High-Level Model Extraction via Symbolic Execution*

Shaohui Wang, Srinivasan Dwarakanathan, Oleg Sokolsky and Insup Lee

Department of Computer and Information Science
University of Pennsylvania
{shaohui,srid}@seas.upenn.edu, {sokolsky,lee}@cis.upenn.edu

Abstract. We study the problem of extracting high-level state machine models from software source code. Our target domain is GUI-driven applications for small hand-held devices such as cell phones and PDAs. In such systems, a natural high-level model is captured by a state machine, where states are GUI screens and button/menu item tappings are actions that trigger transitions between states. The paper presents a symbolic execution technique that allows us to identify states and transitions from the application source code. We discuss an implementation of this technique that operates on a large subset of the C# language and apply as a case study to the subsystem of a decision support tool for medical diagnosis.

Keywords: Model based verification, model extraction, symbolic execution, static program analysis, GUI-driven applications

1 Introduction

We consider the problem of verification of user interface implementations in user-centric reactive systems. Many of such systems are life-critical applications, for examples personal decision support assistants for combat personnel or medical devices. Such systems offer a user interface, through which the user can send signals to the system and observe its responses. The user typically learns to interact with the system by reading the user manual or through targeted training sessions. In either case, the user forms a mental model of the system in his/her mind. A deviation of the observed behavior from the mental model confuses the user and can lead to hazardous situations.

We concentrate on GUI-driven handheld devices as a particular case of user-centric reactive systems. This paper presents the second phase of a case study, in which we analyze a point-of-injury data entry device application called AHLTA-Mobile [1]. It is a point-of-care handheld medical assistant developed by the Telemedicine and Advanced Technology Research Center (TATRC), approved for use by the FDA and deployed in the U.S. Army. AHLTA-Mobile is a C# application on the Microsoft Windows Mobile platform. It consists of a set

* This research has been supported in part by the FDA/TATRC grant MIPR-6MRXMM6093 and NSF grants CNS-0720703 and CNS-0930647.

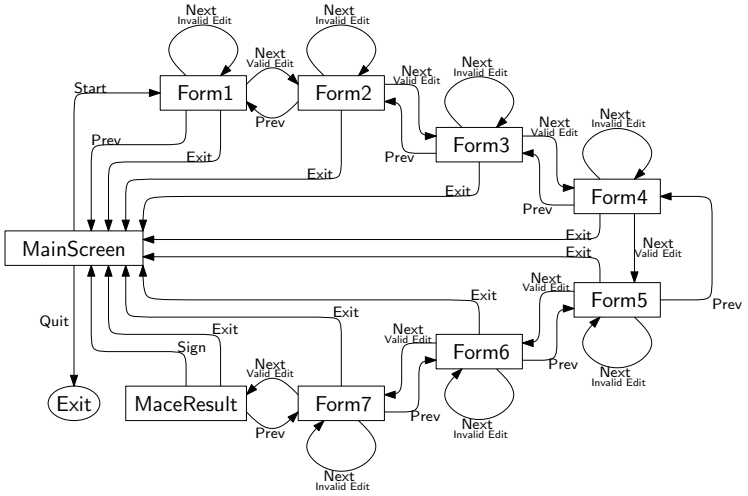

Fig. 1. Expected behavior the MACE exam

of question-and-answer examinations that evaluate common battlefield injuries such as concussions. Medical personnel is rigorously trained in the use of the device and operate in the environment that does not give them time to think about unexpected interactions.

We concentrated on a subset of AHLTA-Mobile's behavior, the Military Acute Concussion Evaluation (MACE) module. MACE is a series of eight GUI screens, displaying forms to be completed by the user. Seven consecutive screens are used to enter results of the user examination, while the last screen, MACE Result, is used to enter diagnosis and offers the possibility to save the results in a database. Entering from the MainScreen to the first, the screens are navigated by invoking the Next Screen button on each screen or the Previous menu item in the Tools menu. In response to users invoking an action, the system moves to a different screen or updates information on the current screen. Note that the user can enter data into the appropriate fields on the screen, but cannot modify user interface actions.

We use finite state machines, extended with local variables, to specify the device behavior. Our modeling approach, suitable for handheld devices with small screens, identifies screens with states and clickable user interface elements (such as buttons and menu items) with actions that effect the change of state. Such an approach may be insufficient for a desktop applications that often display multiple screens simultaneously and dynamically modify screen in response to user actions. In handheld devices, however, screens are typically created statically and only one screen is shown at any time. An extended finite state machine model of the MACE behavior is shown in Figure 1.

In the first phase of the case study, reported in [4], we applied model-based testing using the formalization of user-observable behavior of the device that

had been manually extracted from the detailed but informal specification in the documentation. The objective of the work presented in this paper was to extract a model from the source code of the device software in order to perform a more exhaustive comparison with the specification.

With this modeling approach, we obtain a model that captures the expected behavior at a high level of abstraction, expressed by an extended finite state machine. To facilitate verification, we aim to extract a model from the source code at the same level of abstraction. This is in stark contrast with most of existing model extraction and software model checking tools (e.g. [3, 7, 5, 9, 20]), which extract very low-level models that reflect minute details of the program execution. By skipping over unnecessary details, we can have a simpler and more scalable model extraction algorithm.

The two models, one obtained from the software specification describing how the software should respond to the user interactions, and the other from the extraction following our approach presented in the paper, can be compared and checked for inconsistencies to reveal possible implementation defects[17, 18]. In these approaches, the author assumed the models for user comprehension of the system (mental models) are described in finite state machines, as are the models for the actual system behavior. A mental model and a system model as well as a `consistent` invariant are fed to a model checker to see if violations occur. If so, a so-called *surprise* is found, and a trace is reported by the model checker for cause analysis. Our intent is to use a similar idea for inconsistency checking between software specification and implementation, while the model extraction approach presented in this paper is a first step to obtain such an implementation model.

In order to extract the high-level model, we identify GUI library calls that are used in the creation of screens and UI widgets associated with them. We then apply symbolic execution, starting from the constructor of the main class, and discover how these library calls are made, thus collecting states for the model. For each screen identified in this way, we use symbolic execution of the widget callbacks looking for the library calls that display a screen, each of which is a transition in the model.

The main contribution of this work is the application of symbolic execution to the extraction of a high-level abstraction of the system behavior. A more technical contribution is the notion of *enriched symbolic object*, which makes the application of symbolic evaluation to object-oriented programs easier. Finally, we describe an implementation of the tool that performs model extraction from GUI-driven C# programs.

The paper is organized as follows. Section 2 describes identification of states and transitions within the source code and gives a high-level overview of our model extraction approach. Section 3 describes the application of symbolic execution to model extraction using enriched symbolic objects and introduces the prototype implementation of the analyzer. Section 4 presents results of the case study. We summarize related work in Section 5 and conclude with discussions of future extensions in Section 6 and a summary.

2 The Approach

In our approach, we identify a user screen shown on handheld devices to be a *state* in a finite state machine. On the screen, there are UI widgets which are usually associated with callback functions which make changes to program variables and lead the user from screen to screen. Typically the UI widgets are buttons or menu items which respond to user *tapping* in a specific area of the screen. These tapping events trigger *transitions* in the finite state machine. In situations where only certain values of program variables can enable a button/menu item, these values are reflected as the guard for this transition. The relevant program variables are captured in the finite state machine as *local variables*.

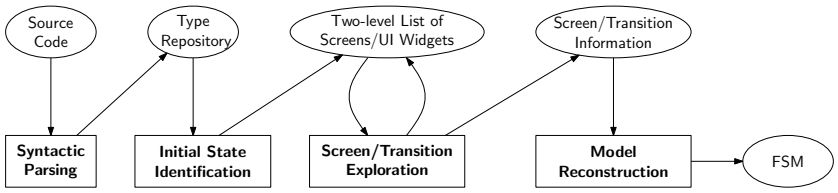

Fig. 2. Overall approach for user-level model extraction

Our approach is depicted in Figure 2. We first parse the source code to generate a collection of abstract syntax trees for the source code, called the *type repository*. It allows us to retrieve method signatures, method bodies, field types, etc., for analysis.

We then apply the symbolic execution technique to the program, described in Subsection 3.1, to gather information for the purpose of screen/transition exploration. Specifically, we collect a list of screen objects that are *potentially* shown to the user. For each collected screen, we identify all the UI elements that have been assigned a callback function.

In the screen/transition exploration phase, we start from the last collected screen as the initial state for the state machine. For each of its buttons or menu items, we perform symbolic execution of the corresponding callback function, looking for the next screen that will appear. If the symbolic execution branches, we use the conjunction of the conditions along the path as a guard for the transition. The name of the widget becomes the label of this transition. In this way, we established one possible transition from the initial screen to a possible next screen, triggered by the widget tapping event.

To identify all such transitions for the reachable screens, we maintain a two-level list, containing the collected screens and entry points to callback functions for their UI widgets to analyze. We then iterate over the two-level list and perform the aforementioned screen exploration technique. When a new screen is shown, we add it (and its callback functions) to the two-level list if it is not processed already, and continue with the exploration. When all screens reachable

from the initial screen and their UI widgets are processed, the exploration phase finishes.

After that, the model reconstruction phase converts the information gathered in the screen/transition exploration phase into the state-machine form by a straightforward traversal.

Assumptions. The approach described above is tailored to the common properties of hand-held applications. The main assumption is that the user is presented with only one screen at a time. Message boxes, which are overlaid on top of the current screen, fit this assumption since the current screen becomes inaccessible until the message box is closed. The second critical assumption is that screen objects, once created, are immutable. UI widgets can be disabled and re-enabled, but they are never added or removed once the screen object is created. Finally, while screen classes can be instantiated any time during the execution of the program, we assume that they do not depend on user input. Thus, conceptually, the set of states is fixed from the beginning.

3 Symbolic Execution for Model Extraction

3.1 Symbolic Execution

Symbolic execution[8, 15] is a static program analysis technique which mimics running the program symbolically. It takes program inputs as symbols and views the execution of the program as computing a function of symbols. When the variables are dereferenced in later computations, their symbolic values are used.

In symbolic execution, the analyzer keeps a tree data structure with *symbolic states* as nodes. A symbolic state consists of a *(symbolic) variable binding* b, a *path condition* p, and a *program counter* c as a triple $\langle b, p, c\rangle$, where

- a symbolic variable binding b is a set of pairs of the form (v, s), where v refers to a variable in the program, and s is the (symbolic) value for v;
- a path condition p is a set of constraints on program variables under which the execution can reach the current symbolic state; and
- a program counter c indicates the next statement to execute in the analysis.

Let $l = \{l_1, \ldots, l_r\}$ be a set of program statements where no conditional or loop statements occur. The symbolic execution on this program l can be formally viewed as a series of transformations starting from an initial symbolic state $\langle b_0, p_0, c_0\rangle$. After symbolically executing the program, a sequence

$$\langle b_0, p_0, c_0\rangle\ l_1\ \langle b_1, p_1, c_1\rangle\ l_2 \cdots l_r\ \langle b_r, p_r, c_r\rangle$$

is produced, from where certain interesting information can be extracted.

Conditional or loop statements potentially branch the analysis into sub-cases if conditional guards or loop bounds cannot be symbolically evaluated. Thus, the process of symbolic execution essentially produces a *symbolic execution tree*, where the tree nodes are symbolic states and labels on edges are the constraints the symbolic analyzer has imposed on program variables, in order to reach from one node to another.

Suppose there is an `evaluate` function which, for each symbolic state and a given expression, is either able to evaluate and return the symbolic value for the

expression at this symbolic state, or, if it cannot, create a brand new symbolic value and return it. The way common program constructs transform symbolic states is informally described below, while a formal characterization can be found in [8].

- For an assignment, the analyzer evaluates the right hand side expression and assigns it to the left hand side variable. In other words, assignments update the variable binding component of a symbolic state.
- Statement blocks are viewed as a list of statements to be analyzed in turn.
- When a method call is encountered, either as a statement or as part of an expression, the analyzer performs *context switch*, which starts a temporary symbolic state with references to variables in the calling context via parameter passing, and symbolically executes the body of the called method. Upon finishing analyzing the method body or encountering a return statement, the analyzer backpropagates changes in the resulting symbolic state to the symbolic state from where the method is called.
- For return statements, only context switching is performed and the changes are backpropagated. An additional symbolic value is available to the calling context if an expression is returned.
- Conditional statements. For simplicity, we consider only `if`...`then`...`else`... statements. When the `evaluate` function can symbolically evaluate the guard of the `if` statement, the analyzer knows which branch to analyze so it proceeds with the corresponding block without branching, while adding the guard (or its negation) to the path condition if the `then` part (or the `else` part) is executed. If it is not possible to do so, the analyzer will branch the symbolic execution tree from the current node. Each child node is initially a duplicate of the current symbolic state. To one branch, the analyzer adds a constraint of the guard being true to its path condition and continues with the `then` block; to the other, the negation of the guard is added to the path condition and the `else` block is executed.
- Loops. Symbolic execution on loops depends on loop boundaries, which may be changed inside the loop. Static analysis cannot always precisely analyze loops due to their dynamic behaviors. One strategy of handling loops is to unroll the loop once and continue the analysis. Specifically, the loop index variable is assigned to the first value in range, and the loop body is processed as a block. After processing this block, the loop is repeated with the boundary changed to exclude the first value.

To use symbolic execution to analyze object-oriented programs in the source code level, we parse the sorce code and build its abstract syntax tree. Thus we collect a repository of class representations containing (a) method representations consisting of method ids and their body statements, and (b) type information for class fields.

The symbolic execution begins with a designated starting class and its `Main` method. The body of the `Main` method is a list of statement nodes which are analyzed sequentially following the aforementioned treatment for different program constructs, according to their respective statement types.

```
using System;
using System.Collections;
class Program {
  static void Main(string[] args) {
    String n1 = "Form1";
    int a = 1; int b = 2;
    ArrayList l = new ArrayList();
    l.Add(a); l.Add(b);
    Screen s = new Screen(n1);
    s.setIds(l);
  }
}
```

```
class Screen {
  String name;
  ArrayList ids;

  public Screen(String n) {
    this.name = n;
  }

  public void setIds(ArrayList al) {
    this.ids = al;
  }
}
```

Fig. 3. Sample program demonstrating enriched symbolic objects/fields.

3.2 Enriched Symbolic Objects

In object-oriented program analysis, it is not enough to simply represent the value of an object using one symbol. If so, data stored inside class/object fields or collection types would become unavailable to the analyzer. A few approaches exist to address this problem[13, 14].

In this paper, we propose the *enriched symbolic objects* data structure, in which the contents of a symbolic object are directly captured. An enriched symbolic object contains a variable's name, type, creation point, as well as a *symbolic field* which can be one of several possible kinds, according to what kind of program variable the symbolic object represents. Specifically, the supported kinds for symbolic fields are summarized and explained briefly below.

(a) SIMPLE: the same as usual symbolic values, used to represent simple types, such as integers, booleans, etc.
(b) COMPOUND: an unordered collection of symbolic objects, such as sets.
(c) ARRAY: an ordered collection of symbolic objects, such as lists and vectors.
(d) OBJECT: a collection of symbolic objects, used to represent member fields of an instance of a class.
(e) CLASS: a collection of symbolic objects, used to represent static member fields of a class. This is unique per class.
(f) SYMBOL: a wrapping of another symbolic object, representing variable alias.

Each enriched symbolic object represents a program variable, and its symbolic field contains its value. For a simple example, the program presented in Figure 3 will create a hierarchy of enriched symbolic objects/fields shown in Figure 4. Note that not all fields in `SymObj`s and `SymField`s are shown in the figure.

Constructors and getters/setters are implemented for creating, reading and updating these symbolic objects. With the enriched symbolic objects, whenever the analyzer needs the symbolic values of a certain program variable, it will check the kind of the symbolic field and use appropriate manipulation functions.

The analyzer is then able to easily inspect the inside of a symbolic object and retrieve its relevant fields. Enriched symbolic objects also simplify the manipulation of static fields and methods of classes. We assign a unique global symbolic object holding the static fields and methods of a class, and direct all accesses to them from instance variables (if any) to this unique object.

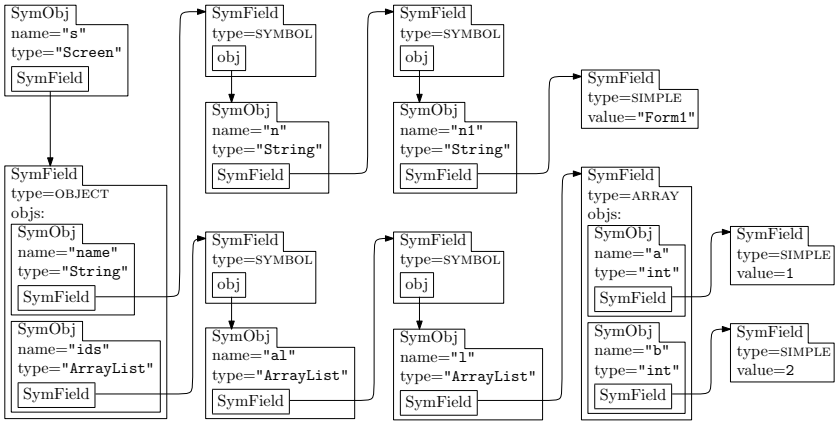

Fig. 4. Enriched symbolic objects/fields representing program in Figure 3.

3.3 Exploring Reachable Screens and Transitions

We describe the algorithm to extract high-level models using the technique of symbolic execution based on enriched symbolic objects in this subsection. The idea behind our algorithm is to augment the standard symbolic execution analyzer so that it becomes aware of screen creation and showing, records possible triggers for screen transitions and performs analysis on the collected triggers.

In GUI programming, screens are created via instantiating a system/framework predefined screen class, or its subclasses, e.g., `System.Windows.Form` in C#. We equip the analyzer with necessary knowledge on which class types and their derived classes define screens. To identify the set of *all potential screens* that might appear on user device during the process of symbolic execution, we record whenever a variable of screen types is created. Recall that the type information can be retrieved from the type repository during the parsing phase. These variables hold references to potential screens.

There are two more tasks to finish: (a) to identify which screen is initial; and (b) to identify what screens are *actually reachable* via which transitions from the initial screen. They are achieved in two phases, namely Initial State Identification and Screen/Transition Exploration in Figure 2, respectively.

In the Initial State Identification phase, the analyzer starts with the `Main` method of a designated class, performs the analysis, and builds the symbolic execution tree as described in Subsection 3.1.

When the symbolic analyzer reaches a point where initialization for the program has finished and a screen is shown to the user waiting for interactions,[1] the last screen shown is the initial screen, as well as the initial state of the finite state machine model we extract. We start the exploration phase from the symbolic

[1] In our case study, the particular statement is `Application.Run()`. In essence, the code for keeping a program running in idle state waiting for user inputs would vary, but the idea here is the same.

state at the detection of the initial screen, which we call the *exploration starting state*.

At this point, the range of user inputs may be overwhelming. To extract a finite state machine model from the program, we use a special exploration technique.

We maintain a conceptual two-level list L for use in the exploration. The first level contains all the symbolic objects of screen types the analyzer has encountered. Corresponding to each screen variable s in the first level, there is a list, denoted $U(s)$, containing all the UI widgets which have been assigned a callback function. We use $f(u)$ to denote the callback function for widget u. Initially, the two-level list contains only the initial screen s_0 and $U(s_0)$.

The exploration starts with a widget u from $U(s)$ for a selected screen s by performing symbolic execution of the callback function $f(u)$ in the *exploration starting state*. Whenever another screen s' is shown to the user during the symbolic execution, we add s' and $U(s')$ to the two-level list L, if $s' \notin L$. This discovers one transition $s \xrightarrow{u} s'$ from screen s to screen s' via the selected UI widget u. The exploration collects information for model reconstruction, marks this transition as processed, and continues with the next UI widget in $U(s)$, with a fresh copy of *exploration starting state*. When all elements in $U(s)$ is processed, we mark s as processed and continue with the next screen in L.

If during the exploration the analyzer branches due to a conditional statement, the condition is added to the path condition of the symbolic states. We obtain the guard for the transition by computing the set difference of the path condition for the symbolic state where s' is discovered and that of the *exploration starting state*. Algorithm 1 details the process.

Scenarios where the execution of a callback function does not show a new screen are treated as it creates a self-loop in the state machine. An important special case occurs if the callback hides the current screen but does not show another one. This situation is treated as an error in the application that causes the target state to become unspecified.

With the collected state/transition tuples, the finite state machine model is easily reconstructed. We identify any instance of the same screen class to be the same state in the finite state machine, and then we add transitions according to their source/target states, and attach guards if any.

3.4 Analyzer Implementation

We implemented the algorithms described above in a Java-based prototype tool. We use a version of C# parser generated by JavaCC [12]. The parser implements a sufficiently large subset of C# language features. The parser constructs the type repository, storing class representations and method bodies in the form of abstract syntax trees.

The analyzer takes the name of the main application class as input. The first phase of the analysis runs symbolic execution of the constructor of the main class and identifies the initial screen. This phase completes when each

Algorithm 1: Screen/Transition Exploration

input : two-level list `L` with initial screen, exploration starting state `state`
output: list `trans` of gathered states transitions for FSM reconstruction
initialization: `trans` $\leftarrow \emptyset$;
while `L` *contains unprocessed screen* **do**
  `screen` $\leftarrow$ next screen in `L`; `widgets` $\leftarrow U$(`screen`);
  while `widgets` *contains unprocessed widget* **do**
    `widget` $\leftarrow$ next one in `widgets`;
    `state_0` $\leftarrow$ `state`;
    start symbolic execution from `state_0` with the callback function f(`widget`) until (a) `target` screen is shown, or (b) the callback function is finished, or (c) a statement for exiting the program;
    if *Case (b)* **then** `target` $\leftarrow$ `screen`;
    if *Case (c)* **then** `target` $\leftarrow$ `Exit`;
    `guard` $\leftarrow$ $\text{PathCondition}_{\texttt{CurrSymbolicState}} \setminus \text{PathCondition}_{\texttt{state}}$;
    add the tuple $\langle$`screen`, `widget`, `guard`, `target`$\rangle$ as a piece of model reconstruction information to `trans`;
    mark `widget` as processed;
    if *Case (a)* $\wedge$ `target` $\notin$ `L` **then** add `target` and U(`target`) to `L`
  end
  mark `screen` as processed
end
return `trans`

branch of symbolic execution reaches the `Application.Run()` statement, which transfers control to the C# message loop and displays the initial screen. The second phase of screen/transition exploration collects the tuples used in the model reconstruction phase as defined above.

The current implementation produces the extracted state machine in textual format. Graphical representations as well as inputs to other finite state machine analysis tools can be straightforwardly generated.

Currently the analyzer supports essential features in object-oriented program analysis. Class definitions and inheritance, method representations, field information, built-in primitive types and basic operations on them are supported. The analyzer processes most statement types including assignments, method calls/returns, statement blocks, conditional statements, and loops (`for` and `foreach`). Issues in dealing with a few remaining program constructs are found in Section 6. A complete list of supported and unsupported C# language features is attached in Appendix A.

4 Experimental Results

For purposes of testing our model exploration algorithm and implementation of the symbolic execution analyzer based on the data structure of enriched symbolic objects, we applied the analyzer to the MACE subsystem of the AHLTA-Mobile project.

```
----Identified Screens----
MainScreen(MainScreen.MainScreen at ms =
    MainScreen.MainScreen(this);)
Form1(Form1.Form1 at s[0] = Form1.Form1(this);)
Form2(Form2.Form2 at s[1] = Form2.Form2(this);)
Form3(Form3.Form3 at s[2] = Form3.Form3(this);)
Form4(Form4.Form4 at s[3] = Form4.Form4(this);)
Form5(Form5.Form5 at s[4] = Form5.Form5(this);)
Form6(Form6.Form6 at s[5] = Form6.Form6(this);)
Form7(Form7.Form7 at s[6] = Form7.Form7(this);)
MaceResult(MaceResult.MaceResult at s[7]
    = MaceResult.MaceResult(this);)
Initial state: [MainScreen(ms.Show at ms.Show();)]
----Identified Transitions----
MainScreen:quitButton::Exit
MainScreen:examButton::Form1
Form1:prevMenuItem::MainScreen
Form1:nextMenuItem:NOT(textBox1.Text=="");:Form2
Form1:nextMenuItem:textBox1.Text=="";:Form1
Form1:exitMenuItem::MainScreen
Form2:prevMenuItem::Form1
Form2:nextMenuItem:NOT(textBox2.Text=="");:Form3
Form2:nextMenuItem:textBox2.Text=="";:Form2
Form2:exitMenuItem::MainScreen
Form3:prevMenuItem::Form2
Form3:nextMenuItem:NOT(textBox3.Text=="");:Form4
Form3:nextMenuItem:textBox3.Text=="";:Form3
Form3:exitMenuItem::MainScreen
Form4:prevMenuItem::Form3
Form4:nextMenuItem:NOT(textBox4.Text=="");:Form5
Form4:nextMenuItem:textBox4.Text=="";:Form4
Form4:exitMenuItem::MainScreen
Form5:prevMenuItem::Form4
Form5:nextMenuItem:NOT(textBox5.Text=="");:Form6
Form5:nextMenuItem:textBox5.Text=="";:Form5
Form5:exitMenuItem::MainScreen
Form6:prevMenuItem::Form5
Form6:nextMenuItem:NOT(textBox6.Text=="");:Form7
Form6:nextMenuItem:textBox6.Text=="";:Form6
Form6:exitMenuItem::MainScreen
Form7:prevMenuItem::Form6
Form7:nextMenuItem:NOT(textBox7.Text=="");
    :MaceResult
Form7:nextMenuItem:textBox7.Text=="";:Form7
Form7:exitMenuItem::MainScreen
MaceResult:prevMenuItem::Form7
MaceResult:signMenuItem::MainScreen
MaceResult:exitMenuItem::MainScreen
```

Fig. 5. Analyzer output for MACE subsystem of AHLTA-Mobile

For the case study, we modified the source code of the application in the following ways. Several major features, such as storing information in the database, transmission over the network, and creating installer cabinet files, should be transparent to the user, according to the system documentation. These features were removed by manual "slicing", since we are not aware of program slicing tools that work at the C# source code level. We reduced the large number of input fields on the screens and simplified data validation to just check for the presence of inputs. Links on the main screen to other exam modules that were not included in the case study were also removed.

We made several other modifications to replace C# constructs that are not supported by the analyzer with equivalent code. For example, constants defined in enumerations were replaced with their values directly. Partial class definitions, commonly seen in C# GUI programming, were manually combined.

The output of the analyzer invoked on the source code of the resulting application is shown in Figure 5. It is easy to see that the resulting finite states machine model is isomorphic to the one shown in Figure 1. The only difference is in names of states and transition guards.

5 Related Work

State-machine modeling of GUI-based programs. This modeling approach has been primarily applied in the context of testing. In [2, 23], the authors explored observable effects of GUI on the program (e.g., changes on program data reflected by some UI components) by identifying *complete interaction sequences* (UI widgets involved and sequence of actions invoked by the user). These sequences and changes of program status were represented using FSM models. In

our previous work [4], we used the NModel tool [11] to test compliance of a GUI-based system with a behavioral specification, constructed with the same modeling approach used in this paper. However, none of these papers addressed the question of how these models are obtained.

Model Extraction. There is much work on the extraction of formal models from software, primarily in the context of software model checking [3, 5, 7, 9, 20]. The main difference is that models extracted in these approaches are very low-level: the state is an assignment of values to program variables at each step of the execution. Our approach serves a different purpose and obtains a much higher-level model.

The work on model extraction for GUI programs closest to ours is [9]. The authors also used symbolic execution techniques, but for the purpose of test case generation with regards to on-screen widgets for data collection. Their approach relies on concrete execution of the code using instrumentation. Symbolic execution of event handlers is mentioned but not discussed in detail.

The work presented in [20] bears a broader goal than ours in that they try to reverse-engineer Java Swing programs to obtain a program model. The state is defined as a collection of program variables that record user inputs from text fields and radio buttons for choices. Each user event may lead from one state to another. Their techniques may be useful in our context for more data-intensive applications. However their work is ongoing and details of model extraction techniques are not revealed.

Symbolic Execution. Symbolic execution [8, 15] has been studied extensively and has its applications in a large variety of program analysis [6, 13, 14, 16]. For example, in [6], symbolic execution has been employed to analyze partial correctness and general safety properties of concurrent Ada programs. In [13], the authors investigated invariants for particular data structures using *universal symbolic execution*, i.e., not only for inputs but also for every reading of an *lvalue* that is not bound, the analyzer will assign a new symbolic value to it. [16] also surveys several other research trends and applications of symbolic execution.

In [14], the ideas of *lazy initialization* and *generalized symbolic execution* are proposed. The fields of an object are only lazily initialized when dereferenced. The generalized symbolic execution differs from universal symbolic execution in [13] slightly in that a `null` element can be assigned to a variable during its symbolic evaluation so that potential program errors caused by reference to `null` pointers can be systematically analyzed.

Subclassing are also addressed in [14] by remembering certain type information from the program under analysis, while type casting is not mentioned in their approach either. In our approach, we used enriched symbolic objects for the purpose of handling collection objects. This is a simple yet effective approach in our setting. Usually the number of screen objects in GUI applications on handheld devices is limited. Our approach explicitly and directly handles these objects, while the approach in [14] essentially asks an external constraint solver for the symbolic value to assign to the object.

Several existing tools such as Java PathFinder[22], jCUTE[19], or Pex[21] are based on, or utilize, symbolic techniques. Aside from the language differences (Java vs. C#, bytecode vs. source code), a common missing feature from these tools is the ability of state/transition exploration, proposed in this paper. Weaving our state/transition exploration algorithm into these tools may be a choice, but our focus at the current step is to verify the effectiveness of our algorithm, rather than a general purpose extension to these tools.

6 Discussions

Our tool is a useful first step toward extraction of user-level models from source code. To make the tool more useful in practice, we can extend it in several ways.
Completion of C# language features. Several features of C# are not supported by our parser/analyzer. Unsupported features include namespaces, partial classes, accessibility controls, interfaces, data types other than `int`/`uint`/`bool`/`string`, some expression/statement types (`switch`/`case` conditionals, `do`/`while` loops, `break`/`continue`/`goto` statements, etc.). Implementing these features would be important to make the tool applicable to real code; yet they do not bring any new aspects to model extraction.
Exception handling. Exceptions that are raised during the interaction with the user are, conceptually, easy to handle. Most such exceptions – such as invalid inputs or a broken network connection – are caught by the application and presented to the user in warning messages. However, exceptions that occur during initialization present a challenge. Such exceptions are typically catastrophic; that is, execution is aborted before interaction with the user begins and the state machine is, effectively, never constructed. In our current implementation, uncaught exceptions result in aborting the model extraction. This can be improved in the future by introducing special error states into the model. Exceptions that are caught are processed as alternative branches by the analyzer. If different screens are shown as a result of an exception, the state machine will contain transitions labeled by exceptions.
Slicing. Many details of the application code are transparent to the user and are not reflected in the state machine model. For example, the device in the case study stores entered data and, once the exam is complete, transfers results in a database. Performing a slicing step prior to symbolic execution will reduce the amount of code to be processed and the size of the context, making the tool more efficient.
Abstracting from UI libraries. Currently, our tool targets programs built using the .NET `Forms` UI framework. However, our model extraction approach is not specific to `Forms`. It is tempting to encapsulate the model extraction in such a way that it can work with different UI frameworks, such as the Windows Presentation Framework. Our model extraction needs to be able to identify several operations that any UI framework provides: creating screens, adding UI widgets to a screen, adding callbacks to a widget, etc. Each of these operations corresponds to library calls, which vary in different UI frameworks. It should be possible to provide a

description of the relevant calls and their parameters for each UI framework, instead of hard-coding this information into the analyzer. Starting the analyzer, the user will specify the UI framework to use. The analyzer then interprets the corresponding description.

By the same token, our approach is not specific to C#. We may be able to abstract the source language, as long as the parser can produce the appropriate type repository. However, making the type repository format independent of the source language is a research problem in its own right, since the symbolic execution engine needs to have precise understanding of its semantics.

Overcoming tool limitations. Because we rely on static analysis techniques for model extraction, we by necessity over-approximate the set of behaviors captured by the state machine model. Techniques that improve accuracy of static analysis (for example, predicting loop bounds) would improve precision of model extraction. One can also consider combining symbolic execution with concrete execution of code portions in the spirit of [19].

7 Conclusion

We presented an approach for the extraction of high-level models of GUI-driven software. Our modeling approach is targeting applications on small hand-held devices, which present the user with one screen at a time. In applications that run on such devices, screens tend to be defined statically in the code and are not modified during the user interaction. We are modeling screens as states in the model, and invocations of UI widgets as transitions between states.

The model extraction technique is based on the symbolic execution of the application source code, which allows us to statically track references to the screens as symbolic objects and use them to reconstruct the state machine. To be able to do this, we present the enriched symbolic objects data structure that keeps track of the symbolic values of class instances.

We have implemented our model extraction technique in a tool and used it to obtain the model of a decision support device for medical personnel.

Besides extending the capabiblities of our symbolic analyzer tool for model extraction with more language features, we are also studying the problem of detecting significant inconsistencies between software specification and implementation in the medical diagnosis software domain, with model-based techniques and our extracted models as a basis.

A Feature List

In this appendix, we list the assumptions and language features of the C# language that our analyzer currently does and does not support. The categorization for the feature list is taken from the C# language definition [10]. The "Parser" and "Analyzer" columns indicate whether, and if yes, to what extent, they are supported in the parser level or the analyzer level, separately.

Table 1: BMIST Analyzer Feature List

C# Feature	**Sub Category**	**Sub Category**	**Parser**	**Analyzer**	**Notes**
Types	Value Types	Simple Types	Yes	Partial	Only `int`, `uint`, and `bool` are now supported.
		Enum Types	No	Partial	Currently manually replaced with `uint` values in the source code.
		Struct Types	Yes	Partial	Treated the same as class types.
		Nullable Types	Yes	Yes	Null pointers are detected in the parser.
	Ref Types	Class Types	Yes	Yes	Support fields and functions, static modifiers, static initializer, sub-classing.
		Interface Types	No	Yes	Whenever a member function definition is required, we find it in the corresponding class definition. In other words, we assume all interface fields and functions are defined in the classes implementing the interfaces.
		Array Types	Partial	Partial	Only one-dimensional arrays are supported. Array indexing uses special treatment. Not quite extensible for more complex array manipulation.
		Delegate Types	No	Partial	Only very specific types of delegate functions are supported (namely, the callback functions for buttons and/or menuitems).
Expressions	Primary	Member acces	Yes	Yes	Resolving x and look inside its fields.
		Method call	Yes	Yes	
		Array indexing	Partial	Partial	Only one-dimensional.
		`x++`, `x--`	No	No	
		`new T(...)`	Yes	Partial	Treated as if the constructor function is called. Overloading of constructor functions is not supported.
		`new T(...) {...}`	No	No	Object creation with the initializer syntax.
		`new T[...]`	Yes	Partial	Array creation. Only one-dimensional.
		`typeof(T)`	Yes	Yes	Parsed as a function call, but treated specially.

Continued on next page...

Table1: BMIST Analyzer Feature List (Continued)

C# Feature	Sub Category	Sub Category	Parser	Analyzer	Notes
		`checked(x)` `unchecked(x)`	Yes	No	Parsed as a function call, but will result in error (definition of function not found).
		`default(T)`	Yes	No	Parsed as a function call, but will result in error (definition of function not found).
		`delegate {...}`	No	No	Anonymous function/method.
	Unary	`+x`, `-x`	Yes	Yes	Identity and negation.
		`!x`	Yes	Yes	Logical negation.
		`~x`	No	No	Bitwise negation.
		`++x`, `--x`	No	No	
	Multiplicative	`x*y`	Yes	Yes	
		`x/y`	Yes	Partial	Results are only integers.
		`x%y`	Yes	Yes	
	Additive	`x+y`	Yes	Yes	
		`x-y`	Yes	Yes	
	Shift	`x<<y`	No	Partial	If constant, replaced with actual values in source code.
		`x>>y`	No	Partial	If constant, replaced with actual values in source code.
	Relational	`<`, `>`, `<=`, `>=`	Yes	Yes	Integers only.
	Equality	`==`, `!=`	Yes	Yes	Integers, bools, reference types, etc.
	Type testing	`x is T`	No	No	
		`x as T`	No	No	
	Type casting	`(T)x`	Yes	Partial	Always assume casting is successful, returning a new variable representing an object of the designated type.
	Logical	`x&y`, `x^y`, `x|y`	No	No	Integer bitwise operations, or logical operations (AND, XOR, and OR).
		`x&&y`, `x||y`	Yes	Yes	
		`x??y`	No	No	If `x` is null then `y`, otherwise `x`.

Continued on next page...

Table1: BMIST Analyzer Feature List (Continued)

C# Feature	**Sub Category**	**Sub Category**	**Parser**	**Analyzer**	**Notes**
		`x ? y : z`	Yes	Yes	Ternary if-then-else expression.
	Assignments	`x=y`	Yes	Yes	
		`x` *op*`= y`	Yes	Partial	Works with `+=`.
		`(T x) => y`	No	No	Lambda expressions.
Statements	Declarations	Local variables	Yes	Yes	E.g., `static void Main() { int a; int b = 2; ...}`
		Local Constants	Yes	Yes	E.g., `static void Main() { const float pi = 3.14; ...}`

	Expression Statements		Yes	Partial	Only when the expression is supported. E.g.,
					`static void Main() {`
					`int i;`
					`i = 123; // Expression statement supported`
					`i++; // Expression statement not supported`
					`}`

	`if` statement		Partial	Partial	Only one pair of `if (...) {...} else {...}` is supported. For other formats, needs manual rewriting in source code.
	`switch` statement		No	Partial	Manual rewriting.
	`while` loop		Yes	Partial	Only the syntax used in the Windows Message Loop is supported.
	`do` loop		No	No	
	`for` loop		Yes	Partial	Assuming bounds to be a fixed number. No jumping statements are allowed except for `return`.
	`foreach` loop		Yes	Partial	No jumping statements are allowed except for `return`.
	`break` statement		No	No	
	`continue` statement		No	No	

Continued on next page...

Table1: BMIST Analyzer Feature List (Continued)

C# Feature	**Sub Category**	**Sub Category**	**Parser**	**Analyzer**	**Notes**
	`goto` statement		No	No	
	`return` statement		Yes	Yes	
	`yield` statement		No	No	
	`try / catch / throw`		Yes	No	Always works only with code in try block.
	`checked / unchecked` statements		No	No	
	`lock` statement		No	No	
	`using` directive		No	No	The application class hierarchy is flattened with full names and the `using` directives are ignored.
Classes	Members	Constants	Yes	Yes	A special "class object" is created to contain class constants and also static fields.
		Fields	Yes	Partial	See below for "Fields."
		Methods	Yes	Partial	Overloading is not supported.
		Properties	No	Partial	Converted to methods in source code.
		Indexers	No	No	
		Events	No	Partial	Restricted to Buttons and MenuItems.
		Operators	No	No	
		Constructors	Yes	Partial	Overloading is not supported.
		Destructors	Yes	No	

Continued on next page...

Table1: BMIST Analyzer Feature List (Continued)

C# Feature	Sub Category	Sub Category	Parser	Analyzer	Notes
		Nested Types	No	No	Nested type declarations inside classes are not supported.
	Accessibility	`public`	Yes	Yes	All fields/methods are assumed to be public to the analyzer.
		`protected`	Yes	No	Access limited to this class or classes derived from this class.
		`internal`	Yes	No	Access limited to this program.
		`protected internal`	Yes	No	Access limited to this program or classes derived from this class.
		`private`	Yes	No	Access limited to this class.
	Type Parameters		No	No	Generic types are not supported.
	Type Hierarchy		Yes	Partial	As long as no overloading occurs.
	Fields	`static`	Yes	Yes	A special "class object" is created to contain class constants and also static fields.
		`readonly`	Yes	No	
	Methods	`static`	Yes	Yes	A special "class object is created to contain class constants and also static fields.
		instance	Yes	Yes	
		parameters	Yes	Partial	The `ref`, `in`, and `out` modifiers are not supported. Parameter arrays are not supported.
		`virtual`	Yes	No	
		`abstract`	Yes	No	
		`override`	Yes	No	
		Overloading	No	No	

Continued on next page...

Table1: BMIST Analyzer Feature List (Continued)

C# Feature	**Sub Category**	**Sub Category**	**Parser**	**Analyzer**	**Notes**
Structs			Yes	Yes	According to [10], quoted below, they are treated as classes. "Like classes, structs are data structures that can contain data members and function members, but unlike classes, structs are value types and do not require heap allocation. A variable of a struct type directly stores the data of the struct, whereas a variable of a class type stores a reference to a dynamically allocated object. Struct types do not support user-specified inheritance, and all struct types implicitly inherit from type object."
Arrays			Partial	Partial	See "Types → Ref Types → Array Types" part.
Interfaces			No	Partial	It is assumed that a class implementing an interface will define the interface methods, so analyzing for interfaces solely relying on finding method definitions in classes.
Enums			No	Partial	See "Types → Value Types → Enum Types" part.
Delegates			No	Partial	See "Types → Ref Types → Delegate Types" part.
Attributes			No	No	Ignored by the analyzer. The effects of some attributes appeared in the source code are taken into account according to their functions (e.g., dll loading directives).

References

1. AHLTA-Mobile fact sheet. Medical Communications for Combat Casualty Care Web Site, https://www.mc4.army.mil/AHLTA-Mobile.asp
2. Belli, F.: Finite-state testing and analysis of graphical user interfaces. 12th International Symposium on Software Reliability Engineering (ISSRE) (2001)
3. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker BLAST: Applications to software engineering. International Journal on Software Tools for Technology Transfer 9(5–6), 505–525 (2007)
4. Chinnapongse, V., Sokolsky, O., Lee, I., Wang, S., Jones, P.L.: Model-based testing of GUI-driven applications. In: The 7th IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (2009)
5. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach, S., Păsăreanu, C.S., Zheng, H.: Bandera: extracting finite-state models from java source code. In: Proceedings of the 22nd international conference on Software engineering. pp. 439–448 (2000)
6. Dillon, L., Kemmerer, R., Harrison, L.: An experience with two symbolic execution-based approaches to formal verification of Ada tasking programs. In: The 2nd Workshop on Software Testing, Verification, and Analysis (1988)
7. Dwyer, M.B., Hatcliff, J., Hoosier, M.: Building your own software model checker using the bogor extensible model checking framework. In: Proceedings of 17th Conference on Computer-Aided Verification (CAV) (2005)
8. Fahringer, T., Scholz, B.: Advanced symbolic analysis for compilers: new techniques and algorithms for symbolic program analysis and optimization. Springer-Verlag, Berlin, Heidelberg (2003)
9. Ganov, S.R., Killmar, C., Khurshid, S., Perry, D.E.: Test generation for graphical user interfaces based on symbolic execution. In: The 3rd International Workshop on Automation of Software Test (AST) (2008)
10. Hejlsberg, A., Wiltamuth, S., Golde, P.: The C# Programming Language (Microsoft .Net Development Series). Addison-Wesley Professional, 2nd edn. (Oct 2003)
11. Jacky, J., Veanes, M., Campbell, C., Schulte, W.: Model-based Software Testing and Analysis with C#. Cambridge University Press (2008)
12. JavaCC: The Java Compiler Compiler, https://javacc.dev.java.net/
13. Kannan, Y., Sen, K.: Universal symbolic execution and its application to likely data structure invariant generation. In: Proceedings of the 2008 International Symposium on Software Testing and Analysis. ACM (2008)
14. Khurshid, S., Păsăreanu, C., Visser, W.: Generalized symbolic execution for model checking and testing. In: Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 2619 (2003)
15. King, J.C.: Symbolic execution and program testing. Commun. ACM 19(7), 385–394 (1976)
16. Păsăreanu, C., Visser, W.: A survey of new trends in symbolic execution for software testing and analysis. International Journal on Software Tools for Technology Transfer (STTT) (2009), http://dx.doi.org/10.1007/s10009-009-0118-1
17. Rushby, J.: Analyzing cockpit interfaces using formal methods. In: Bowman, H. (ed.) Proceedings of FM-Elsewhere. ENTCS, vol. 43. Elsevier (2000)
18. Rushby, J.: Using model checking to help discover mode confusions and other automation surprises. Reliability Engineering and System Safety 75(2), 167–177 (Feb 2002)

19. Sen, K.: Concolic testing. In: Proceedings of $22^n d$ International Conference on Automated Software Engineering (ASE). pp. 571–572 (2007)
20. Silva, J., Campos, J., Saraiva, J.: Models for the reverse engineering of java/swing applications. In: 3rd International Workshop on Metamodels, Schemas, Grammars, and Ontologies (ateM 2006) for Reverse Engineering (2006)
21. Tillmann, N., De Halleux, J.: Pex: white box test generation for .Net. In: Proceedings of the 2nd International Conference on Tests and Proofs (TAP). pp. 134–153. Springer-Verlag (2008)
22. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Automated Software Engineering 10(2) (Apr 2003)
23. White, L., Almezen, H.: Generating test cases for GUI responsibilities using complete interaction sequences. In: Proceedings of the 11th International Symposium on Software Reliability Engineering (ISSRE) (2000)