Model-Driven Performance for the Pattern and Advancement of Software Exhaustive Systems

Muzammil H Mohammed
Department of Information Technology
College of Computers and Information Technology
Taif University, Taif, Saudi Arabia

Sultan Aljahdali
Department of Computer Science
College of Computers and Information Technology
Taif University, Taif, Saudi Arabia

Nisar Hundewale
Department of Computer Science
College of Computers and Information Technology
Taif University, Taif, Saudi Arabia

Abstract

Model-Driven Engineering (MDE) is an advance to widen software systems by build models and pertaining robotic alteration to them to finally make the execution for a intention platform. Although the main focus of MDE is on the creation of code, it is also essential to bear the study of the design with esteem to quality attributes such as performance. To balance the model-to-execution path of MDE approaches, an MDE tool infrastructure should provide what we call model-driven analysis. This paper project will examine how the pattern and advancement of software exhaustive systems (e.g., all software in an aircraft) can be supported by model-driven techniques, e.g. interactive visual tools, model transformations, and automated consistency analyses. Essential model-driven development, software evolution, software visualization, software product lines, embedded systems Experience in development of Eclipse-based tools.

I. INTRODUCTION

Model-Driven Engineering (MDE) is an advance to create software systems that engage making models and applying mechanical alteration to them. The models are uttered in modeling languages (e.g., UML) that describe the Structure and behavior of the system. MDE tools successively apply pre-defined transformations to the input model created by the developer and ultimately generate as output the source code for the application. MDE tools typically impose domain-specific constraints and generate output that maps onto specific middle-ware platforms and frameworks. MDE is often indistinctively associated to OMG's Model-Driven Architecture and Model-Driven Development.

The ability to create a software design and apply automated transformations to generate the implementation helps to avoid the complexity of today's implementation platforms, component technologies and frameworks. Many MDE solutions focus on the generation of code that partially or entirely implements the functional requirements. However, these solutions often overlook runtime quality attribute requirements, such as performance or reliability. Fixing quality attribute problems once the implementation is in place has a high cost and often requires structural changes and refactoring. Avoiding these problems is the main motivation to perform analysis early in the design process. To complement the model-to-implementation path of MDE approaches, an MDE tool infrastructure should provide what we call model-driven analysis. The model to code path and the model-driven analysis path are notionally represented in Figure 1. The goal of model-driven analysis is to verify the ability of the input design model to meet quality requirements.

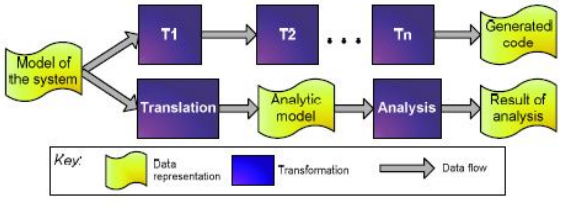

Fig. 1. Model-Driven production and Model-Driven examination

Model-Driven Performance Analysis

Model-Driven Engineering (MDE) is an approach to develop software systems by creating models and applying automated transformations to them to ultimately generate the implementation for a target platform. Although the main focus of MDE is on the generation of code, it is also necessary to support the analysis of the designs with respect to quality attributes such as performance. To complement the model-to- implementation path of MDE approaches, an MDE tool infrastructure should provide what we call model-driven analysis. This paper describes an approach to model-driven analysis based on reasoning frameworks. In particular, it describes a performance reasoning framework that can transform a design into a model suitable for analysis of real-time performance properties with different evaluation procedures including rate monotonic analysis and simulation. The concepts presented in this paper have been implemented in the PACC Starter Kit, a development environment that supports code generation and analysis from the same models.

II EFFECTIVENESS AND INTRINSIC WORTH OF MODEL-DRIVEN ENGINEERING (MD):

Model-driven Engineering (MDE) helps in reduction of effort that is put forth for development and maintenance of the systems. Using the MDE we are not worried a lot about the code. The effort has to be put in only to design the system effectively using the model. Models define what is variable in a system, and code generators produce the functionality that is common in the application domain. The important role of MDE in developing a system is that it always aims to achieve high-performance with low effort. MDE is unification of initiatives that aims in improving software development by employing high-level, domain specific, implementation, maintenance and testing.

The important phase in developing a new system is to know what the exact requirement is and to prepare an abstract model of the system. MDE allows us to effectively build a system which is best in quality and low in effort. This is where all architects prefer the MDE model of building systems because here there is no need of putting a lot of effort in the code generation segment. The efforts are much needed only to create and build the model of the system. MDE also allows us to have two phases to be done parallel. One phase is the development of the software system and the other is the maintenance phase.

MDE concentrates on the domain and so the outcome of the software system that is developed using the model-driven engineering will definitely be very strong in nature with high-level of technical codes.

Enhancing Abstraction:

Abstraction is the most important issue in any software language code. MDE enhances the abstraction and helps in maintaining a rigid functionality between

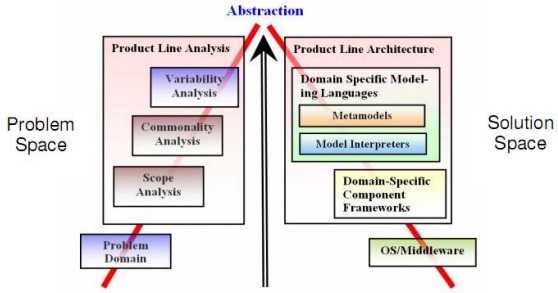

Figure 2: using DSMLs and Domain-specific Component Frameworks to Enhance Abstraction and narrow the gap between problem and solution space of software-intensive systems

Various software system modules. Any vulnerability in abstraction could lead to a large flaw in the developed system which is prone to some illegal attacks. MDE aims at this abstraction enhancement so that the outcome of code is molded in a secured and in a flawless manner.

III PROBLEM AND SOLUTION SPACE:

Problem and solution are very much important for any system of engineering. Only when the problem is stated clearly the output (Expected) will be achieved exactly. To understand the exact problem there are lot of analysis to be done. After analyzing the problem the domain has to be chosen. MDE is Domain Specific and so the system is to be designed by the selected domain. MDE provides various tools in helping the programmer to achieve many complex codes and algorithms that are necessary in developing some real-time systems.

The solution phase is achieved stage by stage and so the process cycle tends to get more complex and difficult to implement. The solution must be tested with various discrete values for robustness. MDE allows various testing to be done in order to evaluate the system's efficiency and reliability. Once the solution to the stated problem is done, the system is ready for release.

MDE reduces the gap between the problem and the solution space which means that the system is closely working with the solution based on the problem

defined. Any Engineering system should have to minimize the space between the problem and the solution so as to achieve better results.

IV AUTOMATED SOFTWARE MODULE CLUSTERING:

Many metaheuristic methods have been successfully applied to software module clustering. The field was established by the seminal work of the Drexel group. In this work, hill climbing was the primary search technique, leading to the development of a tool called Bunch for automated software module clustering. Several other metaheuristic search technologies have been applied, including simulated annealing and genetic algorithms However, these experiments have all shown that other techniques are outperformed in both result quality and execution time by hill climbing.

In order to formulate software engineering problems as search problems, the representation and fitness function need to be defined]. In the case of module clustering, previous work has used the Module Dependency Graph (MDG) as a representation of the problem. The MDG is represented as a simple array mapping modules (array indices) to clusters (array elements used to identify clusters) .The array f2; 2; 3; 2; 4; 4; 2; 3g denotes a clustering of eight modules into three clusters, identified by the numbers 2, 3, and 4. For example, modules numbered 0, 1, 3, and 6 are all located in the same cluster (which is numbered 2). The choice of numbers of module identifier is arbitrary, so this clustering is equivalent to f1; 1; 3; 1; 4; 4; 1; 3g and f3; 3; 2; 3; 4; 4; 3; 2g. The MDG can thus be thought of as a graph in which modules are the nodes and their relationships are the edges. Edges can be weighted, to indicate strength of relationship, or unweighted, merely to indicate the presence or absence of a relationship. As will be seen, the algorithms studied in this paper differ noticeably in their performance on weighted MDGs when compared to the results obtained for unweighted MDGs and so this distinction between weighted and unweighted turns out to be an important aspect of problem characterization. The choice of what constitutes a "module" and what precisely can count as a "relationship" is parameters to the approach. In previous work (and in the present paper), a module is taken to be a file and a relationship is an inclusion of reference relationship between files (e.g., a method invocation). In order to guide the search toward a better modularization, it is necessary to capture this notion of a "better" modularization. The intra-edges are those for which the source and target of the edge lie inside the same cluster. The inter-edges are those for which the source and target lie in distinct clusters. MQ is the sum of the ratio of intra-edges and inter-edges in each cluster, called the Modularization Factor (MFk) for cluster k. MFk can be defined as follows:

MFk ¼
0; if i ¼ 0;
i
iþ12
_ j; if i > 0; ð1Þ

Where i is the weight of intra-edges and j is that of inter edges, that is, j is the sum of edge weights for all edges that originate or terminate in cluster k. The reason for the occurrence of the term 1 2 j in the above equation (rather than merely j) is to split the penalty of the inter-edge across the two clusters that connected by that edge. If the MDG is unweighted, then the weights are set to 1. The MQ can be calculated in terms of MF as

MQ ¼X
n
k¼1
MFk; ð2Þ

where n is the number of clusters. The goal of MQ is to limit excessive coupling, but not to eliminate coupling altogether. That is, if we simply regard coupling as bad, then a "perfect" solution would have a single module cluster containing all modules. Such a solution would have zero coupling. However, this is not an ideal solution because the module would not have the best possible cohesion. The MQ measure attempts to find a balance between coupling and cohesion by combining them into a single measurement. The values produced by MQ may be arbitrarily large because the value is a sum over the Number of clusters presents in a solution and so the MQ function is not a metric. The aim is to reward increased cohesion with a higher MQ score and to punish increased coupling with a lower MQ score. In order to handle weighted and unweighted graphs using the same approach, an unweighted graph is essentially treated as a weighted graph in which all edges have an identical weight.

V VISUAL AND INTERACTIVE TOOLS (GUI):

A GUI is the front-end to software's underlying back-end code. An end user interacts with the software via events; the software responds by changing its state, which is usually reflected by changes to the GUI's

widgets. The complexity of back-end code dictates the complexity of the front-end. For example, a single-user application such as Microsoft Paint employs a simple single-user GUI, with discrete events, each completely predictable in its context of use, used to manipulate simple widgets that change their state only in response to user-generated events. More complex applications require synchronization/timing constraints among complex widgets, e.g., movie players that show a continuous stream of video rather than a sequence of discrete frames, and nondeterministic GUIs in which it is not possible to model the state of the software in its entirety (e.g., due to possible interactions with system memory or other system elements) and hence the effect of an event cannot be predicted. To provide focus, this paper will deal with an important class of GUIs. The important characteristics of GUIs in this class include their graphical orientation, event-driven input, hierarchical structure of menus and windows, the objects (widgets, windows, and frames) they contain, and the properties (attributes) of those objects. Formally, the class of GUIs of interest maybe defined as follows:

$MF_k =$ 0 if i = 0

I / i+1/2 j if i > 0.

A Graphical User Interface (GUI) is a hierarchical, graphical front-end to a software system that accepts as input user-generated and system-generated events from a fixed set of events and Produces deterministic graphical output. A GUI contains graphical objects; each object has a fixed set of properties. At any time during the execution of the GUI, these properties have discrete values, the set of which constitutes the state of the GUI. The above definition specifies a class of GUIs that have a fixed set of events with a deterministic outcome that can be performed on objects with discrete valued properties. GUI testing, in this paper, is defined as exercising the entire application by generating only GUI inputs with the intent of finding failures that manifest themselves through GUI widgets. Research has shown that these types of GUI testing finds faults related not only to the GUI and its glue code, but also in the underlying business logic of the application .Current techniques used in practice to test such GUIs are largely manual. The most popular tools used to test GUIs are capture/replay tools such as WinRunner1 that provide very little automation, especially for creating test cases. There have been attempts to develop state-machine models to automate some aspects of GUI testing, e.g., test case generation and regression testing. In our past work, we have developed an event-flow model that represents events and interactions. The event-flow model was designed to capture GUI events and event interactions, but it does not model some of the Web application characteristics, as we describe in Section 3. In this paper, we use the event-flow model to obtain test cases for the GUI applications.

VI. TEST PRIORITIZATION OF VISUAL TOOLS AND APPLICATIONS:

The software's that are released by the vendors are not constant. Any released product would have some bugs or complexities that are to be fixed by the vendor when the next version is made. So a lot of testing and bug fixing techniques are followed in order to ensure no such issues arise the next time for the product.

In such situations, a large number of test cases may be available from testing previous versions of the application which are often reused to test the new version of the application. However, running such tests may take a significant amount of time. Rothermel et al. report an example for which it takes weeks to execute all of the test cases from a previous version. Due to time constraints, a tester must often select and execute a subset of these test cases. Test case prioritization is the process of scheduling the execution of test cases according to some criterion to satisfy a performance goal. Consider the function for test prioritization as formally defined in. Given T, a test suite, the set of all test suites obtained by permuting the tests of T and f. In this definition, refers to the possible prioritizations of T and f is a function that is applied to evaluate the orderings. The selection of the function f leads to many criteria to prioritize software tests.

For instance, prioritization criteria may consider code coverage, fault likelihood, and fault exposure potential. Binkley uses the semantic differences between two programs to reduce the number of tests that must be run during regression testing [. Jones and Harrold reduce and prioritize test suites that are MC/DC adequate. Jeffrey and Gupta consider the number of statements executed and their potential to influence the output produced by the test cases. Lee and He reduce test suites by using tests that provide coverage of the requirements. Offutt et al. use coverage criteria to reduce test cases. None of these prioritization criteria have been applied to event-driven systems. In our past work, we have developed additional criteria to prioritize GUI and Web-based programs. Bryce and Memon prioritize preexisting test suites for GUI-based programs by the lengths of tests (i.e., the number of steps in a test case, where a

test case is a sequence of events that a user invokes through the GUI), early coverage of all unique events in a test suite, and early event interaction coverage between windows (i.e., select tests that contain combinations of events invoked from different windows which have not been covered in previously selected tests).

In half of these experiments, event interaction-based prioritization results in the fastest fault detection rate. The two applications that cover a larger percentage of interactions in their test suites (64.58 and 99.34 percent, respectively) benefit from prioritization by interaction coverage. The applications that cover a smaller percentage of interactions in their test suites (46.34 and 50.75 percent, respectively) do not benefit from prioritization by interaction coverage.

We concluded that the interaction coverage of the test suite is an important characteristic to consider when choosing this prioritization technique. Similarly, in the Web testing domain, Sampath et al. prioritize user session-based test suites for Web applications. These experiments showed that systematic coverage of event interactions and frequently accessed sequences improve the rate of fault detection when tests do not have a high Fault Detection Density (FDD), where FDD is a measure of the number of faults that each test identifies on average.

VII. MODELING TEST CASES

A test case is modeled as a sequence of actions. For each action, a user sets a value for one or more parameters. We provide examples of test cases for both GUI and Web applications next. A sample test case for a GUI application called TerpWord. The test case sets nine parameters to values and visits three unique windows. The test includes visits to the TerpWord main window, Save, and Find windows. An action occurs when a user sets values to one or more parameters on a window before visiting a different window. From this

Start of TC	<Testcase>
No. of Actions	<Length>4</Length>
Action 1	<Menu> <Window>TerpWord</Window> <Nonterminal>File</Nonterminal> </Menu> <Menu> <Window>TerpWord</Window> <Nonterminal>Save</Nonterminal> </Menu>,

we see that in Action 1, the user selects File->Save from the TerpWord main menu. The parameter values associated with this action are shown in first two rows of

Window name	P-V No.	P-V description (<parameter,value>)
TerpWord	PV.1	<File,null>
	PV.2	<Save,null>
Save	PV.3	<File name text field, SETTEXT="exampleFile">
	PV.4	<Files of Type drop-down box, LEFTCLICK SELECT="Plain Text File (*.txt)">
	PV.5	<OK button, LEFTCLICK>.

The parameter-values set in Action 2 occur on the Save Window to set the file name to "exampleFile," select the file type as plain text, and click the OK button. The user sets parameter-values in Action 3 on the TerpWord main window by selecting Edit->Find. Action 4 involves parameter-values on the "Find" window. The user sets the text of the "Find what drop-box" to "software defect" and then executes a "left-click" on the Find Next button. Table summarizes the windows, parameters, and values in this test case and assigns unique numbers to each window and action.

CONCLUSION

Preceding effort delight stand-alone GUI and web-based applications as break up areas of research. However, these types of applications have many resemblance that let us to create a single model for testing such event driven systems. The main focus of this paper is to create a code for MDE, it is also necessary to bear the study of the design with esteem to quality attributes such as performance. We have covered balance the model-to-execution path of MDE approaches, an MDE tool infrastructure should provider what we call model-driven analysis. This paper plan will inspect how the prototype and advancement of software exhaustive systems. This replica may endorse future research to more generally spotlight on stand-alone GUI and web based applications instead of addressing them as disjoint topics. Other researchers can use our universal model to apply testing techniques more broadly. Within the background of this model, we develop and empirically assess numerous prioritization criterion and pertain them to four stand-alone GUI and three web-based applications and their existing test suites. Our experiential study assesses the prioritization criteria. Our ability to develop prioritization criteria for two types of event-driven software indicates the usefulness of our combined model for the problem of test prioritization. This paper introduces the multi-objective approach to software module clustering. It introduces two multi-objective formulations of the multi-objective problems.

References

1. Schmidt, D.: Model-driven engineering. IEEE Computer Magazine 39(2) (2006)
2. Ivers, J., Moreno, G.A.: Model-driven development with predictable quality. In: Companion to the OOPSLA'07 Conference. (2007)
3. Bass, L., Ivers, J., Klein, M., Merson, P.: Reasoning frameworks. Technical Report CMU/SEI-2005-TR-007, Software Engineering Institute (2005)
4. Klein, M.H., Ralya, T., Pollak, B., Obenza, R., Gonzalez Harbour, M.: A practi-tioner's handbook for real-time analysis. Kluwer Academic Publishers (1993)
5. Hissam, S., Klein, M., Lehoczky, J., Merson, P., Moreno, G., Wallnau, K.: Per-formance property theories for predictable assembly from certi¯able components (PACC). Technical Report CMU/SEI-2004-TR-017, Software Engineering Insti-tute (2004)
6. Gomaa, H., Hussein, M.: Model-Based Software Design and Adaptation. Int. Conferenceon Software Engineering for Adaptive and Self-Managing Systems, p. 7 (2007).
7. Haugen, O., MOller-Pedersen, B., Oldevik, J., Solberg, A.: An MDA-based framework for model-driven product derivation. In: M. H. Hamza, editor, Software Engineering and Applications, pp. 709--714. ACTA Press, Cambridge (2004).
8. Object Management Group, UML Profile for Modeling and Analysis of Real-Time and Embedded Systems, OMG Adopted Specification ptc/07-08-04 (2007).
9. Oldevik, J., Haugen, O.: Higher-Order Transformations for Product Lines. In: 11th Int. Software Product Line Conference (SPLC), pp. 243--254, Kyoto, Japan (2007).
10. Petriu, D.C., Shen, H.: Applying the UML Performance Profile: Graph Grammar based derivation of LQN models from UML specifications. In Comp. Performance Evaluation (T. Fields, P. Harrison, J. Bradley, U. Harder, Eds.) LNCS 2324, pp.159--177 (2002).
11. Smith, C.U., *Performance Engineering of Software Systems*, Addison Wesley, (1990).
12. Voelter, M., Groher, I.: Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In: 11th International Software Product Line Conference (SPLC), Kyoto, Japan (2007).
13. Woodside, C.M., Neilson, J.E., Petriu, D.C., Majundar, S.: The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-like Distributed Software. In IEEE Trans. on Computers, vol.44, Nb.1, pp. 20—34 (1995).
14. Woodside, C.M., Petriu, D.C., Petriu, D.B., Shen, H., Israr, T., Merseguer, J.: Performance by Unified Model Analysis (PUMA). In WOSP'05, Palma de Mallorca, Spain (2005).
15. Woodside, C.M., Petriu, D.C., Xu, J., Israr, T., Merseguer, J.: Methods and Tools for
Performance by Unified Model Analysis (PUMA). Technical Report SCE-08-06, Carleton
University, Systems and Computer Engineering, 35 pages (2008).
16. Ziadi, T., Jézéquel, J.M., Fondement, F.: Product line derivation with uml. In Software Variability Management Workshop, pp 94–-102, University of Groningen Department of
Mathematics and Computing Science (2003).
17. Ziadi, T., Jézéquel, J.M.: Product Line Engineering with the UML: Deriving Products. In Software Product Lines, pp 557--586, Springer (2006).