

MODEL ORIENTED CAPTURE ANALYSIS (MOCA)

W.E. Howden, R.T. Kim and P.P. Tran
CSE, UCSD July, 2002.

Keywords: Testing, model coverage, extreme testing, extreme programming, abstraction, capture-playback, GUI, Rational Robot, context-free states, context-sensitive models, loops, traces

Abstract

Model-oriented testing requires that a program model be covered during testing. Often a model is not available, either because other kinds of specifications were used or because no specification is available. In the MOCA paradigm, as in extreme programming, tests are used as a specification for a system. MOCA carries this further, using them to construct a model of the kind that is associated with model-based testing. A MOCA model summarizes the tests that have been performed, provides a high level functional specification, and can be used to suggest additional kinds of tests to perform. The basic ideas of the MOCA approach are described, along with a prototype MOCA tool that builds models from Rational Robot traces. Additional features of the MOCA tool include wizards that suggest additional kinds of tests that might be carried out, either because the model seems incomplete, or because it has features that suggest the possible occurrence of commonly occurring kinds of faults. A theory of behavioral modeling is introduced that formalizes the concepts of model abstraction and model equivalence. It introduces the idea of a context-sensitive state model. Additional concepts include that of the context-free-completion of a set of traces, which is used to analyze the soundness of a model-synthesizing procedure.

INTRODUCTION

When testing at the integration or system level, it is often not practical to use code coverage as a measure of test completeness. Model-based testing has received increasing attention as a systematic approach to functional testing and as a way of measuring test completeness at this level.

In the traditional approach to model-based testing, the tester assumes the existence of some kind of model of the software, usually a finite-state model, and then constructs tests that "cover" the model. One possible standard requires that all transitions in a state model be exercised in at least one test. Other kinds of models can also be incorporated into a model-based approach, such as decision tables. The models form a kind of specification for the system.

Often, no model-oriented system specification is available, and some other approach to functional, system level testing must be taken. In the MOCA approach, we assume that the tester runs tests, using some initially unspecified approach to test selection, and that a MOCA model that is based on the observed behavior of the system under test is automatically constructed. The MOCA model serves two purposes:

- i) it gives the tester a summary of the testing effort that has been carried out, and
- ii) it produces a specification of the software.

In the MOCA approach, a model is built up incrementally. Each time a test is completed, a description of the observed behavior can be added to the MOCA model. When the model is considered to be complete by the tester, testing can be halted. This may occur repeatedly, as increasingly expanded versions of a system are produced.

This paper describes a particular implementation of the MOCA paradigm that is based on Rational Robot [26]. Robot is a capture-playback tool from Rational Software. When a user runs a program under Robot, Robot constructs a behavior trace, which allows it to play back the program as though a user were duplicating the actions he or she previously carried out. Robot can be used for different kinds of programs. One of these involves executing programs with which a user interacts through a GUI. The traces generated by Robot show events corresponding to user actions, and the transitions from one window to another. This paper includes the description of a MOCA tool that uses GUI traces to build a MOCA behavioral model.

The paper begins with a discussion of MOCA behavioral models. This is followed by a section on certain theoretical properties of models that are relevant to the MOCA approach. After this, a way of classifying program defects is described. The classification is used as the basis for a testing methodology that uses MOCA, and forms the basis for the development of the MOCA test wizards. More details are given for the MOCA tool, followed by sections on related work and future developments.

MOCA BEHAVIORAL MODELS

State vs Behavioral Models

Model-based testing is usually based on state models. In a formal approach to state models, we define a set of state variables. Different states in the model correspond to different possible combinations of values for the state variables. State transitions in the state model usually correspond to exogenous events that are initiated outside the system. They are associated with changes to state variables that define the new state that results from the a transition.

Implicit state models are given in the form of a set of state variables and a description of the state variable changes that are made in response to events. Explicit state models list all the states with the transitions from one state to the other. The former are often represented as a list or table of conditional assignments and the latter as 2-d graphs, with state and transition labels. A state label in a graphical representation may be thought of as the value of a kind of state variable.

Some kinds of state models allow actions to be associated with transitions and states. A state may have both an entry and an exit action. Other variations include the idea that the total model is the product of a set of submodels, or that one model may be nested inside another. Transitions may also have conditions: when an event associated with a transition occurs, the transition only takes place if the condition is satisfied. Typically, the condition will be a predicate on the state variables.

MOCA models are a kind of state model. However, in MOCA we are concerned with behavioral as opposed to specification models, which means we build a model that is based on observations of what a program does, as opposed to a specifications model that describes what a (possibly unbuilt, unobserved) program is supposed to do. One consequence of this is that we may only have a partial model in the sense that we only know the values of some subset of the state variables. i.e. the state variables associated

with observable behavior. This leads to a situation where a model could have two states with the same "name", i.e. the same set of values for the (observable or abstract) state variables. A MOCA model may be partial in another way also: since it is built up from observations corresponding to Robot test traces, at any one time it only models the behavior that has been seen so far. Parts of the model that correspond to other traces may be missing.

DS and Robot Traces

Robot GUI traces are generated by Rational Robot, as a user interacts with a program through the program's GUI interface. The resulting trace can be used along with Robot to play back the trace/test.

The examples in this paper were generated using a very simple Java program that was written to illustrate the use of MOCA. The program, referred to as DS (dating system), allows a user to search for a date from a database (actually, a simple file) of dating system members. In this simple system, DS begins with a Start/End dialog window with a Start and an End button. If the user clicks the Start button, then a Login dialog is displayed. When the user logs on by entering text in an edit box, and clicking the Enter button, DS determines if this is a regular user, the administrator, or an unknown user. Clicking the Enter button in the Login dialog results in: the display of a Member Functions choice dialog, an Administrator Functions dialog, or an unauthenticated user Message. In the Member Functions dialog, the user can choose to get a date or to change his or her personal characteristics. In this very simple system there are only four dater/member characteristics: religion, occupation, name, gender and email address. The change option allows any of these to be changed. The get-a-date option allows a search for a match in the member database, based on gender and religion. A warning message is displayed before displaying the results of a successful search in case the discovered datee is determined to be a "frequent datee".

If the user is the administrator (recognized by name), the Administrator Functions dialog is displayed, which allows the user to add or delete a member. Attempts to add or delete a member result in a message dialog that indicates whether or not the operation was successful. After the completion of a member or administrator option, or in the case where the logged on user was not recognized, the system returns to the Start/End dialog.

Figure 1 contains the Robot trace for the case where a user named Ray who logged on and then successfully found a date whose gender was female and whose religion was spiritual. In the Robot script, (un-indented) lines 8, 11, 15, 18, 25, and 28 each denote a new window via the `Window SetContext` statement. Each of these windows is uniquely identified by the title of the window (the `Caption` value). The events corresponding to the actions carried out by the user are denoted in several different ways. For example, the `PushButton Click` statement of line 9 causes the DatingSystem 3 Window to be exited and the Login window to appear. `EditBox Click` and `InputKeys` events are also represented in the trace.

Abstractions and MOCA Models

Robot traces are used to construct MOCA models. A set of Robot traces can be thought of as a simple kind of behavioral model. Since all the traces have the same

program start point, they can also be considered to be a primitive kind of tree model where there is one common root node from which all the (separate) paths emanate.

```

Sub Main
  Dim Result As Integer
  'Initially Recorded: 6/26/2002 10:10:58 AM
  'Script Name: 20020626_GetDateSuccess
  StartJavaApplication Class:="DatingSystem3.Start",
    CP:=".;E:\;E:\DatingSystem3\;D:\Rational\.\Rational
    Test\javaenabler\sqarobot.jar", JvmKey:="Java"
  Window SetContext, "Caption=DatingSystem.3", ""
  PushButton Click,
    "JavaCaption=DatingSystem.3;\;Type=PushButton;JavaText=Start"
  Window SetContext, "Caption=Login", ""
  InputKeys "Ray"
  PushButton Click,
    "JavaCaption=Login;\;Type=PushButton;JavaText=Enter"
  Window SetContext, "Caption=Member Function Choice", ""
  PushButton Click, "JavaCaption=Member Function
    Choice;\;Type=PushButton;JavaText=GetDate"
  Window SetContext, "Caption=Select Date Properties", ""
  ListBox MakeSelection, "JavaCaption=Select Date
    Properties;\;Type=ListBox;Index=1", "Text=Female"
  ListBox MakeSelection, "JavaCaption=Select Date
    Properties;\;Type=ListBox;Index=2", "Text=Spiritual"
  PushButton Click, "JavaCaption=Select Date
    Properties;\;Type=PushButton;JavaText=Enter"
  Window SetContext, "Caption=Your Date's Properties", ""
  PushButton Click, "JavaCaption=Your Date's
    Properties;\;Type=PushButton;JavaText=OK"
  Window SetContext, "Caption=DatingSystem.3", ""
  PushButton Click,
    "JavaCaption=DatingSystem.3;\;Type=PushButton;JavaText=End"
End Sub

```

Figure 1 Robot script for DS test case: Date Found

MOCA does two things to this simple model: it abstracts it and it constructs a condensed representation.

The current version of the MOCA tool can generate several kinds of models. The simplest is called *Abbreviated Robot* (AR). This model is formed by keeping the essential events in a Robot Trace, but abstracting out some of the more detailed operations such as individual keystrokes. Figure 2 contains the AR abstraction of the Robot trace in Figure 1.

The most abstract MOCA model, and most widely used, is the *windows abstraction* (WA) model. In this model, each state node corresponds to a window, such as a frame or dialog. The node label (i.e. state variable) is equal to the window caption. This means that if two windows have the same caption, then the model will have two states with the same label. The transitions in this model correspond to the events, such as clicking a button, that cause a transition from one window/dialog to the next. The actions that occur

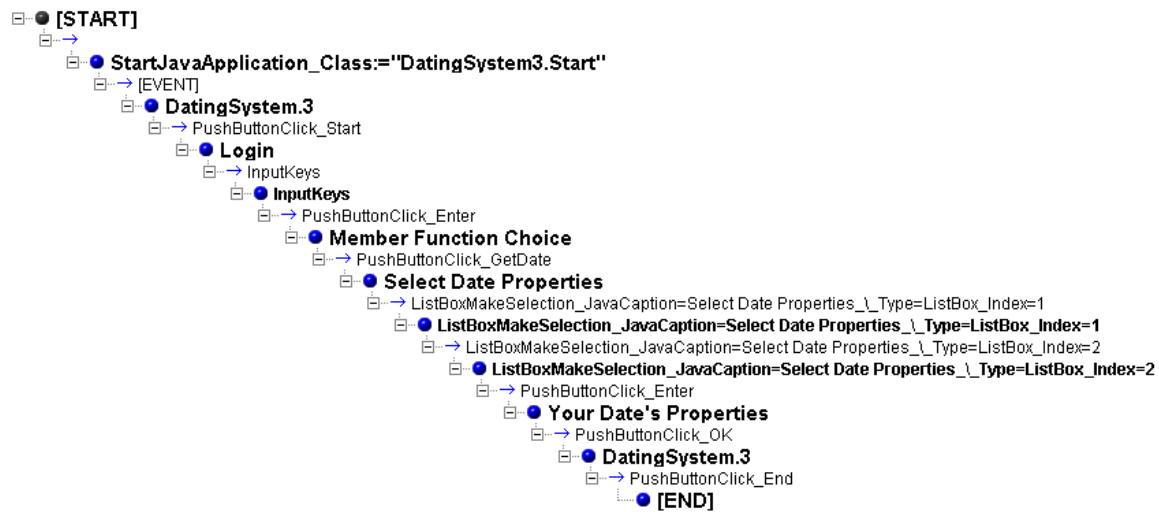


Figure 2: Abbreviated Robot abstraction for Date Found script

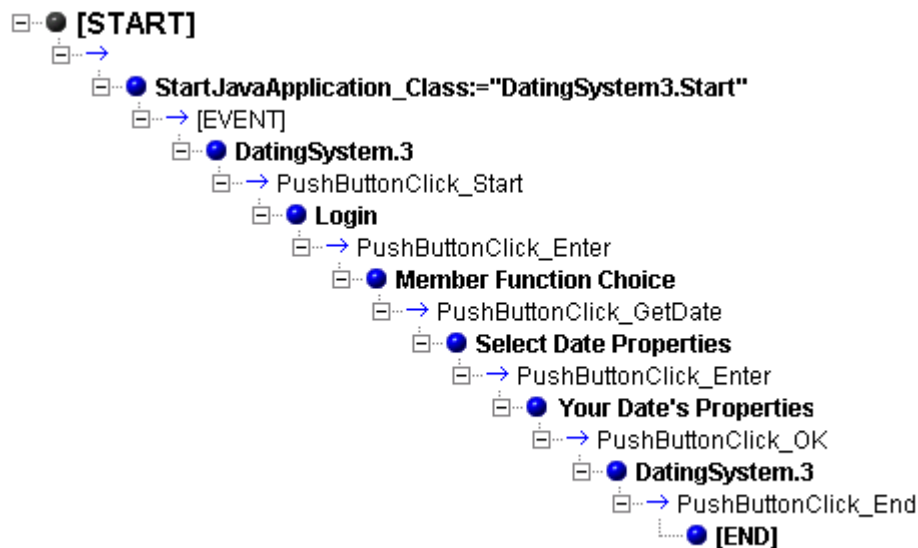


Figure 3: Windows abstraction for Date Found script

"inside" a window are abstracted out. Figure 3 contains the WA abstract path for the Robot script in Figure 1.

A Robot model will consist of a set of paths, integrated into a tree or graph structure. Figure 4 contains a simple, tree-structured model in which two paths have been combined. It contains the WA path from Figure 3 together with the WA path abstraction for the Robot trace corresponding to the case where the user is the administrator, who successfully adds a new member to the DS database.

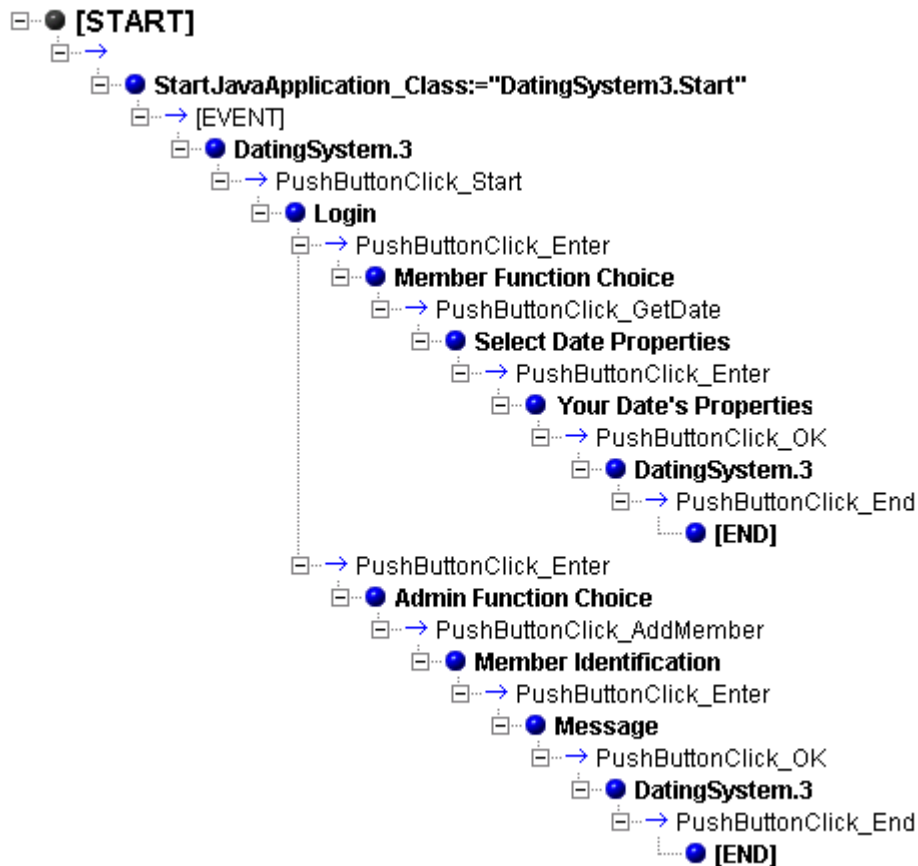


Figure 4: Window Abstraction model with two paths

One of the significant features of a WA abstract model is possible non-determinism. For example, depending on what is entered in the edit box in the Login window, the Enter pushbutton click can lead to three different dialog boxes. Two of these possibilities are seen in Figure 4 where the PushButton event following the Login state leads to two different states: Member Function Choice and Admin Function Choice.

Loops and MOCA Models

In the above examples, the models are all trees. MOCA is also capable of generating tree-based models in which simple loops are allowed. Loops are used to represent a return to a previous system model state, such as may occur when some function has been completed and the system returns to its initial, start state. In MOCA, a loop is identified when a tree path ends in a state node x' whose label is the same as an earlier state x'' on the path. In the notation used by the MOCA tool, such nodes are colored red, which means the path loops back to x'' from x' . This notation allows a model to describe an infinite number of paths (corresponding to different iterations of a loop before some other, terminating subpath is chosen) and also paths of infinite length (corresponding to a path that keeps looping forever). Figure 5 contains a (partial) model for DS, which has several loops back to the Login state. The red states have been underlined to distinguish them in black

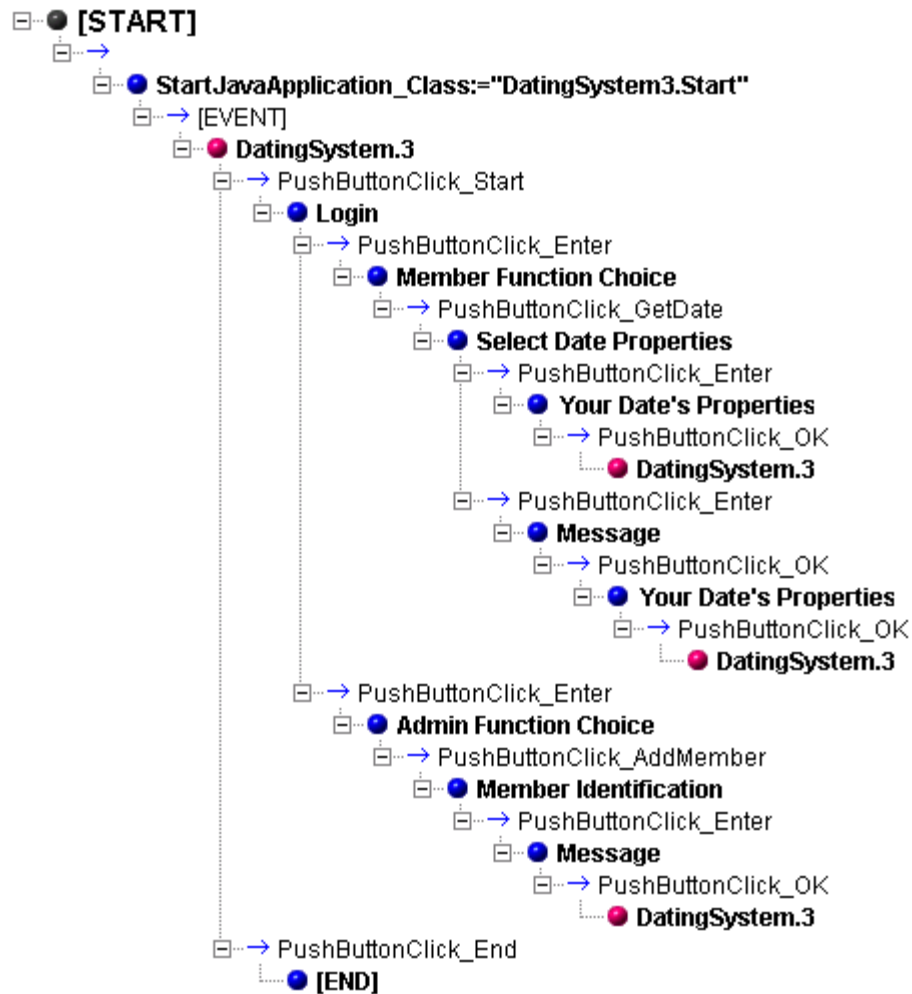


Figure 5: Model containing loops

and white. In this case, the model was produced from two test cases. In the first, a member user logged on, and then used the get-a-date option to successfully find a date. After this, when the system returned to the DatingSystem.3 state with the Start and End buttons, the member continued (by clicking the Start button) and went on to choose another date. This time the behavior of the system was slightly different, because the selected datee was a "frequent datee", which was indicated by the Message state. After this the user member clicked the End button in DatingSystem.3 state, terminating the session/test.

Before these tests were run, DatingSystem.3 was added to MOCA's list of known "context-free" states. These are states for which it is assumed that when you see such a state for the second time, the state is "reset", so that all behavior that could occur the first time, can also occur the second. The model in Figure 5 describes DS as being known to

be possible of exhibiting more kinds of behavior than were actually seen. For example, the loop-back from DatingSystem.3 at the end of the test where the administrator added a member returns to a state from which there are paths where the logon action and add member functions are possible. It indicates behavior in which the user could, at this state, logon as a member, instead of the administrator, and successfully search for a date. It also indicates the possibility of the user/member logging on, and selecting a date, over and over before selecting the End button in the DatingSystem.3 state. All of these different kinds of unobserved behavior that were incorporated into the model are valid forms of behavior if :

- i) we observed the behavior on the two tests that were run, and
- ii) DatingSystem.3 is a valid "reset" state.

One of the more powerful features of the MOCA tool that was used for these examples is its ability to synthesize loops, which is discussed more thoroughly in the following section on theoretical foundations.

Initial Conditions and Model Interpretation

Our sample DS program, like many programs, involves a database (in DS, a file). These means that each time the system is used, there may be a new set of initial conditions that affects its behavior. A MOCA model is interpreted to be a description of possible behavior that *can* occur for *some* set of initial conditions. As a MOCA DS model is built up, one trace at a time, the contents of the DS database could change if one of the traces involves database modifications. This means that different paths in the same MOCA model, corresponding to different abstracted Robot traces, may not work with each other's assumed database. This fact is the motivation for having a model with the interpretation given above.

THEORETICAL FOUNDATIONS

In this section we will treat some aspects of a MOCA model more formally, and also introduce several additional concepts.

Model Abstraction

We assume that each model consists of a collection of model paths. In the case of an unprocessed Robot model, these are Robot traces. The raw Robot model is considered to be an example of a *concrete*, i.e. non-abstract, model. We assume that each model has a common starting state or root node.

Definition Suppose that we have two models M and M' of a system S, consisting of a set of state model paths. Then M' is an *abstraction* of M if it specifies an equivalence relation over the paths of M where each path in M' corresponds to an equivalence class of paths in M.

Definition Suppose that P' is a path in the abstract model M' and that P is a path in M that is in the equivalence class corresponding to P'. Then P is a *realization* of P'. If P is a realization of P', and P occurs during some use of the system under analysis then we can say that the abstract behavior P' is exhibited.

In practice, we will not want just any kind of equivalence relation to be used as an abstraction. One standard kind of abstraction is *projection*, in which states are merged by eliminating some of the state variables. In practice, projection may be too general. In MOCA, WA models are produced by a restricted kind of projection we will call "contiguous subpath projection". In this case, contiguous path segments for which the states all have the same window caption are replaced by a single state that is labeled with the caption. At the same time, event transitions between combined states are deleted.

Alternative Representations

Recall that we can have different models for the same set of Robot traces. There are two ways in which the models can differ. They could be different abstractions of the same set of concrete paths. Alternatively, we could have different representations of the same set of abstract paths. In the latter case, one MOCA model might be a simple set of paths. A second could be a tree, formed by overlapping the common initial segments of the paths. This is an example of a process in which one representation is formed, or derived, from another.

Definition Suppose that R and R' are two representations of the same set of model paths, and that R' has been derived from R. Then R is a *valid derivation* if it is

- i) *sound*, i.e. every path in R' is also a path in R
- ii) *complete* i.e. every path in R is also a path in R'

The process of forming a tree model representation from a set of paths by overlaying their initial common segment is called *common prefix overlay*. It is easy to see that this derivation is valid if there are no loop-backs. But what if we add a path P to the model, and put in a loop-back because it has a second instance of a state that has been assumed to be a "reset state"? In this case, we now have more paths in M than were in the original version of M, plus the added path P, so that by the above definition the derivation would be unsound.

In order to examine this and other issues more carefully, we introduce a more formal approach to "reset states" here, characterizing them as examples of *context-free behavior*.

Context-free behavior

We consider each node in a MOCA state model to be an instance of a *state* that exhibits certain kinds of *behavior*, in response to specified events. The root node of a MOCA model, for example, is a state instance whose behavior is described by that model. We consider the other nodes to be state instances also, whose behavior is described by the parts of the models that can be reached from that node. In the case of a tree model, the behavior of each state-instance/node corresponds to the subtree rooted at that node. Two nodes x' and x" that have the same "label" x (i.e. state variable values) in a model are *instances* of the same state x, that is identified with that label. In general, we assume that each model has a start node and that each state-instance/node can be reached by some path from the start node/state-instance.

Allowing different model nodes to have the same label (i.e. to be instances of the same state) is necessary for several reasons. One reason, that was mentioned above, is that we may only see partial state information when we observe behavior, especially at some level of abstraction. We may see state instances that look the same in the sense that they have the same label in a model and have the same behavior at the associated level of abstraction, even though they could exhibit different behavior at a lower level of abstraction. Another reason has to do with model representation. We use a tree-structured model in which it may be necessary to duplicate states for the purposes of maintaining the form of the model. In general, we will refer to model nodes as states, distinguishing them as instances of a state when necessary.

Definition Suppose that T a tree of all paths that are rooted at an instance of x in a model M . The *local behavior* of an instance of a state in a model consists of the branches rooted at that state, together with the states at the terminal ends of those branches. The *extended behavior* of an instance x' of a state x consists of the complete (possibly infinite) tree of paths rooted at x' . Unless otherwise specified, the term behavior refers to extended behavior.

Definition Suppose that x' is an instance of a state in a model M . Then a path from the start node of M to x' is called a *context* for state x . i.e. a context for a state defines an instance of the state.

The definitions lead to the following, more detailed characterization of model interpretation. Suppose that x' is a state instance in a model M for a system S , and that x' has a context C . Suppose that Q is a path in the behavior for x' in M . This means that

- i) there is some set of initial conditions for S (e.g. database for DS) such that,
- ii) in the context C , the behavior Q could be exhibited by x (i.e. by instance x' of x). Saying the behavior could occur, means the events in Q could occur, causing Q to be observed.

In the case where the model is abstract, we may want to state this a little more precisely, i.e.

- a) there is some set of initial conditions for S (e.g. database for DS), and some realization of the path CQ , such that
- b) if the events in the realization of the context C occur, the behavior in the realization of Q could occur, causing Q to be observed

As mentioned above, a model can have states with the same "label", i.e. there may be two or more instances of a state. Some representation derivations depend on knowing when these really are the "same" state in some meaningful sense. One way of doing this is to say that states/instances are the same if, in a complete model they have the same extended potential behavior. This would mean that at that level of abstraction, they cannot be meaningfully distinguished. We can approach this issue more formally as follows.

Definition Suppose that x' and x'' are instances of the state x in a model M , i.e. they correspond to states labeled as x (and hence are reached by two different associated contexts in the model). If they have the same (local) behavior, then they are considered to be *(locally) equivalent*.

Definition If every instance of a state x in a model M has equivalent local behavior, then the state is said to be *locally context-free*. If every instance has equivalent (extended) behavior then the state is *extensionally context-free*. In general, we will use the term "context-free" to mean extensionally context-free.

The concept of equivalent state instances is important to the design and use of the MOCA tool. As we will see below, the user is able to indicate that certain states are context-free, while forming a model, in order to introduce "loops" in the model. When a user says a state is context-free, this means that for all instances of x in a "complete" model, the extended behavior for x is the same. The effect on the construction of a model which is based on a partial set of observations is to allow a representation to be constructed which may involve the addition of more behavior than has been observed, but which is known to be valid if certain states are context-free. The assumption of context-free behavior may be assisted by theorems such as the following, which characterizes situations in which complete context-free behavior occurs.

First we state a trivial theorem that is an artifact of the definitions.

Theorem 1 If a state x can only occur in one context it is context-free.

More complex theorems include Theorem 2. First we prove a lemma.

Lemma 2.1

Suppose that x' and x'' are instances of a state x occurring in a model M , with contexts C' and C'' , and that all states in M are locally context-free. Let T' and T'' denote the associated behavior subtrees in M rooted at x' and x'' . Let $k > 0$, and consider the sets S' and S'' of all paths, or initial segments of paths, of length equal to k from root nodes x' and x'' in T' and T'' . Then for every path in T' there is a matching path in T'' , and vice versa. Matching means the states and nodes have the same labels.

Proof By induction on k . Local context-free implies the statement is true for $k=1$. Assume that the state instances x' and x'' are arrived at by the context paths C' and C'' in M . Consider a subpath of length $k+1$ from the root node x' of subtree T' . Assume it has the form $D'y'e'z'$ where $D'y'$ is the initial subpath of length k (terminating with state y'), e' is an event, and z' is the state node in tree T' arrived at by event e' from the last node, y' , in the length k subpath. From our inductive hypothesis there must be a matching path $D''y''$ of length k in the subtree T'' , with final state node y'' , which matches y' . Since y' and y'' are matching states, i.e. are instances of the same state at nodes y' and y'' . and because states are locally context-free, then there must be a branch e'' from the node y'' at the end of D'' that goes to a node z'' , where e'' matches e' and z'' matches z' . Path $D''y''e''z''$ will be a path of length $k+1$ that matches $D'y'e'z'$.

QED

In the above proof, it is interesting to note the following. The existence of a path $D''y''$ of length k that matches the path $D'y'$ means that there is some set of initial conditions, and some sequence of user behavior corresponding to $C''D''y''$ (or sequence of user behavior/events corresponding to a realization of $C''D''y''$, if M is an abstract model) that causes the behavior $C''D''y''$ to be observed. Refer to this as $B1$. When we use the assumption of local context-free behavior to conclude that there must be a branch $e''z''$ from y'' that matches the branch $e'z'$ from y' , this implies that there must be initial conditions, and a sequence of user behavior corresponding to $C''D''y''$ (or realization of this if M is an abstraction) that causes $e''z''$ to be observed. Call this $B2$. Note that this is not necessarily the same as $B1$: it could involve a different set of initial conditions and a different realization of $C''D''y''$ in the case where M is abstract. All they need to have in common is the path segment $C''D''y''$.

Consider the general case of the theorem now, for all paths out of two states x' and x'' that are instances of the same state x .

Theorem 2 Suppose that M is a model in which all states have local context-free behavior. Then all states will have extended context-free behavior.

Proof From the lemma, for every finite path of length k in the behavior T' of a state instance x' , there is a matching path in the behavior T'' of x'' . Consider now the infinite paths in T' .

Suppose that there is some infinite path G' in T' which does not have a matching path in T'' . Then there must be some finite k such that it is the length of the longest matching initial segment E' in G' that has a match with an initial segment E'' , of length k , of some G'' in T'' . Because of assumed locally context-free behavior, there is a path G'' for which k is at least 1. Consider now the initial segment of F' of G' of length $k+1$, which must exist because G' is infinite. From the lemma, we know there is a path H'' in T'' which has an initial length $k+1$ segment F'' that matches the initial $k+1$ length segment of F' in T' . This contradicts the assumed maximal property of k . Hence there is no such k , and there must be a matching infinite path G'' in T'' for G' in T' .

QED

Context-free Completion

We now consider the issue of a more general concept of soundness. When we introduce loops into a model we want some way of confirming that the new paths are not invalid. This problem is approached by defining the concept of the "context-free completion" of a set of paths.

The MOCA tester uses the concept of context-free behavior in the following way. We do not expect state instances to be context-free with respect to their behavior in a given model, which may be a partial model, developed incrementally during a testing effort. When a tester *assumes* that a state x is context-free, it is an assumption made about what a complete model would look like, i.e. about its potential behavior. The idea is that if every possible behavior of a system were tested, then in any model that represents that behavior, x would be context-free.

Suppose that some state x is assumed to be context-free. Then if a second instance, x'' , of it is discovered on some path P , after a first instance x' , the tail subpath Q of P that follows the second instance could also occur after the first instance, x' . In addition, the tail R of P from the first instance x' of A , could also occur after the second instance x'' . We could represent both the path that occurs, and the implied paths by: eliminating the subpath Q after x'' , putting a "loop-back" to x' from the state preceding x'' , and then putting in a new branch from x' to a new path consisting of Q . Figure 5 in the previous section contains an example of the application of this loop-back operation.

We first provide a general definition of this concept, by defining the concept of a context-free completion of a set of paths.

Definition Suppose that PS is a set of (possibly infinite) paths, and that one or more states x have been declared to be context-free. Then the *context-free completion* PS' of PS is the set of paths that is defined as follows. PS' contains all of the paths in PS , plus all paths that can be formed from any two paths P_1 and P_2 in PS' using the following operation.

Suppose that P_1 and P_2 have instances x' and x'' of the common context-free state x . Suppose that P_1 and P_2 have the following form:

$$P_1 = H_1xQ_1 \quad P_2 = H_2xQ_2,$$

where H_1/H_2 is the initial segment of P_1/P_2 up to the identified instance of x , and Q_1/Q_2 is the tail segment of P_1/P_2 after the identified instance of x .

Then the path

$$H_1xQ_2$$

is also assumed to be in PS' .

In the above, P_1 and P_2 may be the same path. Also, it is noted that, in general, PS' may be an infinite set. For example, consider the single path $P = xeyfygwh$, where x, y, w are states, and e, f, g, h are events. Assume that y is context-free. Following the above definition, a general context-free completion of the set of paths that included P would also include

$$xeyfyfygwh, xeyfyfyfygwh$$

and so on.

The following gives a related description of an alternative kind of path completion.

Definition Suppose that PS is a set of paths that contains instances of one or more context-free states. Suppose Q is a path that is constructed from paths in PS as follows. Q starts out at the beginning of some path P , and follows it up to an instance of a context-free state x . At this point it "crosses over" to some other path P' that contains an instance of x , and starts following P' from that instance until it reaches an instance of a context-free state y (which may be an instance of a different state or of the same state x). At this point it crosses over to a path P'' which contains an instance of y , and starts following P'' from that point forward. Q continues this process of following one path in PS then crossing over to another at an instance of a context-free state either indefinitely or until it

reaches the end of one of the paths it is following. This process of generating Q is called a *random path walk* or just a *random walk*.

Note that PS could be a single path, and that a random walk can be made along a single path P , between different instances of the same state in P .

The following theorem is used in Theorem 4.

Theorem 3 Suppose that a path Q is constructed from a set of paths PS using a random path walk through paths from PS . Then Q is in the context-free completion of PS .

Proof Define a sequence of paths Q_i as follows. Suppose that P and P' are the first two paths to which the random path construction is applied. Start with the path Q_0 equal to P . Consider the first crossover to P' , and the tail of P' from that point forward. Construct a path Q_1 that is formed from the head of Q_0 up to the cross over in Q , and the tail of P' after the crossover (retaining one copy of the crossover context-free state). By the definition of context-free completion, Q_1 will be in the context-free completion of PS (i.e. of P and R). Now traverse Q_1 along the tail that was copied from P' and at some point we will cross to another path P'' at a cross-over state. Construct the path Q_2 that is formed from the head of Q_1 up to this point, and the tail of the new P'' from this state. Since P'' and Q_1 are members of the context-free completion of PS , then Q_2 must be also.

In this manner we can define a sequence of paths Q_i . If the sequence is finite, the last member will match the path Q , and it will therefore be in the context-free completion of PS . In the case where k is infinite, we define an infinite path Q by defining for all k , the head of Q up to the k 'th crossover point, for arbitrarily large k .

QED

Before showing that our MOCA model-building procedure is sound, we will give a more formal definition of the procedure.

Definition The following procedure, used in the current MOCA tool, will be referred to as MOCA-Build, or MB. If a path P to be added to a partial model M does not contain any context-free states we simply add P to the model, using common prefix overlay. If P contains instances of the context-free state x , then we add the set of paths defined by the following procedure. We will first define the operation of the algorithm for a simple case and then a slightly more complex case. The general case is then given. In the initial two examples we assume that the subpaths D, E, F, G and H do not contain any instances of any context-free state. MB is recursive in the sense that it creates two paths at each non-final stage. One is ready for addition to the model (using common prefix overlay), and the other needs to be recursively processed by MB.

a) Simple case. Suppose that P has the form $P = DxExG$

- i) add the path $Dx(Ex)^+$ to the model, where $+$ means repetitions of 1 or more. To do this we add the path DxE to M , performing prefix overlay as appropriate, and then when we get to the end of E put in a loop-back to the instance of x just after D . In the notation we are using, this means put a

special loop-indicating instance of x following E . We will refer to this as x^* , so that we add DxE^* to the model. In our graphical figures, x^* is colored red and underlined (to make it visible in black and white).

ii) add the path DxG to the model, performing prefix overlay

b) More complex case. Suppose P has the form $P = DxE^*FxGxH$

i) create the path DxE^* , plus the path $DxFxGxH$; add the first to the model

ii) apply MB to the second path above, creating the path $DxFx^*$, plus the path $DxGxH$; add the first path to the model

iii) apply MB to the second path above, creating the path $DxGx^*$, plus DxH ; add the first to the model, and also the second path, DxH .

c) General case. Suppose that P is a path to be added to model M , and that it contains context-free states. Assume that P has the form $AxBxZ$, where x is the first occurrence of a context-free state in P that is repeated, and that B is such that the instance of x after B is the first repeated instance of x .

i) add $AxBx^*$ to M (using prefix overlay)

ii) if Z has no repeated context-free states, then add AxZ to M (using prefix overlay) else apply MB recursively to AxZ .

We prove that the above procedure is "correct" in the sense that it terminates, and that the model that it builds is complete and sound. Here, "complete" means all the paths from the original path set PS are in the model, and sound means that any extra paths it adds that were not in M and P are in the context-free completion of M and P . First we prove that the algorithm always terminates, and then that it produces "correct" output.

Theorem 4 For any finite set of paths, MB will terminate.

Proof Each application of MB to a path P results in a path to be added to the model, plus another path P' derived from P , to which MB is recursively applied. MB identifies the first pair of instances of a repeated context-free state, if there is an instance, and is then re-applied to a reduced path containing one less such pair, so it must terminate.

QED

Theorem 5 MB is complete in the sense that if M is a model, and we use MB to add path P to M , to form a new model M' , then P will be in M' .

Proof We will consider this in two cases, to help with the explanation. First we note that we can assume that P will be finite, since it corresponds to an actual test trace/observation. If it contains a repetition of a context-free state, then we will add a loop structure to M' that will cause the inclusion of multiple, looping paths, denoted with our special x^* path notation.

i) No loops. If P does not have any repetitions of context-free state instances in it, then it is simply added to M using context-free overlay, which does not change it in any way, so that it will be in P .

ii) Loops. Suppose that P contains at least one repeated instance of at least one context-free state. Then it will be added in sections. Each section that corresponds to a repetition of a context-free state will be added in the form of a loop, using the x^* notation. Sections

that correspond to successive repeated occurrences of the same context-free state will be added at the same loop-back point in M , as a result of using the same common prefix overlay. In this way, not only can the subpaths corresponding to two repetitions of a context-free state be followed in succession in the order that they originally appeared in P , but also in other orders. The other orderings result in additional paths besides P being added to M by MB. This process is illustrated in the following example. Suppose that

$$P = Ax Bx Cx Dy Ey F$$

The MB will add

$$Ax Bx^*, Ax Cx^*, Ax Dy Ey^* \text{ and } Ax Dy F.$$

The common prefix Ax will cause branches B and C to be rooted at x with a loop-back to x . This makes it possible to follow an initial path segment $Ax Bx Cx$. The prefix will also cause $Dy Ey^*$ to be rooted at x . F will be rooted at the y after D . This makes it possible to follow $Dy Ey$ after x , giving us $Ax Bx Cx Dy Ey$. Since E loops back to y , we can then traverse F , to get $Ax Bx Cx Dy Ey F$.

QED

Theorem 6 MB is sound, i.e. suppose that we have a model M , and a path P that we add to M , using common prefix overlay. Let M' be the new model. Then the set of paths in M' is contained in the context-free completion of the set of paths in M and P . i.e. in $\text{paths}(M) \cup \{P\}$.

Proof When a path P is added to M , using common prefix overlay, it will start at the common start root node and follow a path of M until it branches off. If it does not branch off at some point, it will simply correspond to some path already in M , so no new paths Q will be formed.

Because this is a complex situation, we will break down the proof into several simpler cases before considering the general case.

a) Case 1: the path P that is added is not a loop-back path. This means that when it is added to M it is a path that starts at the common root node of M and terminates without looping back on itself. Clearly, if P is not a new path, it will just overlay a path in M , and there will be no new paths in M' . So, suppose that P is new. This means that it will start with the common root node for M , follow a path(s) in M , and then at some point split off. Because of the tree structured nature of M , after this it will not overlay any part of M . Suppose that there is an additional new path Q in M' , besides P . In this case, Q must do the following. If it just runs along a path from M , it will not be new. Similarly if it just follows P it will not be new. So at some point Q has to split off from P , and at this point it will run along a path in M . The only way it could be different from paths in M is to eventually loop-back to a state x in the part of P that is overlaid with the initial paths of M , to follow P in this overlay, and then, at some point when P splits off from the paths of M , follow P . If P has no loops, Q it will simply run to the termination of P . This means that we must have two paths of the following form, where R is in M

$$P = Ax DE.$$

$$R = Ax DCx^*.$$

Q follows P up to the end of D, and then follows C through M, after this takes a (last) loop-back x^* to x . At this point it must follow D and then take the E branch. It may not do this on the first loop back to the common part with P, but it has to do this eventually if it is not just going to be some path in M. This means it can be described as being constructed from a random walk through two paths of the form

$$\begin{aligned} R &= AxDCxDCx.... \\ P &= AxDE \end{aligned}$$

with a single cross-over from x in R to P. In the case where the crossover is at the first repeated instance of x , the addition of P will add an extra path of the form

$$Q = AxDCxDE.$$

From Theorem 3 we know that this will be a path in the context-free completion of P and R.

b) Case 2: P is a loop-back path, and M has no loop-back paths. P being a loop-back path means it has the form

$$P = AxBx^*.$$

Again, there will be an initial common overlay of P and the paths of M, up to the point where P splits off. P has a loop-back node x^* to an earlier occurrence of x in P. Suppose that x is not in the common overlay part of P and M, but comes after the common prefix overlay. Consider a path Q in M' that follows the common overlay up to the split. If it takes the branch that was contained in M, it will follow some path to its termination point and will be no different from the paths in M. If it follows the other branch, it will loop indefinitely around the loop in P with no chance for exiting, and will just be the path P.

The above implies that the loop-back target x for P must be in the common prefix overlay of P and M. This means that Q, if it is to be a new path, must follow the P path, traversing the loop some number of times, and then follow a path R from M to its conclusion in M. Therefore, Q is generated by following a random walk through the paths with the form

$$\begin{aligned} R &= AxCE \\ P &= AxCDxCDx... \end{aligned}$$

where Q starts by following P and then at some point does a crossover at an instance of x in P to the instance of x in R and then follows R when it gets to the split between P and M that occurs after C. From Theorem 3 we know that this path will be in the context-free completion of R and P, and hence of the paths in M and P.

c) Case 3: P and M may both have loop-back paths. Note that when we add P, we are actually adding an infinite set of paths if P is a loop-back path, corresponding to different numbers of traversal of the loop. Of course, we actually add a path of the form $AxBx^*$, where x^* is the "loop-back" to the previous instance of x on the path. Also, when we perform prefix overlay when we add a path P to M, we are overlaying a "bundle" of paths

in M that have common prefixes up to some point. The bundle will change as paths in M split off and are no longer part of the overlay.

The basic idea of the proof in the general case is the same as in the two simpler cases given above. Suppose that we have a path Q in M' that is not P and was not in the original model M . We can think of it as alternatively traversing paths in the set of paths consisting of P and the paths in M . It must switch from a path in M to P , or from P to a path in M at least once, or else it would not be a new path that is different from P and the paths in M . We argue that we can represent its traversal as a random walk.

Suppose that the addition of P to M results in a new path Q that was not in M , and it is also not P . Assume M' is the new version of model M that resulted from the addition of the path P to M .

Now because all paths in M come from a common root, they have an initial common overlay, so that Q will overlay both P and one or more paths R from M' . Follow Q along P/R until they diverge. At this point, Q will either follow P or a path R .

Suppose that it follows a path R . If multiple paths are overlaid, when we say we are following R we are actually following a bundle of paths. Assume we follow them all in parallel, so that R could be any one of them.

Because M' is a tree model with loop-backs, once Q starts following R it will continue to follow it until

- i) it loops back to a part of the tree where it is overlaid with P and then after having traversed some part of the common overlay with P , branches off from R and follows P , along a part of P that is not in the common prefix overlay with M .
- ii) it gets to a termination point in M and terminates
- iii) it gets to a sequence of one or more loop-backs on itself in M and continues looping around on itself forever.

If there is no R in the bundle of paths of interest that matches i) then Q does not differ from the paths in M or P . So case i) must hold for some R that Q is following.

This means that there is a loop-back target state x in the common overlaid part of P/M that is in the overlaid part before the split. We can define a path R_1 to be a path in the bundle of paths in M that Q is following, that takes us up to this state x . After this, think of Q as following P .

Assume that Q follows P , possibly looping some number of times in P . If it loops forever, Q is an infinite path that follows only P from this point forward, and Q is constructed from R_1 and P .

Suppose that eventually, after having followed P back to the part of P that overlays M , Q follows the common overlay and then branches off, following a path R_2 from M . Assume that x_2 is the loop-back state to which P returned in the common prefix with paths from

M. R_2 is some path with a common prefix overlay with P, so that the loop-back node x_2 is in R_2 also and we can think of Q has having crossed over from P to R_2 , which it is now following.

At this point Q could follow some R_2 that never makes it back to P. If it does go back to following part of the common prefix overlay with P, it must branch back to a loop-back state x_3 in the common overlay at some point and we can continue the argument as above.

In this way, we can construct a set of paths R_i from M and from P, such that Q does a random walk, crossing over at common loop-back states in the common overlay of P and M. By construction, all loop-back states in a model are context-free states. It follows from Theorem 3 that Q is in the context-free completion of the set of paths R_i and P. This implies each additional path, besides P, that is in the model M' that is constructed by MB when it adds a path P to M, is in the context-free completion of M and P.

QED

The above indicates that MB includes the original paths in a model that it builds from those paths, and that the paths in the generated model are all valid in the sense that they are in the context-free completion of the original set. But it does not tell us exactly what those paths are. In the following we show how this can be done for systems like DS, or any system for which the looping behavior is restricted as it is in DS. As in the above discussion, when we say x is a context-free state, we do not mean in the given partial model, but that we are assuming that it would be context-free in a complete model of behavior.

Definition Suppose that x is a context-free state, and that all paths P in a set of paths PS have the form

$$Ax(\prod B_i x)D \quad 1 \leq i \leq n$$

Then PS is a *single-level path set*.

We will show that if PS has this form then MB generates the context-free completion of PS, i.e. the total set of resulting paths are not just in the context-free completion, they comprise it. The theorem will be proved by defining a regular expression that describes the paths generated by MB, which also describes the context-free completion of PS.

Lemma 7.1 Suppose that we have a set PS of single-level paths of the form

$$Ax(\prod B_i x)D = Ax B_1 x B_2 x \dots B_n x D, \quad 1 \leq i \leq n,$$

where no subpath A, B_i , or D contains an instance of x . Let P be any path in PS. Some of the B_i in a path P may be the same, and different paths in PS may have initial subpaths in common. We assume they all have the common initial path A and terminal path D. Assume that the subpaths B_i in the paths in PS are constructed from the set of subpaths

A_j , $1 \leq j \leq k$, i.e. each $B_i x = A_j x$ for some A_j , and $A_j x$ appears in at least one path in PS.

Then the context-free completion of PS is equal to the set of all paths defined by the regular expression

$$Ax((\sum A_j)x)^*D, \quad 1 \leq j \leq k.$$

Proof

a) Choose some pair of paths from PS (which could be the same path), say

$$Ax(\prod B_i x)D \quad 1 \leq i \leq n, \text{ and}$$

$$Ax(\prod B'_i x)D \quad 1 \leq i \leq n'.$$

The context-free completion construction will combine the head of one path with the tail of the other to produce a new path. It will have the same form as the above. This implies that repeated applications of the context-free completion construction will produce paths having the same form. Each such path will be contained in the set

$$Ax((\sum A_j)x)^*D, \quad 1 \leq j \leq k.$$

b) Suppose that we have a path in the path set described by the regular expression.

We will give the argument for an example, which can be easily generalized. Suppose that

$$P = Ax A_1 x A_1 x A_2 x D$$

and that the regular expression of interest is

$$Ax((A_1 + A_2 + A_3)x)^*D.$$

If the regular expression has this form, then there must have been paths in PS of the form

$$Ax[F_1 x]A_1 x[G_1 x]D, \quad Ax[F_2 x]A_2 x[G_2 x]D \quad Ax[F_3 x]A_3 x[G_3 x]D$$

where the $[F_i x]$ and $[G_i x]$ terms stand for zero or more repetitions of terms in A_1 , A_2 and/or A_3 . We can use these paths from PS, with the construction operation for the context-free completion of PS, to construct the path P. In this case we only need the first two, since A_3 does not appear in P. P can be constructed by building it up as a sequence of intermediate paths Q_i , as below.

Let

$$H_1 = Ax \quad T_1 = F_1 x A_1 x G_1 x D$$

$$H_2 = Ax F_1 \quad T_2 = x A_1 x G_1 x D$$

$$Q_1 = Ax A_1 x G_1 x D$$

$$H_1 = Ax A_1 x \quad T_1 = G_1 x D$$

$$H_2 = Ax F_1 \quad T_2 = x A_1 x G_1 x D$$

$$Q_2 = Ax A_1 x A_1 x G_1 x D$$

$$H_1 = Ax A_1 x A_1 x \quad T_1 = G_1 x D$$

$$H_2 = Ax F_2 \quad T_2 = x A_2 x G_2 x D$$

$$Q_3 = Ax A_1 x A_1 x A_2 x G_2 x D$$

$$H_1 = Ax A_1 x A_1 x A_2 x \quad T_1 = G_2 x D$$

$$H_2 = Ax F_2 A_2 x G_2 \quad T_2 = x D$$

$$Q_4 = Ax A_1 x A_1 x A_2 x D$$

The basic idea in above is the following. The regular expression was built from the elementary subpaths that appear in the paths in the path set PS. We know that each such elementary subpath must appear in at least one path in PS. We use this fact to choose a "basis set" of paths. Then from this basis set, we can use the context-free completion construction to build any path, such as P, that consists of repetitions of subpaths in the basis set.

In the general case, we identify the basis set of paths that must exist in PS, and then take a random walk through the basis paths, as in the above example, to produce the target path.

QED

This means that we know that the context-free completion of the set of paths PS, is equal to the set of paths described by the regular expression:

$$Ax((\sum A_j)x)^*D, 1 \leq j \leq k.$$

Now we need to prove another Lemma showing that this set describes the paths in a MOCA MB model. If this is proved, then we know that the MB model generation algorithm produces the context-free completion for any set of paths PS having the given form.

Lemma 7.2 Assume that PS is a set of paths of the form

$$Ax(\prod B_i x)D \quad 1 \leq i \leq n,$$

which is constructed from the set of subpaths

$$A_j, 1 \leq j \leq k, \text{ i.e. each } B_i = \text{some } A_j.$$

Then the set of paths PS' that is described by the MOCA model that is generated from this set using MB is equal to the set of all paths defined by the regular expression

$$Ax((\sum A_j)x)^*D, 1 \leq j \leq k.$$

Proof

First we prove that each path in the set described by the regular expression will be in the MOCA model, and then that each path that is in the MOCA model is in the regular expression set.

a) Suppose that we have a path P in the regular expression. It will be of the form

$$Ax(\prod B_i x)D, \quad 1 \leq i \leq n, \text{ where each } B_i \text{ is equal to some } A_j, \text{ with repetitions allowed.}$$

As in Lemma 7.2, this means there must be paths in PS of the form

$Ax[F_j x]A_j x[G_j x]D, \quad 1 \leq j \leq k$ (where some of these might be the same path, just written differently). When this path is added to the model by MB there will be a branch from the x after the initial x (i.e. Ax) to A_j , and then a loop-back notation x^* back to x. This means we can traverse M, generating a path, as follows: start with A, arrive at x, then choose the A_j , and loop-back to x, as necessary to form P, giving

$$Ax(\prod B_i x)D \quad 1 \leq i \leq n.$$

Because of the form of the paths, each path ends with a tail xD , so there must also be a branch from the initial x, just after A, to D, which means we can tack it on to the end of the above partial path to get

$$Ax(\prod B_i x)D \quad 1 \leq i \leq n.$$

This shows that each path in the regular expression is also in the set of paths represented by the MOCA MB model.

b) Suppose that we have a path P that is in the set of paths described by the MOCA model that was constructed by MB from the paths in PS. Recall that the paths in PS are of the form

$$Ax(\prod B_i x)D \quad 1 \leq i \leq n, \text{ where each } B_i = \text{some } A_j, 1 \leq j \leq k.$$

The model, because of the way it is constructed, will consist of an initial root subpath of the form Ax , which then branches out to subpaths of the form $A_j x^*$ or $Ax D$. The x^* will

be the loop-back to the instance of x just after A in the model. Some of these will have common overlays, and in fact there could be duplicates that get completely overlaid. This means that any complete path generated from the model will have the same form as the paths in PS, given above, i.e.

$Ax(\prod B_i x)D \quad 1 \leq i \leq n$, where each $B_i = \text{some } A_j, 1 \leq j \leq k$
which means it will be contained in the regular set

$Ax((\sum A_j)x)^*D, 1 \leq j \leq k.$

QED

Theorem 7 Suppose that we have a set PS of single-level paths of the form

$Ax(\prod B_i x)D = Ax B_1 x B_2 x \dots B_n x D, 1 \leq i \leq n,$

where no subpath A , B_i , or D contains an instance of x . Let P be any path in PS. Some of the B_i in a path P may be the same, and different paths in PS may have subpaths in common. We assume they all have the common initial path A and terminal path D . Let PS" be the context-free completion of PS, and let PS' be the set of paths that are described by the model that will be generated by MB for PS. Then PS" = PS'

Proof

The set of paths that are in the context-free completion of PS is equal to the set of paths described by the given regular expression, which also is equal to the set of paths defined by the MOCA generated model for that set of paths.

QED

In the above proofs we restricted ourselves to paths of the form

$Ax(\prod B_i x)D \quad 1 \leq i \leq n,$

which corresponds to the case where we only identify a single context-free state, whose instances are denoted by x . This does not mean that there could not be other context-free states, which in our DS example there are, but that we only recognize one, in this case x , in forming the model. The purpose here is to find a way to introduce loops into the model, and to identify situations in which we can expect the model to be context-free complete with to the chosen loop-back state.

Abstraction and Context-free Behavior

As mentioned above, context-free behavior, or even isolated, equivalent state instances, is important in the construction of models. At some point, during a program usage scenario that results in repeated occurrences of some program state, the tester may want to conclude that equivalent behavior is being exhibited and put a loop in the model. One of the values of model abstraction, like that found in MOCA WA (windows abstraction) models, is that it can transform a non-context-free situation into a context-free one, allowing the use of model representations such as those defined by the loop-back notation.

This can be illustrated using the DS example. In the DS system, after some function such as adding a new member has been performed and completed, the program displays the DatingSystem.3 window, which has the Start and End buttons. Intuitively, there is a

loop-back to the initial starting state of the system. But it would not be correct to generate a MOCA model having this structure unless this state really was context-free. It turns out it is not context-free at the AR level, but is at the WA level. The reason for this is the Login window, and its local behavior. When a user types a name into the edit box in the Login window it is retained so that the next time the Login window is displayed the edit box will still contain the previous name. This has the following effect. Consider the instance of this state that appears for the first time on a path (i.e. with a context consisting only of the DatingSystem.3 state and its Start button event). If the user does not type anything in to the edit box, and clicks the Enter button, then the subsequent behavior will consist of a Message window (which will contain an unauthorized user message). Call this case 1. Now consider a different context in which: the user types in a legal dating system member name, a Member Functions window is displayed, the user chooses the get-a-date function, the user selects a date, the system returns to the DatingSystem.3 window, the user clicks Start, and then the Login window appears for the second time. Call this case 2. If the user simply clicks the Enter button at the end of case2, without entering any text, then the local behavior will not be the display of the Message state as in case 1, but will be the same Member Functions window as before. This indicates that the Login state has different local behavior in these two contexts.

Figures 6a and 6b contain AR abstraction examples. Figures 7a and 7b contain WA examples. Figures 6a and 7a correspond to Case 1 above, where no entry is made in the Login window before the Enter button is clicked. In the scenario for 6b, all we did when the second Login appeared was to click the Enter button. In forming the WA model in 7b, before we clicked the Enter button, we backspaced over the contents of the edit box, which caused the second occurrence of the Login state in 7b to produce the same observable behavior as in 7a. This is because the actions we performed "inside" the Login window do not appear in the WA abstract model.

It may appear that this is cheating, that in 6b we could have done the same backspace operation for the AR model and then get the same effect: clicking the Enter button after doing this will cause a transition to the Message behavior as in Figure 7b. The problem is that these backspace actions will show up in the AR model, appearing in its behavior between the Login state and the Enter button click, and the local behavior of the Login state in the AR model would still not be context-free.

Additional analysis indicates that at the WA level, the Login window has context-free local behavior, as do the other states, which implies by Theorem 2 that the DatingSystem.3 state has extended context-free behavior, and when it appears for the second time on a path P, it is valid to apply the loop-back operation.

DEFECT CATEGORIES AND THE MOCA WIZARDS

The MOCA MB tool abstracts and adds Robot traces to a model M. It automatically performs common prefix overlay. The theorems in the theoretical foundations section ensure that this results in a valid model. A user can identify selected states as being context-free and when MB detects a repeated instance of such a state in a path, it inserts a loop-back state x^* in the model. A critical question is: how does a user know that a state

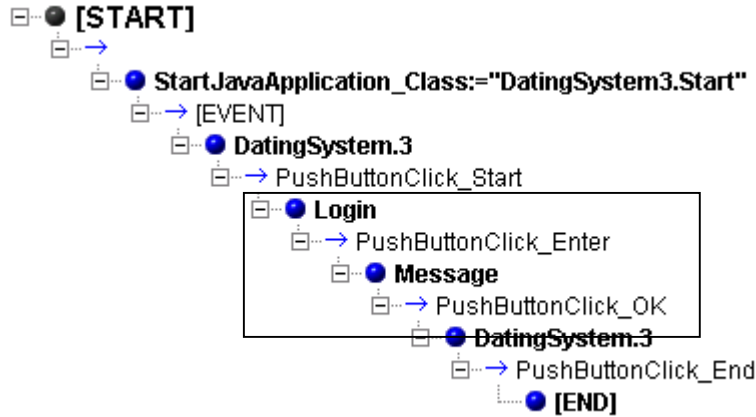


Figure 6a. AR abstraction - no data entered

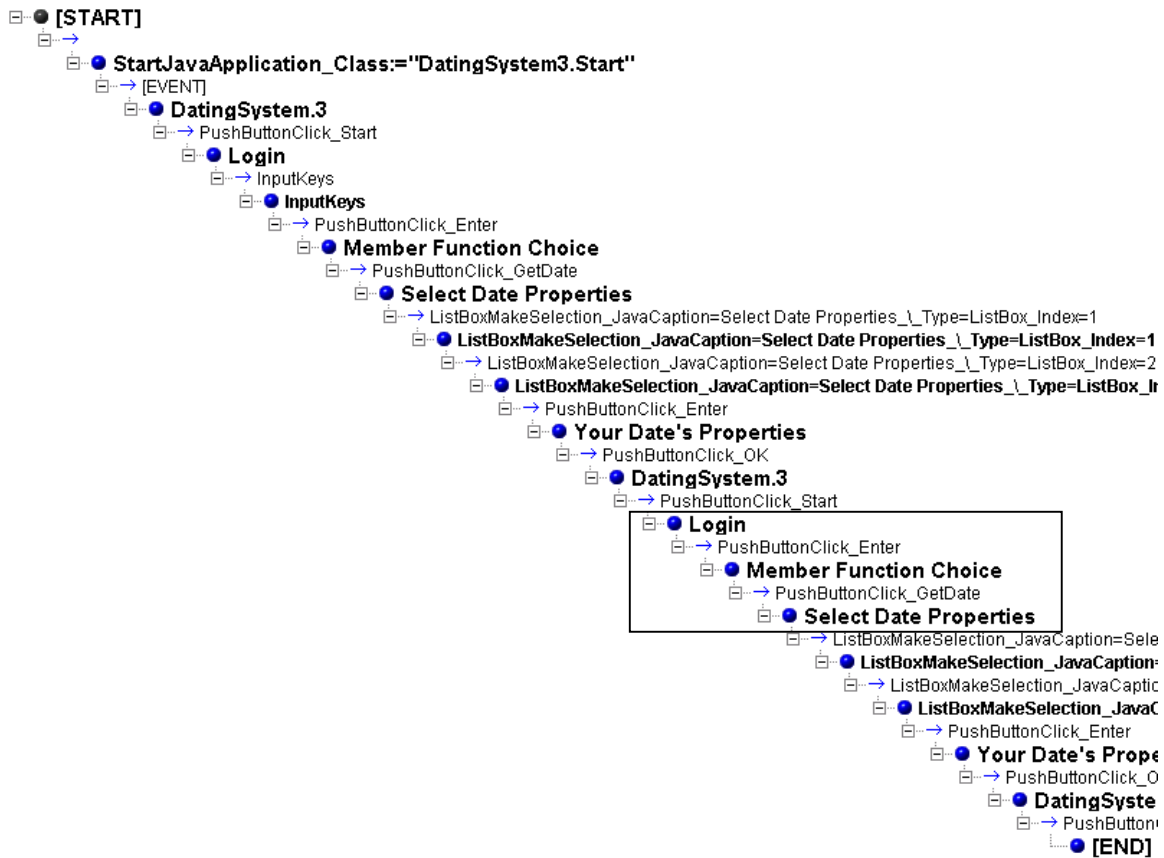


Figure 6b. AR Abstraction: no data entered in the second instance

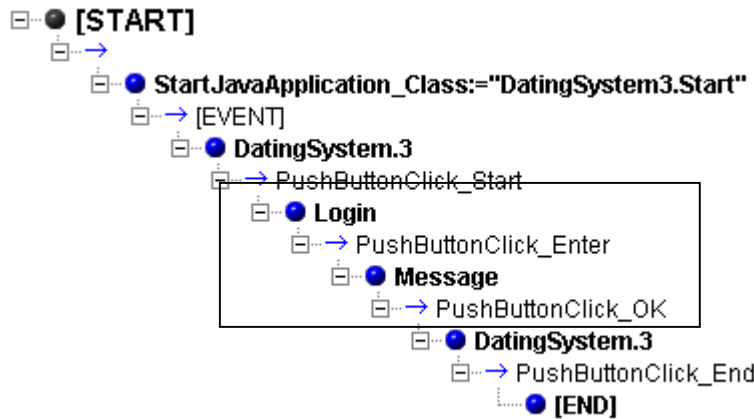


Figure 7a. WA Abstraction: no data entered in first instance

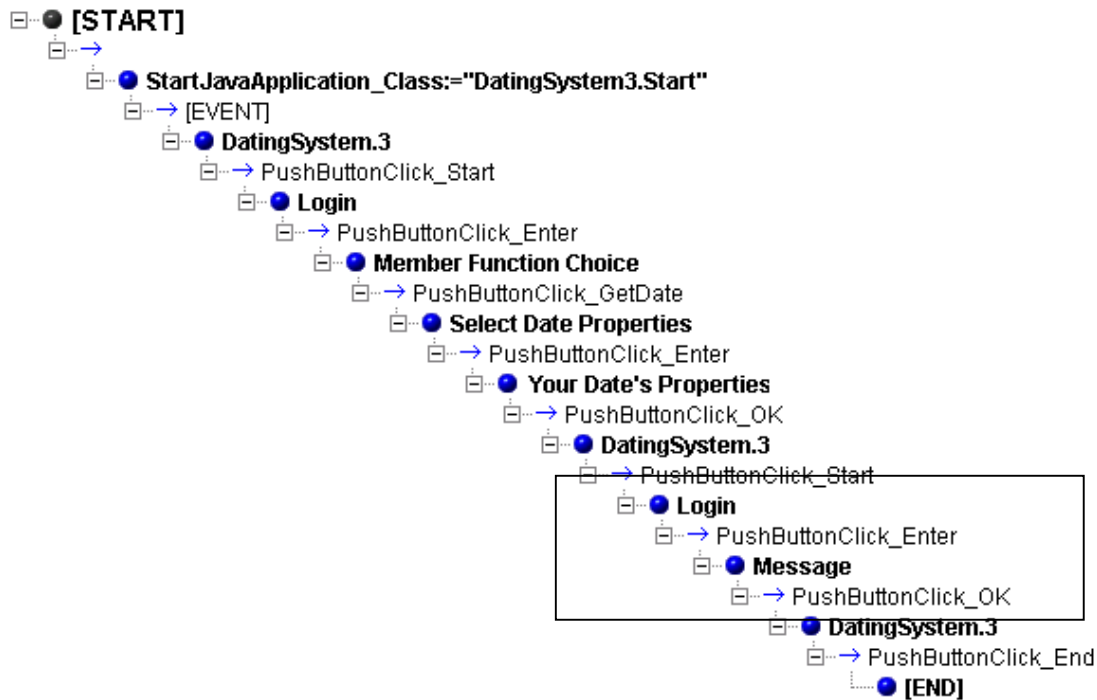


Figure 7b. WA abstraction - no data entered in second instance, with hidden backspace/erase actions

is an instance of a context-free state? Suppose, for example, we assume that an instance is context-free, resulting in a loop-back in the generated MOCA model, but it is not context-free, and we should have continued testing down some path where additional, different behavior could occur. Another critical question has to do with special cases.

When using MOCA to build a state model, the tester will generally focus on testing new functionality that produces new parts of the model. What about defects that produce failures for some of the uses of a previously tested functional capability, but not all of it? This is part of the more general question of how to develop a testing strategy that incorporates MOCA MB, and its underlying definitions and theory.

In the MOCA testing paradigm, we divide defects into the following three informal defect categories.

Defect Categories.

i) A *functional defect* is one where a program fails due to some basic function that is explicitly referenced in a model for the program. In the case of a state model, the function is associated with some state or event, so that whenever the program exhibits behavior corresponding to that part of the model, the function is being performed. In addition, it is assumed that the defect is such that the function fails every time it is encountered. Defects like this can be discovered if all of the states and transitions in a model are tested (which automatically occurs in the MOCA approach, where the model is generated from traces).

Functional defects like this tend to show up during development since they often correspond to functions that will be covered during developmental testing.

ii) A *special-case defect* is one that involves some kind of special value, such as a boundary or extremal condition. These include the familiar defects in which a program fails for an empty file, or on data that causes a loop to be executed zero times.

Examples of special-case defects that showed up during the testing of the DS example include the case where, if the user fails to make a choice for one of the desired datee properties (i.e. religion or gender), the system transitions to an illegal next state.

iii) A *combinational defect* is one in which a (usually rare) combination of conditions has to occur in order for the defect to cause a system failure. For special-case defects, it is reasonable to construct general-purpose rules that require that certain kinds of tests always be carried out when certain kinds of related data structures or other kinds of program entities occur. In the case of combinational defects, the construction of general rules based on these defects seems like a fruitless endeavor because it will result in many test selection rules that are seldom applicable and seldom effective.

Examples of combinational defects that showed up during the testing/usage of the DS example included the case where, if the user tries to delete a non-existent member during some session where this deletion attempt is not preceded by a (successful) deletion of a valid member, then the system fails. This fault, which is caused by certain peculiarities in data structure indexing arithmetic, does not cause a failure during a session if an attempt to delete a non-existent user is preceded by a valid deletion.

Defect Wizards

i) *Functional wizards*. In the traditional model-oriented approach to testing functional defects will be tested by constructing tests that "cover" the specifications model. This is not relevant for the MOCA approach: our functional tests will cover the model because it was generated from them. What is more relevant is whether or not the model is complete and accurate. It will be incomplete if it is missing functionality that is contained in the current (partial) system from which it was constructed. It will be inaccurate if it contains paths that were generated by MOCA MB, based on assumed context-free behavior of certain states, that are not consistent with the actual system.

In the MOCA paradigm a functional test wizard would help the user find these kinds of functional defects. Suppose that MOCA has been used to construct a program model for a current, possibly partial, implementation of a system. We may want to ensure that the model is "complete" in the sense that there are no branches from state instances in the existing model to other state instances that are not yet in the model, but are part of the current (partial) implementation. A functional wizard need to have some way of determining this. One possibility is building a wizard that could examine the code, and in the case of a GUI-based system, look to see what windows can be opened from a window. This would allow it to give suggestions as to possible missing components in a WA model.

This kind of wizard could also be used to attack the problem of false context-free assumptions. Suppose that a state occurs for the second time on a path, which has been declared to be context-free. A functional wizard might be able to detect additional branches that are possible from the second instance of the state that have not yet been explored, and which may be different from the observed local behavior of the first instance. If the wizard discovered this, it might warn that the state may not be context-free. This feature would only be useful at a point where a model was considered complete with respect to the current (partial) system. Earlier, before the tests have been run that will incrementally build up the expected behavior, it would be constantly giving false warnings.

ii) *Special-case defects wizard*. In this case, we assume that we have completed the functional model (at least for some stage of development) and now want to see if there are additional special-case defect-oriented tests that we might want to carry out. The wizard could determine this by analyzing the model, the underlying tests (i.e. concrete Robot test traces), and possibly the program code. If it found, for example, that for some edit box in a window/state there was no test that involved an instance of that state for which that control was not used (e.g. had no text entered) it could bring this to the attention of the user.

iii) *Combinational defects wizard*. Combinational defects could be dealt with using a *legacy defect* approach. The idea is that we will keep track of combinational defects that have occurred in the past. This could be done on a functional basis for different kinds of applications, or on an application basis for the sequence of versions of a given system or

application. The tester is expected to examine the associated legacy defect databases and manually construct a test for each relevant defect. Combinational defects are treated in the same way as special-case defects: we assume that we have a functional model of a program, and we are looking to see if there are defects corresponding to some aspect of that model.

MOCA MB Wizards

Wizard functionality in the current MB tool is limited to information that can be generated by modifying Robot traces to collect information while they are being run by Robot. Specifically, it is possible to insert VB commands, the language of Robot scripts, into the scripts to determine the child-controls used in a window's definition. This information can be used to construct a simple prototype wizard.

Suppose that we have a state/window W for which the instances in a model are W_i , $1 \leq i \leq n$. Let P be the set of possible child-controls that could be used in W , and for each instance W_i , let U_i be the set of actual controls that were used in that instance. Suppose that we compute

$$N_i = P - U_i \quad (1)$$

for each instance W_i for some window W . If the set for some instance is non-empty, then the user may want to consider the use of the other controls, which in this context may cause additional branches to other states/windows, i.e. there may be other possible paths in the model.

Alternatively, suppose that the following is computed

$$A = \cap (W \sim U_i), 1 \leq i \leq n. \quad (2)$$

This will consist of all the controls in W that have not been used in some instance of W , i.e. there is no instance of a window in which they have not been used. If one of the kinds of special-cases we consider is that in which a control was never used, then this set will suggest possible special-case tests that should be carried out.

The current tool includes the following simple wizard. The user can click on any state instance in a model. The wizard will then display the set of controls that are in the child set of controls for the window, plus the set of controls that were and were not used in that instance. This allows the user to play the role of functional wizard by stepping through the states, looking for possible un-followed branches and subsequent window/states from an instance of a state. It also allows a user to play the roll of the special-cases wizard by looking at the instances for a given window/state, and computing formula (2) in order to identify possible additional, special-case tests to carry out.

The current MOCA MB tool does have a simple combinational wizard, built around the idea of legacy defects. This wizard requires that a file of prose defect descriptions be given. The user is then expected to scroll through these, looking to see if any are relevant to the system under test. If so, the user can construct a test and cause the resulting trace that is generated to be stored in a special file along with the prose description that describes the legacy defect that motivated its use. As in the case of the other two wizards, the user can have the trace added to the current model

When the tester uses the wizards, and constructs a new test based on the information they return, MOCA MB does not automatically add the trace for a wizard-motivated test, but leaves this up to the discretion of the tester. There are 4 possible outcomes when a new test is run, summarized in the table in Figure 9. In case 1, the new trace should be added to the model. In case 3, if the behavior is incorrect, then it may be that the goal will be to correct the program and not to add the trace unless it reveals a newly discovered state that should be added to the model. In cases 2 and 4 adding the trace will not change the model, so the question is irrelevant.

	New state instance or event	No new state instance or event
Correct behavior	1	2
Incorrect behavior	3	4

Figure 8: Model augmentation cases

MOCA MB Details

In this section we give additional details about the current MOCA MB prototype, and describe additional capabilities of the tool that have not been described earlier. In addition, the user interface for MOCA is described, and a section on MOCA internals is included.

Additional MOCA features

Manual model construction and editing

In the above discussion, an MB model was described as being built automatically from Robot traces. MOCA MB also allows the user to manually build an AR model, and/or edit an existing model. This has to be carefully done, since the tester has to use the exact format of the traces.

Trace illumination

Suppose that a MOCA model has been constructed from a collection of traces. The user can identify any of the traces that were used, and ask to see the corresponding (abstract) path in the model. This will "light up" the path in a different color from the rest of the model.

Test initialization

Suppose that the user looks at a model and determines that he or she wants to try a new test whose trace will have an initial segment in common with a one or previous tests whose traces were used to build the model. If a state/window instance is identified, MOCA is able to use the previously generated Robot traces to automatically generate an initial trace segment up to that model state, which can be run to jump-start a test at an intermediate point. This facility may fail if the initial conditions, e.g. DS database, have changed since the trace was first generated and added to the model.

Control set abstraction (CSA)

We have focused on two kinds of abstract models that MOCA can build: WA (windows abstraction) and AR (abbreviated robot). We also experimented with an a third kind of model abstraction called "Controls Set Abstraction"

CSA is midway between AR and WA. It has two kinds of states/states. One is the same as in WA, corresponds to a GUI window, and is labeled with the window caption. The other summarizes the controls that were used while the user was "working in a window". This abstraction was not found to be as useful as the AR and WA abstractions, but the underlying tool functionality that it depends on is similar to what was needed to implement the MOCA MB version of the functional wizard, so it is a kind of side-effect benefit of those capabilities. In fact, at this time, the functional wizard can only be used with the CSA model.

MOCA MB User Interface

Basic MOCA MB tool use

The MOCA tester uses Rational Robot to carry out one or more tests, each of which will result in a trace that will be stored in a trace file. MOCA MB can be used to select traces from this file, abstract them as necessary, and add them to a (partial) MOCA model. The model is represented as a tree, using prefix overlays, plus loop-backs for instances of windows/states that are repeated in a path and which have been previously identified as being context-free.

MOCA MB Commands

Figure 9 shows a typical screen shot for MOCA. It contains a WA abstract model that is derived from the three Robot Screens that are listed in the left hand side of the menu. Figures 10-14 describe most of the functions of that can be invoked from the MOCA interface.

The example in Figure 9 gives a fairly full picture of the functionality of DS and indicates the power of MOCA models in giving a concise picture of a program's possible behavior. In this case the model was formed from four tests: administrator adds a member, administrator deletes a member, member asks for date, and member sets his or her member data. DatingSystem.3 was indicated to be a context-free state, so that MOCA MB generated a model that includes the context-free completion of the behavior observed on these four tests, with respect to this context-free state.

MOCA MB Internals

A Robot script consists of a sequential set of commands recorded in a language that is similar to Visual Basic. This language includes statements that denote: window get focus, button click, edit field selection, and key press events. MOCA MB parses this script and derives states and events from the various statements in the script.

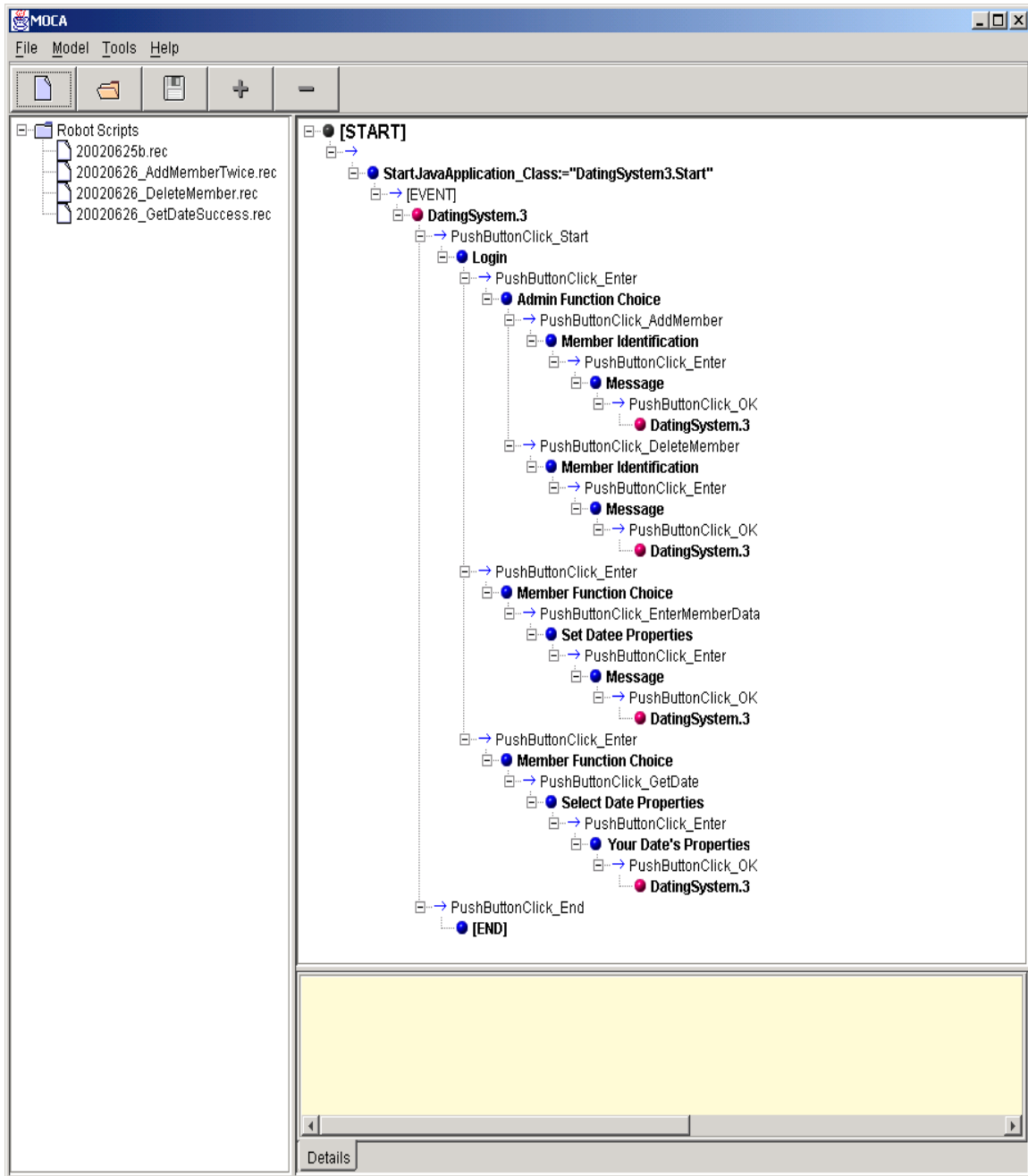


Figure 9: Typical MOCA MB Screen Shot

The statements in a Robot script are sequentially organized, which makes it easy to determine adjacent states because they just follow one another. The general rule for Window Abstraction is: "Each window is a state". This means a state is created for each window that gets focus (by clicking in the window or performing some other action within it).

Operation	Description
New	Begin a new tree model. This removes all states and events from the on-screen tree model.
Open ...	Open a previously saved tree model. The file loaded must be in the MOCA-specific format.
Save	Save current tree model. The current tree model is saved to disk.
Add Path ..	Convert Robot Script and integrate it into existing AR model
Add Path with context-free state	Create a set of paths using context-free completion procedure and integrate them into existing AR model
Accumulate Robot Scripts ...	Allows user to specify a directory and have each Robot script in the directory integrated into a model
Exit	Exits the MOCA application

Figure 10: File Menu

Operation	Description
Set context free state name	Allows user to specify the name of a state that is to be treated as being context-free
Abbreviated Robot	Displays AR level abstraction for current model
Control Set	Displays CSR level abstraction for current model
Windows Abstraction	Displays the WA level abstraction for current model

Figure 11: Model Menu

Once the sequence of states is identified then the events that transition from state to state are parsed from the actions that occur in between window statements.

Internally, MOCA maintains a list of the Robot traces that have been generated by the current set of tests. For each trace that has been added to the MOCA model(s) an abbreviated robot abstraction (AR) of the trace is generated and stored. If the user asks to see the model at a certain level of abstraction, it uses the AR paths to create a set of paths at the appropriate level of abstraction, which are used to construct the model.

Operation	Description
Set Robot Project Parameters ...	Set parameters need to run Rational Robot from within MOCA. Parameters needed to run MOCA wizard
Execute path as Robot Script ...	Used to replay previous script to allow a new test to start at an intermediate model state
Functional Coverage Wizard ...	Used to run the Test Wizard for a selected path or all paths in the model

Figure 12: Tools Menu

Operation	Description
About...	Displays version and contact information.

Figure 13: Help Menu

Operation	Description
New	Begin a new tree model. This removes all states and events from the on-screen tree model.
Open...	Open a previously saved tree model. The file loaded must be in the MOCA-specific format.
Save...	Save current tree model. The current tree model is saved to disk.
+ (Add New Event and Next State)	Adds a new event and a next state to the state that is highlighted. Unique, numbered names are chosen by default, i.e. "New Event 3" and "New State 3".
- (Remove Event or State)	Removes the highlighted event/state. Given states u, v and an event e from u to v : if either v or e is highlighted then v and e are removed; in addition, any child states and events of v are removed.

Figure 14: Tool Bar Operations

There are three basic model-building steps that are carried out for each AR path

- i) abstract the AR path (if required) to get the CS or WA abstraction P'
- ii) if there are context-free states in P' , apply the MB path computation procedure in order to generate the paths in the context-free completion for P'
- iii) add the paths generated in P' to the current model representation, using common prefix overlay

The two basic algorithms used here, context-free completion and common prefix overlay, are described below.

Context-free completion

If a path has a repeated instance of a context-free state, then we put a kind of loop-back to that state in the path, as in the loop-back derivation procedure defined earlier in the Theoretical Foundations Section. In our current representation, this is done by coloring the loop-back node red, which indicates there is an implied loop-back to the previous instance of the state. In the paper such nodes were also underlined, and in the text we have denoted loop-back state instances using the notation x^* .

As has been emphasized throughout the paper, the construction of a context-free extension to a model may add new paths to the model. Suppose we have an initial model containing a single path $P = AxBwC$, where A,B, and C are states, and x and y are nodes. Now suppose we add the new path $Q = Ax Dx E$, where x has been indicated to be a context-free state. In the loop-back approach, we will terminate the new path at the second x, and put a red x-node to indicate a loop-back to the first x node. This means we will now have to create a subpath xE at the first x, in order to keep the tail end of Q that comes after the second instance of x. But this means we now have a new path $R = Ax E$, which was not in the original set of paths. If in fact, x is context-free, then we know that this is a valid path, and the test associated with path Q has already covered this path.

Figure 15 contains an algorithm for constructing the context-free completion for a path P. This algorithm may result in multiple copies of the same path in the set PS. This can be checked, or we can just count on common prefix overlay to remove the extra paths.

In the algorithm of Figure 15, PS initially consists of the path P. The algorithm will add the paths in the context-free completion of P to PS. Assume that P has start node S. In the algorithm, states are considered the same if they are instances of the same context-free state.

Common prefix overlay

The algorithm in Figure 16 is used to form a tree from a collection of paths. There are several possible approaches to this. For example, we could represent the paths in the order in which they were generated by the algorithm in Figure 15. Alternatively, the paths could be sorted in order to maximize the overlaid parts of the tree. In MOCA MB, the first strategy is followed since it adds the paths to the model one at a time, as they are produced by the abstraction and loop-back derivation procedures for a path. It is important to note that since we allow multiple copies of the same state (i.e. nodes with the same label) it is not necessary, or even correct, to overlay all nodes with the same label, only those in a common prefix overlay, which makes it possible to maintain the basic tree-like structure of the model.

Suppose we have a (partial) tree model M, and a path P to be added to M. Let S be the first state in P and T the first state in M. We assume that all paths start from a common state, so that S and T are the same state and can be "overlaid". Figure 16 contains a common prefix overlay algorithm.

```

{Let S' = S
while (S' is not the terminal state in the path P)
    {
        let e be the branch out of S' and S'' be the
            state at the end of e;
        if (there is a previous state S''' in P that is the same as S'')
        {
            add the path from S to S'' to PS, with S'' colored red;
            let P' be the path from S to S''', plus the tail end
                of P starting after S''
            reset S' to S''' in P' (which is now the head of the subpath that
                started at S'' in P);
            reset P to P'
        }
        else
        {
            reset S' to S'';
        }
    }
add the path P to PS;
}

```

Figure 15: Context-free completion

```

{
while (there is an event e from S in P and event f from T in M)
    {
        if (there is no event/next-state pair f/T' from T for which e/S' = f/T'
            /* i.e., they have different labels */) break;
        else {
            choose the first matching event/state pair
            {
                "overlay" the event/state;
                reset S and T to S' and T';
            }
            else break;
        }
    }
while (S is not terminal)
    {
        e = the event/transition out of S;
        S' = the state at the end of event e;
        add a new branch with label e to the model from state T
        and put a new state equal to S' at the end of the branch;
        reset T to be the new state in M that was equal to S';
        reset S to S';
    }
}

```

Figure 16: Common Prefix Overlay algorithm

RELATED WORK

Model-based Testing

As mentioned earlier, model-based testing usually depends on the availability of a specifications/model, on which tests are based. Although MOCA takes a different approach, in which it is assumed that no such model exists, they both depend on relating tests to models. We review here a very small sample of the previous work in model-based testing.

A general survey of the basic ideas of model-based testing can be found in [13]. The automatic generation of tests from models is explored in [5]. The application of the ideas to systems of components is explored in [16], in which it is assumed that a finite-state model describes each component. The product of the individual models is then formed to construct a global model, which facilitates the generation of integration tests.

Although model-based testing usually refers to state models, other kinds of models may be involved in this general approach. In his book [2], Bender identifies a variety of models that are involved test data generation. He emphasizes the interpretation of the models as being implicit or explicit fault models. This is related to the way that, in the MOCA paradigm, we first construct a finite-state model that is then used by the wizards to suggest tests that are based on kinds of faults that might be associated with that model.

In [7] the authors describe a data-oriented interpretation of the phrase "model-based testing". In this case, the model is patterned after an experimental design. It is assumed that there are a number of factors, which may correspond to variables, plus a number of levels for each factor, which may correspond to classes of variable values. In addition, constraints on possible combinations of factor levels can be given. A tool then generates (valid) combinations of factor levels in such a way that every level in every factor is paired with every level in every other factor in at least one combination. The tool chooses the combinations efficiently, which are used to construct test data. Further work on MOCA could consider other kinds of models, such as experimental design models, as well as those covered in [2].

Model Abstraction

One of the principal components of the MOCA paradigm is model abstraction. The approach we have followed here, that an abstraction is a state model whose paths map to equivalence classes of paths in a more concrete model, was formulated to serve the needs of the MOCA work. Different concepts of model abstraction have been proposed in the past.

Perhaps the most common idea of abstraction is that of submodels, in which a super-state in an abstract model can be refined into a submodel that refines the superstate. This idea occurs in different forms, both for state models and for other kinds of models such as data flow diagrams. Sub/super state abstraction was formally explored in an early paper on state model abstraction by Courtois [9].

Other work has been concerned with abstraction for specific areas of application, as in the case of simulation models. In this case, both state transition and timing are important properties of a model. In [28] Sevinc briefly reviews work in this area and describes a procedural approach in which state procedures observe detailed behavior and determine the transition, output, and time characteristics of the states of a more abstract model.

More recent work on traditional model-based testing includes the results described by Farchi *et al* in [15]. In the approach described here, test objectives are defined using projections of a given model. Each state in a model is assumed to be defined by a set of values of the state variables. A projection of a state is defined by naming specific values for a subset of the variables. In the resulting state equivalence class, the unspecified variables are allowed to take on any value. An abstract state machine A is formed from a concrete state model C by first constructing the projection states, and then by adding transition arcs from an abstract state a_1 to an abstract state a_2 in A if there is a transition in C that goes from a state that maps to a_1 to a state that maps to a_2 . Tests are carried out by "executing" C , while observing which states in A are being covered. When some abstract test criteria, such as "cover all states in A " or "cover all transitions in A ", has been satisfied, test generation can be halted.

The first of the above approaches, in which abstraction is modeled as a process whereby submodels are shrunk to superstates in an abstract model, is similar to the abstraction concept used in MOCA. It guarantees the abstraction equivalence-class restrictions that are used in MOCA, i.e. each path in an abstract model must map to an equivalence class of paths in the concrete model. The abstraction procedure in [15] does not necessarily guarantee this property, in which state projection may result in an abstract model in which there are paths for which there is no corresponding path in the concrete model. This problem can be avoided if projection is limited, as it is in MOCA. In MOCA MB, we used what was called "contiguous path segment projection", in which each abstraction superstate results from the projection of a set of less abstract states that occur in sequence along a contiguous segment of a path in the less abstract model.

In addition to work that focuses directly on abstraction of a given model, abstraction also occurs during automatic specifications and model generation, as discussed below.

Automatic Generation of Invariants

The general idea of constructing specifications from traces is conceptually related to the idea of automatically generating program invariants. Much of the invariants work involves static analysis methods [e.g. 3,4,30], in which invariants are generated from source code. Other approaches use dynamic analysis in which the values of variables that occur during system execution are used to synthesize simple invariants. Reference [31] describes a kind of hybrid approach in which values of state variables that occur during a model-checking state-space search are used to construct simple invariants at model-state locations.

Examples of work in which invariants are constructed from program variable values during program execution include [14] in which simple invariants such as linear functions

in 3 or less variables, or arithmetic relations between 2 variables, were constructed from intermediate program states.

Automatic synthesis of state models

Automatic program invariant generation seems more closely related to program analysis at the unit or module level, for which input/output assertions are used as the specification, than system-wide functional analysis level, as in MOCA. At the subsystem or system level, we will be interested in generating specifications that are more abstract than the code and that use separate computational models rather than code annotations. Previous examples of this kind of work include [8]. In this case, the model generation process is static. In an initial phase, program slicing is used to extract the relevant parts of the code. After this, a user-guided process is carried out, that can perform abstractions such as replacing variable domains and data structure instances with abstract values such as {item-in-vector, item-not-in-vector}. Objects and object access methods can be replaced with abstract objects and abstract object access methods.. This kind of abstraction is similar to the kind of abstraction carried out in MOCA in going from the AR abstraction to the more abstract CS and WA abstraction models. The kinds of abstractions that are used here [8] are further discussed in [12].

In [6] Boigelot and Godefroid describe work that contained some of the foundational concepts that were re-discovered in MOCA. It included the following basic ideas: generation of a finite-state machine from a tree of traces, equating trace tree nodes that are roots of the same trace-subtrees, and interpreting a finite-state model as a test case summary. Some of the differences are: our application area was GUI-based applications - theirs was communicating processes; we used a COTS capture-playback tool (Robot) - they used a special purpose trace tool; our models have event-labeled transitions and behavior-labeled state nodes - their models have event-labeled transitions and unlabeled state nodes. More significant differences involve the way abstraction is treated and certain aspects of incremental model development.

In [6] two kinds of abstraction are mentioned. One is the simple act of using more general event labels. The other concerns the mechanism for grouping states when a finite-state model is formed. As in [15], equivalence classes of states are determined which are represented by abstract states. Unlike [15], the approach described in [6] retains all of the state transitions, using self-transitions when two transition related states are grouped into the same abstract state. The resulting, abstract, finite-state model will be a "simulation" (as in [25]) of the original concrete model. The abstraction procedure in [6] has the same potential problem for our approach as that in [15]: there can be paths in the abstract model that are not in the original concrete model. In our approach, the abstract model involves equivalence classes of paths, and not states. There are no new paths in the abstraction that do not have concrete realizations. This is important because paths correspond to functional use-cases and have to be preserved, even when they are abstracted. In the communicating processes example in the approach of [6], functionality is associated with communication events primitives, and the concept of path preservation may be less important.

It is noted that MOCA MB does introduce new paths in a model, due to context-free completion, but this is not part of the abstraction process, and in addition, the kinds of paths that are introduced are tightly specified: they are in the context-free completion of the original path set.

In [6] states in the trace tree are grouped into an equivalence class if they have the same subtree, up to some user defined level. The examples that are given commonly use a depth of 1, which would correspond to our concept of local context-free behavior. In our approach, it is assumed the user has some expectation of the operation of the system and guides the generation of the current model from the current set of tests by identifying a state as being context-free. Since, unlike the approach in [6], our states have labels, there is a way of independently identifying certain kinds of states that are expected to have this property. There is no direct way of doing this if, as in [6], the states are unlabeled.

The approach in [6], in which state equivalency is only recognized when states have common subtrees would not work for our approach, which focuses on incremental testing and model development. Consider the case where a system has a state *S* with multiple exit branches (having different labels) to the next state. Suppose the user runs a long test in which one of these exits is chosen and then a loop is taken back to that state and another exit is chosen, until all exits have been tested. This would result in a long trace in which *S* occurs repeatedly, each time followed by a branch with a different label. Since none of the exits from *S* in this trace would have common subtrees, grouping them together on the basis of their being the roots of equal subtrees would have no effect. Alternatively, in our approach, the user could assume (hypothesize) that *S* was a context-free state, which would cause the generation of a state model that collected instances of *S* into one state, with loop-backs for each of the exit branches. One approach to checking the validity of such a context-free assumption is to use a MOCA functional wizard, where source code information is used to check for possible unexpected branches from an instance of a state.

Finally, the MOCA approach seems to have a richer semantic content than that of [6]. For example, it permits the consideration of additional kinds of results, such as characterizing situations in which local context-free behavior implies extended context-free behavior. This result is not interesting in [6] because state nodes are unlabelled and the assumption that all nodes have local context-free behavior would mean that all event transitions in the tree had the same label, so that extended context-free behavior would be a trivial consequence. Another example is our proof of the correctness of the context-free completion algorithm. This occurs because we are able to give a specification of what an extended set of paths in a synthesized finite-state model would consist of. In the approach in [6], when a finite-state model is generated from a set of traces, extra paths may be an artifact of the synthesis process, and could disappear if a different set of traces were used.

More recent work on synthesis of state models from traces relates it to the kinds of traces used in modeling languages such as UML [e.g. 21,23,30,32]. This work is also concerned with distributed systems as in [6]. However, it uses sequence diagrams instead

of simple event/state traces, so just as MOCA traces have a richer semantic content than the traces in [6], the UML traces used in this work have an even richer content. We will first review this work, and then show how MOCA relates to it.

The construction of state diagrams from sequence diagrams, such as UML interaction sequence diagrams or MSC's (message sequence diagrams), involves several common themes. In all of the work, separate state models are constructed for each object/life-line in a sequence diagram. States are identified in different ways. One common theme is to associate states with intermediate points in a sequence diagram lifeline. The details of how this is done varies.

In [23], states are identified with object behavior. If an object, after it sends out a sequence S of one or more messages, receives a message, then it is considered to be in state S . In [30], a more fine-grained approach is used, in which each successive pair (m_s , m_r) of a sent message m_s and a received message m_r is identified as a state. If there are sequences in which sent and received messages do not alternate, a dummy message is inserted.

In [21], the MSC condition feature is used to allow the engineer to denote the places in an object message sequence lifeline where a state occurs. This has several beneficial effects. It allows the creation of states that have some independent meaning that may not directly correlate with message receipt and transmission. In addition, such condition-states serve as "anchor points" that show how parts of a model fit together.

In [32], states occur between each pair of messages, but their identity is established in a more complex way. Users are expected to construct pre and post conditions for each message. These are written using a set of declared global state variables, which are used to augment the sequence diagrams. The pre and post conditions for a message show its effects on the variables. A state vector describes the before and after properties of each state variable. Two state vectors are considered the same if they are "unifiable" in the sense that there is a set of values for their variables that causes them to evaluate to the same value.

More complex issues in the synthesis of state models from sequence diagrams are similar to those that had to be resolved in MOCA: how to recognize the occurrence of loops and how to merge multiple sequence diagrams into a single model. The related problem of over-generalization occurs for both endeavors: when new paths are introduced in the synthesized model how can we be sure that they are valid?

In general, in the work using sequence diagrams, loops are inserted into models when common states occur. Of course, this will depend on the technique for recognizing states and, in some cases, additional properties are involved. In [32] a loop is only inserted if a new state matches an earlier state and the new state involves a state change, i.e. its precondition is not the same as its post condition. This restriction reflects an attempt to halt false loops. In [23], an interactive approach is used, in which the analyst can accept or reject proposed structures, including those with loops. Facilities exist for guiding the

system in altering its synthesized model. This problem is not dealt with in [30]. In [21], loops are also formed when common states occur, but since states are identified by the analyst, and not automatically, the problem of false loops would seem to occur less often.

Suppose that several sequence diagram traces are given for an object. Then it is necessary to devise some procedure for combining them in a single, synthesized state model. A common approach in the sequence diagram work is to first construct a state diagram for each trace, and to then combine the state models. In [30] relationships such as sequence, disjunction and conjunction are identified, presumably by the user, and then the corresponding state-chart relationships are used to combine the individual object models. This approach does not co-mingle separate models for alternative courses of action, they are simply listed side by side as alternative flows in a disjunctive combination of submodels, one for each sequence diagram. [30] also contains suggestions on how to recognize state machine hierarchies. For example, if a disjunctively related set of submodels forms a kind of "cluster" then they can be joined in a super-state.

The exception to this approach, in which separately formed models are merged together, is described in [21] where the sequence diagrams have user-defined abstract states that can be used to fit different sequence diagrams together as they are being constructed. The approach includes an "optimization" phase in which the details of the initially formed diagrams are condensed.

In [23], the merging of models and/or sets of sequence diagrams is not discussed. In [32], states in models constructed from object lifelines in different sequence diagrams are merged if they are the same state, and they have at least one common incoming transition with the same message label. [32] also considers the structuring of diagrams into a hierarchy, suggesting several ideas, such as partitioning states based on state vector values, and using an interactive, approach as in [23].

MOCA has both similarities and differences with the work on finite-state machine synthesis from sequence diagrams. We first note that its process context is different. We are working from traces generated from executed code, rather than specifications. As mentioned earlier, one critical difference is our focus on incremental development, because of the association of MOCA with extreme testing [18]. This means we need an approach where we can easily add new traces and see the updated model. This led to the use of the MOCA MB tree-structured model, with loop-back nodes x^* , which is both easy to draw and to incrementally update. In this regards as in others, it is different from the graph-oriented models used in the other work. The use of the MOCA-style models affects each of the issues discussed above: state identification, loops, overgeneralization and correction, merging of models, and compound model structuring.

In the MOCCA approach, the equating of states with behavior indicates a similarity with the approach of [23], where a message arriving at an object corresponds to a state transition and the related messages sent out from the object correspond to the behavior that is exhibited by that state. With respect to the questions raised in [32] of identifying

abstract states that would ignore repeats of the same message, in MOCA this is dealt with using the model abstraction procedures. An abstraction for concrete traces would be defined that would generate a more abstract model in which these inconsequential details would be abstracted out. This is seen in both the CSA and WA models and how they differ from the original Robot trace and from the AR level of abstraction

In some ways, the treatment of states and events in MOCA is closer to [21], which contains the idea of states as an explicit separate entity, and not just an implicit side-effect of message event occurrences. In the application of MOCA to DS, for example, states are screen images that show the current state of the system. Events are messages that could cause transitions to new states, but both states and events are first-class entities.

The MOCA theoretical concept of a context-free state provides a well-founded systematic approach to the problems of loop recognition and overgeneralization. MOCA identifies repeated instances of the same state based on their expected behavior. But we recognize that two states with the same observed behavior are not the same state, they are instances of the same "state" whose equivalence may depend on their context, i.e. in some other context the behavior may be different. Given this distinction, we rely on the user to indicate when multiple instances of a state can be considered the same state (i.e. when a state is context-free) and a loop can be inserted into a model. Since the identification of states as being context-free can be easily changed, this approach is related to the interactive user approach described in [23] in which the user examines a generated model with loops and if it is not agreeable, information is input which causes this feature of the model to be altered.

MOCA avoids overgeneralization through its underlying theoretical foundations. If a state has been declared to be context-free, then we can formally specify the additional model paths S whose existence is implied by this property. We proved that the additional, valid paths that are created when the MB model-building procedure introduces loops into a model for such a state are either contained in or are equal to this set of paths S . This means that MB will always produce a valid set of additional paths and overgeneralization (relative to the validity of the context-free assumption) cannot occur.

The problem of joining together the information that comes from multiple traces is easily dealt with in MOCA, since it is specifically designed for incremental model update. We do not generate separate models for each trace and then merge them, since we incorporate (abstracted) traces into the model one at a time. If there were some reason for merging separately generated MOCA models, the basic features of the MOCA approach and the kinds of model that it uses would allow this to be easily implemented.

In the MOCA work, we have not yet examined the issue of compound models. By this we mean joining multiple models into a hierarchical state model, where one or more state models may be nested inside the state of a more general model. In the referenced work on synthesizing state models from sequence diagrams, this appears to be a topic that is still being researched, and the suggested approaches indicate some degree of user

involvement or guidance. MOCA MB includes hierarchical model facilities, but in the form of models at multiple levels of abstraction where the user can, with a click of a button, go from one level to another.

Extreme and Agile Testing

Extreme Programming (XP) [1] includes several testing themes. One is that tests should be developed before code, or code increments. Another is that testing should be automated. Perhaps the most relevant to the work described here is that the tests form a kind of specification for the current version/increment, given that it is both unlikely and undesirable that formal specifications will exist. A number of papers have been written that focus directly on the testing aspects of XP. MOCA can be viewed as a model-based "extreme testing" method, in which a model is built to keep track of the program behavior exhibited by tests, in a format that allows the model to be used as a specification for the current program increment/version.

A variety of papers have been written on extreme testing, but they focus on other aspects of XP testing. One of the earliest uses of the phrase "Extreme Testing" occurred in an article by Jeffries in [19]. In this case, emphasis is on the early development of unit and functional tests that are automated, and can be re-run when a change is made to the software. Functional tests are identified as test that check each of the functional increments that are added as the system develops. Test reporting is restricted to graphs of tests passed and failed.

In both Jeffries article and in articles on testing within the broader context of XP, [e.g. 10]] indicate the possibility that the acceptance/functional testing phase of a project may involve the user in order to validate the tests and test results. This is consistent with the MOCA approach, which makes it possible, to both re-play a set of tests for the ultimate users of customer, and to provide a visual summary of the tests, in the form of a MOCA model.

The subject of Extreme Testing and Visualization was discussed in [20, 22]. However, in this as in the article by Jeffries, visualization is limited to the reporting of test run results, such as failed and passed test percentages. The automated approach reported in this article includes other kinds metrics as well, such as coverage measures and amounts of time to complete test cases.

More recently, the general subject of what was introduced as extreme programming has been referred to as agile methods, and the phrase "agile testing" has appeared. In [24], Marick, uses this phrase to refer to a variety of issues, including XP practice of testers working alongside developers. In addition, he emphasizes the importance of an ongoing relationship between the testers and the customers. The phrase also appears in [29] where the author argues that XP is itself morphing into the agile testing methodology. In both references the emphasis is on early tester involvement.

SUMMARY, CONCLUSIONS AND FUTURE WORK

Traditional model-based testing presupposes the existence of a specifications model that is used for systematic test identification. It can be used to specify functionally complete test sets. Often such a model is not available. In the MOCA approach, tests are used to generate a model. The model

- i) summarizes the tests that have been run, and
- ii) provides a specification of a program, reflecting its observed behavior over a set of tests.

The idea here is that testing is complete when the test-summarizing model is complete. In order to assist the user in deciding if a model is complete, the MOCA paradigm includes the idea of a functional wizard that can be used to generate a kind of upper bound on possible behavior. The user is meant consider the possible additional kinds of behavior, and either construct an appropriate test or conclude that the wizard suggestions are not relevant.

MOCA models are constructed from abstract paths that characterize a program's behavior over a set of tests. In the MOCA project described in this paper, the paths were derived from traces that are generated using the Rational Robot capture-playback tool. Each time a program is run under Robot a trace of the program's observed GUI behavior is generated. The tester can then use MOCA to build an abstract model from the traces.

Robot traces may be too detailed to provide meaningful functional testing models. The MOCA tool is capable of generating models at three different levels of abstraction. The most popular at this point, the Windows Abstraction model, shows the behavior of a system in terms of the GUI windows that were used, and the events that caused transitions to new windows.

A program can contain different kinds of faults. Three kinds were identified, which are used to define a MOCA-based functional testing paradigm. The first, called *functional faults*, occur when some observable system functionality fails over all uses. These will be discovered during construction of a functional model, and the MOCA functional wizard that guides the user in the construction of a MOCA model can be interpreted as a wizard that helps a tester find such functional faults.

The other two kinds of faults are *special-case* and *combinational*. In the MOCA approach, wizards are also used to assist in the selection of tests for revealing these kinds of faults. The first includes the familiar kinds of boundary faults that are often the source of a problem in a program. The approach taken in MOCA is to interpret a current (partial) functional model that was built from the traces as a (partial) model of a program's functional capabilities. The task of the special-cases wizard is to identify the need for tests for certain kinds of possible defects in a program having that functionality. Combinational faults correspond to rare combinations of conditions for which general-purpose rules are unlikely to be useful. A *legacy defect* approach is taken for these kinds of faults, in which past defect histories for an application domain are used to suggest additional tests. Testers evaluate the entries in the legacy defect database and construct tests as appropriate. If relevant, their resulting abstracted traces are added to the MOCA model.

The MOCA project accomplishments that are reported in this paper include: a proof-of-concept design and implementation of the MOCA paradigm, and the development of a theory of abstract behavioral models. When the project began it was not clear what features would be needed in a MOCA-style approach to testing, and the development of and experiences with the MOCA MB tool were critical to the identification of key ideas such as the defect categories and the related MOCA wizards. It also led to the development of the common-prefix-overlay and loop-back model. The problem of how to introduce loops into a model, and the concept of a context-free state, became more thoroughly understood while designing and using earlier versions of the MOCA MB tool. It led to the discovery of the need for an underlying theory. Important aspects of the theory that was developed include: formal characterization of path abstraction and model equivalence; formal definitions for context-free states, for local and extended state/state behavior, and for context-free completion; analysis of the relationship between abstraction and context-free behavior; and theorems that establish completeness and soundness properties for the MOCA MB model builder.

There are several avenues for future research. One includes more experimentation with and development of MOCA MB. This tool is build to deal with traces that are generated by Rational Robot from Java programs with a GUI interface. At present, it only works for a subset of the different possible Java GUI controls, and is being extended to include more of them. Additional tools and related underlying theoretical foundations are being designed and developed for other kinds of applications. The concept of a testcase wizard, as introduced in this paper, is also a rich area for future research and investigation. Wizards for web sites, for example, might work off DOMs for web pages. Wizards for client-server and distributed systems will need additional concepts and techniques.

Acknowledgements The authors would like to thank Dr. Ingolf Krüger (CSE, UCSD), for helpful discussions on the MOCA theoretical foundations.

REFERENCES

- [1] Beck, Kent. *Extreme Programming Explained*, Addison Weseley, 2000.
- [2] Bender, Robert, *Testing Object Oriented Systems*, Addison Wesley, 2000.
- [3] Bensalem, S., Lakhnech, Y., and Saidi, H., Powerful techniques for the automatic generation of invariants, *CAV*, 323-335, July, 1996.
- [4] Bjorner, N., Browne, A., and Manna, Z., Automatic generation of invariants and intermediate assertions, *Theoretical Computer Science*, 173, 1, pp. 49-87, Feb, 1997.
- [5] Blackburn, Mark, Using Models for Test Generation and Analysis, *Proceedings, Digital Avionics System Conference*, IEEE, 1999.
- [6] Boigelot B. and Godefroid, P., Automatic synthesis of specifications from the dynamic observation of reactive programs, *TACAS '97*, 321-333, July, 1996.
- [7] Cohen, D.M., and Siddhartha R. D, Fredman, M.L., and Patton, G.C., The AETG System: An Approach to Testing Based on Combinatorial Design, *IEEE Transactions on Software Engineering*, Vol, 23, 7, pp. 437- 444, July, 1997.

- [8] Corbett, James, Dwyer, Mathew, Hatcliff John, Pasareanu, Corina, Robby, Laubach, Shawn, and Zheng Hongjun, Bandera: Extracting finite-state models from Java source code, *Proceedings of the 22nd International Conference on Software Engineering*, 439-448, June, 2000.
- [9] Courtois, P.J., On Time and Space Decomposition, *CACM*, Vol. 28, No. 6, pp. 590-603, 1985.
- [10] Crispin, Lisa, Extreme Rules of the Road, *Software Testing and Quality Engineering*, Jul/August, 2001.
- [11] Dalal, S.R., Jain A., Karunanithi N., Leaton J.M., Lott, C.M., Patton, G.C., Horowitz, B.M., Model-Based Testing in Practice, ICSE 99, ACM, 1999.
- [12] Dwyer, M. B., Hatcliff, J. Joenanes, R., Laubach, S., Pasareanu, C, Zheng H., Visser W., Tool Supported Program Abstraction for Finite-state Verification, *Proceedings of the 23rd International Conference on Software Engineering*, May 2001.
- [13] El-Far, Ibrahim K., Whittaker, James A., Model-based Software Testing, *Encyclopedia of Software Engineering*, ed. J.J. Marciniak, Wiley, 2002.
- [14] Ernst, Michael, D, Cockrell, Jake, Griswold, G., Notkin, David, Dynamically Discovering Likely Program Invariants to Support Program Evolution, *IEEE Transactions on Software Engineering*, vol. 27, no. 2, Feb. 2001, pp. 1-25.
- [15] Farchi, A., Hartman, A., and Pinter, S.S., Using a Model-Based Test Generator to Test for Standard Conformance, *IBM Systems Journal*, Vol. 41. No. 1, 2002.
- [16] Hartman, Jean, Imoberdorf, Claudio, Meisinger, Michael, UML-Based Integration Testing, *Proceedings, International Symposium on Software Testing and Analysis*, ACM, pp. 60 –70, 2000.
- [17] Holzmann, G.J., From Code to Models, *Proc. 2nd Int. Conf. on Applications of Concurrency to System Design*, pp. 3-10, Newcastle upon Tyne U.K., June 2001.
- [18] Howden, W.E., Kim, R.T., and Tran P.P. Model-Based Extreme Testing, CSE, UCSD, (Extended Abstract) 2002.
- [19] Jeffries, R.E., Extreme Testing, *Software Testing and Quality Engineering*, March/April, 1999.
- [20] Kelliher, T., Blezek, D., Lorensen, W., Miller, J., The Frost Extreme Testing Framework, *Proceedings, Software and Internet Quality Week*, 9T2, SRI, May, 2001.
- [21] Krüger, Ingolf, Grosu, R. Scholz, and Broy, M., From MSCs to StateCharts, *DIPES 98*, Kluwer, 1998.
- [22] Lorensen, William, and Miller, James, Visualization Toolkit Extreme Testing: A Production Release Every Day, *Proceedings, Software and Internet Quality Week*, 4A1, SRI, May, 2000.
- [23] Makinen, E. and Systa, T. An Interactive Approach for Synthesizing UML Statechart Diagrams from Sequence Diagrams, OOPSLA 2000, ACM, 2000.
- [24] Marick, Brian, Agile Methods and Agile Testing, *Software Testing and Quality Engineering*, Sept/Oct, 2001.
- [25] Milner, R. *Communication and Concurrency*, Prentice Hall, 1989.
- [26] Rational Software Corporation, *Users Guide for Rational Robot*, Rational Software Corporation, 2002.
- [27] Rintanen, J., An iterative algorithm for synthesizing invariants, *AAAI/IAAI*, 806-811, July 2000, Austing.

- [28] Sevinc, Suleyman, Theories of Discrete Event Model Abstraction, *Proceedings of the 1991 Winter Simulation Conference*, IEEE, pp. 1115-1119, 1991.
- [29] Silverstein, Michael, SQE Software Test Automation Conference and Expo, Spring, 2002.
- [30] Vasilache S. and Tanaka, Siro, Synthesizing Statecharts from multiple interrelated scenarios, *Proceedings of the International Symposium on Future Software Technology (ISFST2001)*, Zheng Zhou, China, Nov. 5-8, 2001, pp. 158-163.
- [31] Vaziri, M., Holzmann, Automatic Generation of Invariants in SPIN, *Proceedings, 4'th International Spin Workshop*, Paris, 1998, pp. 124-133.
- [32] Whittle Jon, Schumann Johann, Generating Statechart Designs from Scenarios, *Proceedings of the 22nd International Conference on Software Engineering, 2000*, ACM, 2000, pp. 314-323.