Boolean expression (precondition) defining the set of states in which it is possible to execute the test. In another way, Goal is the "name" of the test. A and the variables in V describe "what" to do and "how" to execute the test. C describes the final purpose (or why) the test should be executed. P defines when can be executed.*

The *Goal*, variables in V, A and C are defined by the developer during the implementation phase of the language. During the modeling phase, the tester needs to configure each UI Test Pattern within the model. He has to select the testing Goals (tests) and, for each of those Goals, provide test input data, select the checks to perform, and define the precondition that describes the states where it is possible to execute the corresponding actions. The tester can select the same Goal multiple times for a UI Test Pattern providing different configurations.

Input UI Test Pattern. This UI Test Pattern should be used to test the behavior of input fields for valid and invalid input data.

- **Goals:** "Valid data" (INP_VD) and "Invalid data" (INP_ID);
- **Set of variables:** {input};
- **Checks to perform:** {"message box", "label", "error provider"};
- **Sequence of actions:** [provide *input*].

During configuration, the user has to provide valid input data for INP_VD (and invalid input data for INP_ID), select the checks to perform and define the precondition.

Master/Detail UI Test Pattern. This pattern should be applied to test UI Patterns that feature two areas, master and detail, related in such a way that, when the master changes, the detail changes accordingly. Examples of these UI Patterns can be found in software applications such as iTunes[2] and Finder[3].

- **Goals:** "Change master" (MD);
- **Set of variables:** {master, detail};
- **Checks to perform:** {"detail has value X", "detail does not have value X", "detail is empty"};
- **Sequence of actions:** [select *master*].

During configuration the user has to provide master input data for MD, select the checks to perform and define the precondition.

Login UI Test Pattern. This pattern should be used to test an authentication. The goal is to check if it is possible to authenticate with a valid username/password and check if it is not possible to authenticate with invalid usernames/passwords.

- **Goals:** "Valid login" (LG_VAL) and "Invalid login" (LG_INV);
- **Set of variables:** {username, password};
- **Checks to perform:** {"change page", "pop-up error", "same page"};
- **Sequence of actions:** [provide *username*; provide *password*; press *submit*].

During configuration the user has to provide valid username/password input data for LG_VAL (and invalid username/password for LG_INV), select the checks to perform and define the precondition.

Find UI Test Pattern. This pattern should be used when someone wants to test the result of a search that shows up after a submit action. The goal is to verify that the result of the search is as expected (it finds the right set of values).

[2]http://www.apple.com/itunes/

[3]http://support.apple.com/kb/ht2470

- **Goals:** “Value found” (FND_VF) and “Value not found” (FND_NF);
- **Set of variables:** $\{v_1,...,v_N\}$;
- **Checks to perform:** {“empty set”, “if it has X elements”, “if it does not have element X”, “if the result in line X is Y”};
- **Sequence of actions:** [provide v_1,... , provide v_N].

During configuration, the user has to define the value for N, provide input data for variables $v_1,...,v_N$, select the check to perform and define the precondition.

Sort UI Test Pattern. The Sort UI Test Pattern is used to check if the result (of a sort action) is ordered accordingly to the chosen sort criterion. The idea is to test user interfaces that contain sortable items/elements, such as, tables and lists.

- **Goals:** “ascending” (SRT_ASC) and “descending” (SRT_DESC);
- **Set of variables:** $\{v_1,...,v_N\}$;
- **Checks to perform:** {“element from field X in position Y has value Z”};
- **Sequence of actions:** [provide v_1,... , provide v_N].

During configuration, the user has to define the value for N, provide input data for variables $v_1,...,v_N$, select the check to perform and define the precondition.

Call UI Test Pattern. This UI Test Pattern is used to check the functionality of the corresponding invocation.

- **Goals:** “Action succeeded” (CL_AS) and “Action failed” (CL_AF);
- **Set of variables:** { };
- **Checks to perform:** {“pop-up message”, “stay in the same page”, “change to page X”};
- **Sequence of actions:** [press].

During configuration, the user has only to select the check to perform and define the precondition.

3.3 Syntax

The next step in the DSL development is the definition of its syntax (Figure 6). Hence, we defined the graphical symbols for the language connectors and for the language elements. The reasons for selecting a graphical notation over a textual one, was based on the following aspects: (i) easier to understand; (ii) simpler to navigate and; (iii) intuitive and faster learning for end-users.

All PARADIGM’s elements and connectors are defined by: (i) an icon/figure to represent the element graphically; (ii) a short label to name the element; (iii) a number (between square brackets) to identify the element, according to its context; and (iv) a Boolean value, also placed between square brackets, to indicate if the element is optional.

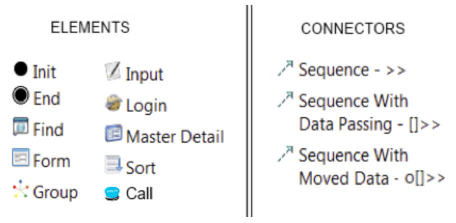


Figure 6: PARADIGM’ Syntax.

3.4 Integration

Eclipse was the selected target platform for PARADIGM. GMF was the selected framework for the development of a specific tool, called PARADIGM Modeling Environment (ME) (Monteiro and Paiva, 2013), targeted to assist the construction of PARADIGM models in the context of PBGT. However, several tools for language implementation have also been evaluated, but they failed in simplicity in the development, maintenance and support aspects.

PARADIGM ME tool development incorporated all the previous definitions of the language: elements and their properties, connectors, graphical representation and constraints. These constraints were implemented in OCL (Cabot and Gogolla, 2012)). In addition, new widgets were defined just for improving end-user interaction (menus, toolbars and buttons). This tool provides support for modeling GUIs using PARADIGM and further functionality: UI Test Pattern configuration; test cases generation from models; test case execution and; test coverage information. Since PARADIGM aims to be an expandable language (to cope with latest UI trends), new elements and connectors may be added. This means the language can be expanded during implementation phase (development) and/or afterwards (modeling phase). For instance, the tester is able to create his own set of behavioral elements by reusing and combining the existent ones within Forms. The set of those Forms can be seen as a library of new elements that can be reused within other model.

4 EMPIRICAL EVALUATION

To assess the effectiveness and real-world relevance of our approach, we have conducted a set of case studies following guidelines from Runeson and Höst (Runeson and Höst, 2009). The goal is to measure the feasibility of the PBGT approach, starting from the modeling phase until the generated test cases output. More specifically, to assess the less efforts required to model a GUI, to configure the UI Test Patterns and the capability to model common UIs. Furthermore, we also want to known the acceptance level of PARADIGM, among GUI testers. Our evaluation addresses the following research questions:

RQ1: What are the overall modeling efforts required to craft a model with PARADIGM, when compared with other Model-Based GUI Testing approaches?
RQ2: Are current language elements and connectors adequate and effective to model UIs, with the goal to finding errors?
RQ3: What is the graphical effectiveness and power of PARADIGM?

4.1 Experimental Objects

italiancharts. This is a public site that reflects the best selling music singles and albums in Italy. It can be found at http://italiancharts.com (to be referred as EO1).
The Address Book. The Address Book is an open source address management system, that in short, allows to create unlimited number of addresses and associated information (to be referred as EO2).
Professional Calendar/Agenda. This application is written in C#, and provides a calendar view for creating appointments and day events (to be referred as EO3).
zkCalendar. This tool features scheduling functionality, with daily, weekly and monthly views (to be referred as EO4).
Industrial Applications. Two industrial applications (to be referred as IA1 and IA2) that due to confidentiality terms cannot be revealed.

4.2 Experimental Setup

To address **RQ1** we have selected three Model-Based GUI Testing approaches, namely: Spec#, VAN4GUIM, and PARADIGM. With these approaches, models have to be crafted manually.

The experiments were performed by two different teams: one comprised by research students and the other comprised by professionals working in a software company (to test their two own internal developed applications). Elements from both teams had not been in contact with PARADIGM developers neither had previous knowledge about PBGT, Spec# and VAN4GUIM. We presented these 3 approaches to the teams in two different sessions during which they had the opportunity to clarify doubts in order to be prepared to start the experiments. In order to address **RQ1** we measured the time (in minutes) required to create and configure the models using the 3 different approaches mentioned before. To address **RQ2** and **RQ3** we gathered feedback from the teams to verify if they were able to model the proposed UIs and to assess language acceptability, graphical effectiveness and communicative aptitude. Thus, to evaluate PARADIGM, we selected 8 dimensions, according to Moody's criteria (Moody, 2007), that in our perspective are the ones that can be applied to evaluate PARADIGM. Then, once the teams finished the construction of the models, we delivered a questionnaire, featuring the selected dimensions. For each dimension, the teams were able to select only one value of the following scale (in terms of perception): High – 3; Medium – 2; Low – 1; Do not know/blank – 0. Further, we also evaluated VAN4GUIM according to the same criteria, in order to perform a comparision between PARADIGM and VAN4GUIM. Spec# was not included in this comparision, since it is not a graphical language.

4.3 Findings

After completion of the case studies above, we obtained the following results. Table 1 displays the time measurements, in minutes, concerning the modeling (displayed as **M** in the table cells) and configuration (displayed as **C** in the table cells) efforts that were required for the experimental objects (EO). This table registered the data to address **RQ1**.

The results regarding the evaluation of PARADIGM and VAN4GUIM, according to Moody's quality criteria, are represented in Table 2. Each table cell shows the average value for each dimension of each method. The total represents the sum of the cells dividing by all dimensions (8). Table 2 displays the data to fulfill **RQ2** and **RQ3**.

4.4 Discussion

By analyzing Table 1, we found that, on average, it took 343 minutes to craft models with Spec#, 243 minutes with VAN4GUIM and 64 minutes with PARADIGM. Regarding the configuration effort, Spec# and VAN4GUIM are similar, i.e., teams spent 48 and 45 minutes respectively. However, with PARADIGM it took 31 minutes. Concerning these results, it is important to mention that with Spec# and VAN4GUIM the focus is directed towards mod-

Table 1: Case studies results.

EO	Method		
	Spec#	VAN4GUIM	PARADIGM
EO1	**M:**303 **C:**46	**M:**181 **C:**41	**M:**62 **C:**20
EO2	**M:**210 **C:**34	**M:**121 **C:**31	**M:**32 **C:**14
EO3	**M:**177 **C:**32	**M:**110 **C:**27	**M:**26 **C:**12
EO4	**M:**183 **C:**40	**M:**125 **C:**38	**M:**30 **C:**22
IA1	**M:**546 **C:**61	**M:**421 **C:**57	**M:**104 **C:**50
IA2	**M:**671 **C:**76	**M:**498 **C:**75	**M:**131 **C:**68

Table 2: PARADIGM and VAN4GUIM findings concerning Moody's quality criteria, with input collected from the teams.

Quality Criteria	**PARADIGM**	**VAN4GUIM**
Discriminability	2.2	2.0
Perceptual and cognitive limits	2.8	2.5
Emphasis	2.9	2.8
Perceptual directness	2.6	2.5
Structure	2.8	2.8
Identification	2.7	2.5
Expressiveness	2.8	2.6
Simplicity	2.7	2.7
Total	$21.5 / 8 \simeq 2.7$	$20.4 / 8 \simeq 2.6$

eling user actions. With PARADIGM the modeling is focused on testing goals. It is possible to conclude that with PARADIGM the modeling efforts and also the configuration aspects are much lower rather than Spec# and VAN4GUIM.

To assess the overall acceptability, the graphical richness and communicative ability of PARADIGM, we collected feedback through a questionnaire fulfilled by the team members. The results are shown in Table 2. We evaluated PARADIGM against VAN4GUIM according to 8 dimensions from Moody's criteria. The values of each cell (from Table 2) represent the teams perception and evaluation of each dimension. The higher the proximity towards value 3, the higher and better their perception is, in respect to the given dimension concerning the graphical notation in question. For example, the discriminability dimension consists on the perception if language elements are easy to differentiate. For this dimension, PARADIGM registered the average of 2.2 and VAN4GUIM the average of 2.0. This result indicates that according to the scale, PARADIGM has a better rating on this dimension. Due to space restrictions, we cannot explain all dimensions. However, the meaning of each one can be found in (Moody, 2007). As for the overall ratings for all dimensions, PARADIGM registered a rating of 2.7 and VAN4GUIM obtained a rating of 2.6. Thus, both notations appear to have similar acceptability, perceptiveness and graphical richness. However, PARADIGM registered the highest score (with a slight difference) among the evaluation performed by the teams. This fact indicates that PARADIGM language elements are expressive, easy to identify, effective and graphically rich. Given the obtained positive feedback, results also point that PARADIGM can be adopted at industry level.

We are aware that more studies are needed which can be seen as a threat to validity. Nevertheless, the case studies presented in this paper point to the diminished effort in modeling GUIs with PARADIGM. Furthermore, we also evaluated the DSL at industrial level and the feedback was promising. Both modeling and configuration efforts for the models, depend on the testing goals designed for each application. This means that depending on the depth and detail of the testing goals, the results could have been different.

5 CONCLUSIONS

This paper presented PARADIGM, a DSL to be used in the context of PBGT, and the steps performed, according to best practices, in the development of such DSL. In addition, it also presents an empirical evaluation of the PARADIGM comparing with other modeling approaches (Spec# and VAN4GUIM) and an evaluation at industrial level. The case studies results show that building models in PARADIGM requires less effort when compared with Spec# and VAN4GUIM. The results obtained from the industrial case study are encouraging and indicate that the overall approach has, in fact, potential of being adopted by the industry.

ACKNOWLEDGEMENTS

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

Brucker, A. D. and Doser, J. (2007). Metamodel-based UML Notations for Domain-specific Languages. In Favre, J. M., Gasevic, D., Lämmel, R., and Winter, A., editors, *4th International Workshop on Software Language Engineering (ATEM 2007)*. Nashville, USA.

Cabot, J. and Gogolla, M. (2012). Object constraint language (OCL): a definitive guide. In *Proceedings of the 12th international conference on Formal Methods for the Design of Computer, Communication, and Software Systems: formal methods for model-driven engineering*, SFM'12, pages 58–90, Berlin, Heidelberg. Springer-Verlag.

Constantine, L. (2003). Canonical Abstract Prototypes for Abstract Visual and Interaction Design. In *Interactive Systems. Design, Specification, and Verification*, volume 2844 of *LNCS*, pages 1–15. Springer-Verlag, Berlin/Heidelberg.

Cook, S., Jones, G., Kent, S., and Wills, A. (2007). *Domain-Specific Development with Visual Studio DSL Tools*. Addison-Wesley Professional, first edition.

Cunha, M., Paiva, A. C. R., Sereno Ferreira, H., and Abreu, R. (2010). PETTool: A Pattern-Based GUI Testing Tool. In *2nd International Conference on Software Technology and Engineering (ICSTE'10)*, SFM'12, pages 202–206.

Fowler, M. (2010). *Domain Specific Languages*. Addison-Wesley Professional, 1st edition.

Grandite (2008). Open Modelsphere – Free Modeling Software Open Source GPL. http://www.modelsphere.org/. Accessed January, 2013.

Gronback, R. C. and Boldt, N. (2013). Graphical Modeling Framework. http://www.eclipse.org/modeling/gmp. Accessed April, 2013.

Jackson, D. (2011). *Software Abstractions: Logic, Language, and Analysis*. MIT Press; 2nd Revised edition.

Karsai, G., Krahn, H., Pinkernell, C., Rumpe, B., Schneider, M., and Vlkel, S. (2009). Design Guidelines for Domain Specific Languages. In Rossi, M., Sprinkle, J., Gray, J., and Tolvanen, J.-P., editors, *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM'09)*, pages 7–13.

Memon, A. M., Soffa, M. L., and Pollack, M. E. (2001). Coverage Criteria for GUI Testing. In *In Proceedings of the 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9)*, pages 256–267. ACM Press.

Microsoft (2012). Model-based Testing with SpecExplorer - Microsoft Research. http://research.microsoft.com/en-us/projects/specexplorer/. Accessed February, 2012.

Microsoft (2013). Spec# - Microsoft Research. http://research.microsoft.com/en-us/projects/specsharp/. Accessed January, 2013.

Monteiro, T. and Paiva, A. C. R. (2013). Pattern Based GUI Testing Modeling Environment. In *ICST Workshops*, pages 140–143.

Moody, D. (2007). What Makes a Good Diagram? Improving the Cognitive Effectiveness of Diagrams in IS Development. In Wojtkowski, W., Wojtkowski, W., Zupancic, J., Magyar, G., and Knapp, G., editors, *Advances in Information Systems Development*, pages 481–492. Springer US.

Moreira, R. M. L. M. and Paiva, A. C. R. (2008). Visual Abstract Notation for GUI Modelling and Testing – VAN4GUIM. In Cordeiro, J., Shishkov, B., Ranchordas, A., and Helfert, M., editors, *ICSOFT (SE/MUSE/GSDCA)*, pages 104–111. INSTICC Press.

Moreira, R. M. L. M., Paiva, A. C. R., and Memon, A. (2013). A Pattern-Based Approach for GUI Modeling and Testing. In *Proceedings of the 24th International Symposium on Software Reliability Engineering*, ISSRE'13, Pasadena, CA, USA. IEEE Computer Society.

Paiva, A., Faria, J. C. P., and Vidal, R. F. A. M. (2003). Specification-Based Testing of User Interfaces. In *Interactive Systems. Design, Specification, and Verification, 10th International Workshop*, volume 2844 of *LNCS*, pages 139–153. Springer.

Paiva, A. C., Faria, J. C., Tillmann, N., and Vidal, R. A. (2005). A Model-to-Implementation Mapping Tool for Automated Model-Based GUI Testing. In Lau, K.-K. and Banach, R., editors, *Formal Methods and Software Engineering*, volume 3785 of *LNCS*, pages 450–464. Springer Berlin Heidelberg.

Paternò, F., Mancini, C., and Meniconi, S. (1997). ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models. In *Proceedings of the IFIP TC13 International Conference on Human-Computer Interaction*, INTERACT '97, pages 362–369, London, UK, UK. Chapman & Hall, Ltd.

Rubel, D., Wren, J., and Clayberg, E. (2011). *The Eclipse Graphical Editing Framework (GEF)*. Addison-Wesley Professional, 1st edition.

Runeson, P. and Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164.

StarUML (2005). Staruml – The Open Source UML/MDA Platform. http://staruml.sourceforge.net/en/. Accessed January, 2013.

Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2009). *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition.

Strembeck, M. and Zdun, U. (2009). An approach for the systematic development of domain-specific languages. *Softw. Pract. Exper.*, 39(15):1253–1292.