

Model-Based Extreme Testing

W.E. Howden, Ray Kim and Peter Tran
CSE, UCSD
{howden, rtkim, pptran}@cs.ucsd.edu

I. INTRODUCTION

A new approach to testing is described that combines extreme programming with model-based testing.

In *extreme programming* (XP) [1], testing plays a special role. Development is incremental and at each stage the current set of tests forms a kind of specification. It is also recommended that tests be automated, and several testing frameworks have been developed to facilitate this part of the process.

There are several potential problems with XP testing. A set of test cases may not be a good partial functional specification of the current partial system. It may be too detailed, and what may be needed is a more abstract partial specification. It may be incomplete, and not adequately represent the partial system.

Model-based testing [e.g. 2] is an approach to testing that focuses on more abstract properties of a system. A state model is constructed at a certain level of abstraction and describes the functionality of the system at that level. A set of functional tests is considered complete if the model is adequately "covered". Model-based testing also has several potential problems. For example, models are often not available, even for systems for which they are a natural specification technique. In addition, model-based test requirements are often given as paths through a model that must be followed by the tests. If the model is abstract, then it is necessary to map an abstract model path onto a concrete test sequence of actual user actions and input data, which may be a non-trivial, non-automatable process.

Model-Based Extreme testing (MXT), as defined here, avoids some of the shortcomings of XP and model-based testing by combining these two approaches.

II. MODEL-BASED EXTREME TESTING

The basic idea in MXT is to trace system behavior during a test, and to then use these traces

to generate an abstract model of the (partial) system. The model: a) summarizes the functional coverage of the tests carried out so far, b) serves as a partial functional specification of a partial system, and c) can be used to suggest additional functional and defect-oriented tests.

III. MOCA

A tool called MOCA (Model-oriented Capture Analysis) has been built that can be used to implement MXT [3]. It was designed as an extension to Rational Robot, a capture playback tool sold by Rational Software. When a user runs a GUI-oriented program under Robot, Robot constructs a capture trace that can be used to play back the sequence of user actions. Multiple tests will result in the generation of a set of traces. The MOCA user selects Robot traces from the set and has MOCA add them to an abstract model that it constructs from the traces.

MOCA allows a user to view traces at different levels of abstraction. It maintains a low level model at a low level of abstraction called *abbreviated robot*. From this it is able to construct and display more abstract models called *control sets* and *windows abstraction*.

A MOCA model fulfills functions a) and b) listed above. In addition, MOCA is able to assist the user in determining if the current MOCA model is a complete model for the current partial system. This is done using the *functional wizard*, which is one of several test wizards that are included in MOCA. The functional wizard examines sources of information associated with the current partial system and identifies a set of possible branches that could occur in the current MOCA state model. It then examines the current model to see if there are some possibilities for branching that have not been included. If there are, the tester may choose to construct additional tests and have the abstracted versions of the new Robot traces integrated into the model. This capability corresponds to item c) in the above list.

Defects are often associated with special cases, such as extreme range values, or degenerate data structures. The *special cases* wizard in MOCA is similar to the functional wizard, but in this case it determines if a standard set of special case tests has been carried out for the currently specified (partial) system. If not, it identifies them. As in the case of the functional wizard, the tester may then choose to construct additional tests.

One of the problems in testing is in knowing when to stop. In a MOCA model, instances of the same state can occur more than once. The MOCA user can identify a model state as being *context-free*. This means that system behavior that is rooted at one instance of this state can be assumed to be the same as for other instances of the state. If an instance of a context-free state is encountered for the second time on a model path, it can be replaced by a loop back to the first instance, and further testing along the path can be eliminated. The functional wizard can be used to assist in detecting mistaken context-free assumptions by determining if two supposedly equivalent state instances have the same potential model branches.

Fig. 1 contains a MOCA generated windows abstraction (WA) model for a partial version of a simple dating system. WA models describe observed functionality at the level of windows and window transitions. In this particular WA model there are only 2 kinds of observed functional

behavior, which have common behavior up to the Login window. The top path shows the user successfully finding a date. In the other path, the user is an administrator and chooses to add a member to the dating system database. DatingSystem.3 was declared to be a context free state, so that MOCA generalized the observed behavior by inserting loops in the model. A loop is indicated by a state instance that is underlined, indicating a loop back to the previous instance of the state.

IV. THEORETICAL FOUNDATIONS

Our work also includes a theory of behavioral models. The theory defines concepts such as model abstraction, equivalent models, and context-free behavior. The work on context-free behavior includes several theorems that characterize situations in which context-free behavior occurs, assisting the tester in determining which states may be context-free.

REFERENCES

- [1] Kent Beck, *Extreme Programming Explained*, Addison Wesley, Boston, 2000.
- [2] Ibrahim K. El-Far, James A. Whittaker, Model-Based Software Testing, *Encyclopedia of Software Engineering* ed. J.J. Marciniak, Wiley, 2002.
- [3] W.E. Howden, R.T. Kim, and P.P. Tran, Model-Oriented Capture Analysis (MOCA), Technical Report, CSE, UCSD, 2002.

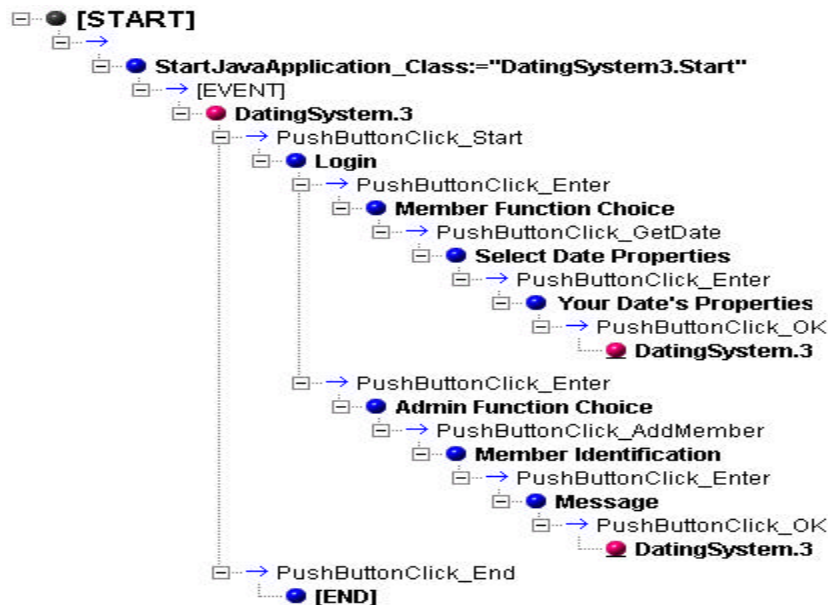


Figure 1. Sample MOCA model for dating system example