University of Pennsylvania
ScholarlyCommons

Departmental Papers (CIS)

Department of Computer & Information Science

11-16-2009

Model-Based Testing of GUI-Driven Applications

Vivien Chinnapongse
University of Pennsylvania, vichi@cis.upenn.edu

Insup Lee
University of Pennsylvania, lee@cis.upenn.edu

Oleg Sokolsky
University of Pennsylvania, sokolsky@cis.upenn.edu

Shaohui Wang
University of Pennsylvania, shaohui@cis.upenn.edu

Paul L. Jones
U.S. Food and Drug Administration, PaulL.Jones@fda.hhs.gov

Follow this and additional works at: http://repository.upenn.edu/cis_papers

Part of the Graphics and Human Computer Interfaces Commons, and the Software Engineering Commons

Model-Based Testing of GUI-Driven Applications

Abstract

While thorough testing of reactive systems is essential to ensure device safety, few testing methods center on GUI-driven applications. In this paper we present one approach for the model-based testing of such systems. Using the AHLTA-Mobile case study to demonstrate our approach, we first introduce a high-level method of modeling the expected behavior of GUI-driven applications. We show how to use the NModel tool to generate test cases from this model and present a way to execute these tests within the application, highlighting the challenges of using an API-geared tool in a GUI-based setting. Finally we present the results of our case study.

Disciplines

Graphics and Human Computer Interfaces | Software Engineering

Comments

The Seventh IFIP WG 10.2 International Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2009), Newport Beach, CA, USA, November 16-18, 2009.

Model-Based Testing of GUI-Driven Applications[*]

Vivien Chinnapongse[1]   Insup Lee[1]   Oleg Sokolsky[1]   Shaohui Wang[1]
Paul L. Jones[2]

[1] University of Pennsylvania
[2] U.S. Food and Drug Administration
{vichi,lee,sokolsky,shaohui}@cis.upenn.edu   PaulL.Jones@fda.hhs.gov

Abstract. While thorough testing of reactive systems is essential to ensure device safety, few testing methods center on GUI-driven applications. In this paper we present one approach for the model-based testing of such systems. Using the AHLTA-Mobile case study to demonstrate our approach, we first introduce a high-level method of modeling the expected behavior of GUI-driven applications. We show how to use the NModel tool to generate test cases from this model and present a way to execute these tests within the application, highlighting the challenges of using an API-geared tool in a GUI-based setting. Finally we present the results of our case study.

1   Introduction

Thorough testing of reactive systems is an active research area with a long history. Reactive systems are primarily event-driven systems that operate by continuously interacting with their environment, responding to received signals. Operation of reactive systems is often safety- and life-critical. Rigorous development and analysis techniques are required to ensure safe and correct operation of such systems. In many safety-critical domains, for example in avionics and medical device areas, government regulators certify or approve systems before they can be used. In particular, the U.S. Food and Drug Administration (FDA) approves medical devices for use in the United States.

An important class of reactive systems comprises systems interacting with a human user. 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 head. This model is then used as a specification, against which operation of the system is assessed. In this paper, we are interested in establishing conformance between the system operation and user expectations. Conformance between the

---

mental model and observable behavior of the system is important from different perspectives. From the development perspective, it will help avoid usability problems in the system. From the regulatory perspective, it may help to evaluate necessary user training and instruction materials that accompany the device.

We concentrate on GUI-driven handheld devices as a particular case of user-centric reactive systems. In the long-standing collaboration between experts at the FDA and the high-confidence systems design group at Penn (e.g., [1,3]), we have considered several medical devices that fall in this category. This paper has been motivated by a recent case study, in which we analyze a point-of-injury data entry device application called AHLTA-Mobile [2]. The Armed Forces Health Longitudinal Tracking Application–Mobile (AHLTA-Mobile) 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^{TM} platform. It assists medical personnel, deployed, on military bases, or at military medical centers, with diagnosis and treatment of patients. Medical personnel also use the solution to record patient clinical encounters and transmit those records to a central data repository. AHLTA-Mobile provides users access to service members' complete medical records and offers advice for diagnosis and treatment. It contains a set of question-and-answer examinations that evaluate common battlefield injuries such as concussions. For the safety of patients it is important that the device always functions correctly, because misdiagnosis and incorrect treatment can cause serious harm.

For the purposes of this paper we are concerned with the correctness of 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 of these screens, to which we refer as MACE 1 [3] through MACE 7, are used to enter results of the user examination, while the last screen, MACE Results, is used to enter diagnosis and offers the possibility to save the results by entering them in a database. Relevant screens including Start Screen, Resume Screen, No Unit, and Error are also sconsidered. 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. This observation led us to represent the mental model of the device as a state machine, in which states are identified with GUI screens and transitions represent changing screens in response to invoking UI elements. Each transition in such a state machine is labeled with the UI element that effects the change. In our case study, we constructed the model manually through the careful reading of the AHLTA-Mobile user manual [16]. We discuss the model in more detail in Section 3.2. Of the 114,000 lines of C# code that

---

[3] Throughout this paper we use a sans font for the names of GUI items. We use a `fixed-width` font to identify model and source code elements.

comprise the AHLTA-Mobile application, MACE screen classes and auxiliary classes contain approximately 6,000 lines of code.

Given the state-machine model of the system, we can pursue two approaches to ascertain compliance of the system to its model. One is model-based testing, where the model is used to generate a test suite, which is then applied to the system implementation. Several tools are available for model-based testing of software. In our case study, we used NModel [7] from Microsoft Research, one of the few tools that target C# applications. The other alternative is to extract a state-machine model from the application source code and compare it directly to the mental model using a suitable notion of state machine equivalence or preorder. Although it makes more thorough testing possible, this alternative is much more challenging, and is one subject of our ongoing work.

The contributions of this paper are threefold. First, we present an approach to capture behavioral models of GUI-driven handheld devices. We believe that the high-level modeling approach we have applied to represent the mental model of the AHLTA-Mobile device will be equally applicable to most devices in this category. Second, we present lessons learned during model-based testing to the AHLTA-Mobile case study. We discuss challenges we faced while applying the NModel methodology in the GUI-based setting, and the ways in which we overcame these challenges. Finally, we present results of the case study, which uncovered inconsistencies between the device behavior and the desired behavior described in the manual.

The paper is organized as follows: Section 2 describes the NModel framework to be used in analyzing AHLTA-Mobile. Section 3 discusses the development of a mental model, both as an extended finite state machine (EFSM) and as an NModel model program. Section 4 explains the creation of a test harness to link an implementation with test cases. The testing of the AHLTA-Mobile application is described in Section 5. Section 6 discusses related research work. We conclude our paper with a discussion of our contributions in Section 7.

2 Using NModel to Analyze MACE

Developed at Microsoft Research, the NModel [6,7] framework is a model-based software testing and analysis tool for C# programs. NModel allows us to create a formal model of an implementation's expected behavior and determine through model-based testing whether or not the implementation's actual behavior and the model are consistent. The open-source tool is freely available online at no cost and there is a good level of support and documentation. No other tool we discovered matched this description and we decided NModel would suit our purposes reasonably well.

The NModel framework consists of the following components:

- a library for creating *model programs*, executable specifications for implementations,
- a *model program viewer* (mpv) for viewing model programs as finite-state machines (FSMs),

– an *offline test generator* (`otg`), which performs link coverage of model programs to produce test cases, and
– a *conformance tester* (`ct`), which takes test cases and executes them within the implementation. [4] This must be coupled to the implementation with a test harness, called a *stepper*.

A diagram of the steps involved in testing implementations with NModel is provided in Figure 1:

1. First, we take the specifications and/or the user manual and write a model program using the NModel library. We can use this model program to generate a graphical FSM using `mpv` for a visual representation.
2. Then, we use `otg` to generate a test suite from the model program.
3. To test the implementation with the test suite, we first write a stepper to couple the test cases described in the model program with the implementation.
4. Finally, we run `ct` with the test suite and the implementation coupled with the stepper to check for consistency between the implementation and the model. The output of `ct` is `Success` if the implementation is correct and `Failure` otherwise.

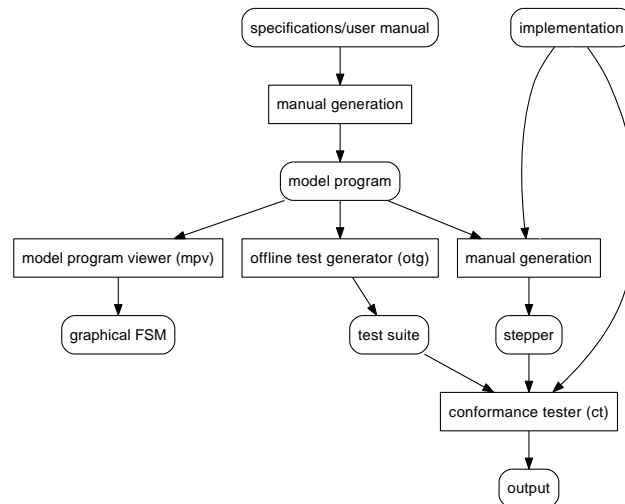**Fig. 1.** Testing implementations with NModel

---

[4] `ct` can also generate test cases on the fly from a model program during test execution, but this is not necessary and is therefore not discussed in this paper.

3 Creating the Mental Model

The first step of our process was to produce a mental model of MACE from the AHLTA-Mobile user manual. The challenge in creating the model was to find an adequate modeling approach that captures the user perception of the application. Taken in its full complexity, the problem of user perception goes well beyond the scope of the case study. However, after showing the AHLTA-Mobile to several potential users, we concluded that the application can be modeled as an extended finite state machine (EFSM), which has been long used in model-based testing [5]. In the following, we give a brief definition of EFSM, followed by the description of our modeling approach and a discussion of the implementation of a given EFSM as a model program in NModel.

3.1 Extended Finite State Machines

Preliminaries. For a finite set of variables $X = \{x_1, ..., x_n\}$, each ranging over the space of values O, a *valuation* is a function $v : X \rightarrow O$ that assigns to each variable x its current value. The set of valuations of X is denoted $\mathcal{V}(X)$. A *predicate P* over X is a boolean-valued function $P : \mathcal{V}(X) \rightarrow \{true, false\}$. A *valuation transformer T* is a function $T : \mathcal{V}(X) \rightarrow \mathcal{V}(X)$.

An EFSM M is a tuple $\langle Q, \Sigma, X, E, q_0, v_0 \rangle$, where Q is a set of states with the designated initial state q_0, Σ is a finite alphabet, X is a set of variables with the initial valuation v_0, and E is a transition relation. A transition $t \in E$ is a tuple $\langle q_1, g, a, u, q_2 \rangle$, where q_1, q_2 are the source and destination states of the transition, respectively. The symbol $a \in \Sigma$ is the event that triggers the transition. The *guard g* is a predicate over the variables of M that states when the transition is allowed to be taken. Finally, the *update u* is a valuation transformer that reflects changes to variables when the transition occurs. For the purpose of this paper, we represent each update as a sequence of assignments $x_i = f_i(X)$.

A *run* of M is an alternating sequence $(q_0, v_0)a_1(q_1, v_1)a_2...$ such that, for each i, M has a transition $\langle q_{i-1}, g_i, a_i, u_i, q_i \rangle$ such that $g_i(v_{i-1}) = true$ and $v_i = u_i(v_{i-1})$. That is, in every step of the execution, a transition of M is taken such that its guard is satisfied by the variable values in the source state and the valuation after the transition is taken is updated according to the update specified by the transition. The update occurs by performing assignments in their syntactic order in u_i.

3.2 EFSM Model for AHLTA-Mobile

The AHLTA-Mobile user manual uses two ways to convey the expected behavior of the application to the user: first, it offers pictures of each GUI screen, and second, it describes the actions that may be performed when a given screen is displayed. With the exception of editing actions, the outcome of performing an action is the new screen being displayed. We found it natural from the documentation to formulate the mental model as an EFSM that encompasses the

observable behavior of the system, identifying screens with states and actions with transitions between the screens.

For this case study we focus on a subset of MACE's behavior, capturing the actions Resume and Suspend. The resulting EFSM is $M_{AM} = \langle Q_{AM}, \Sigma_{AM}, X_{AM}, E_{AM}, \mathsf{StartScreen}, v_0 \rangle$, where

- $Q_{AM} = \{\mathsf{StartScreen}, \mathsf{MACE1}, \dots, \mathsf{MACE7}, \mathsf{MACEResults}, \mathsf{ExamIndex}, \mathsf{ResumeScreen}, \mathsf{NoUnit}\}$,
- $\Sigma_{AM} = \{\mathtt{Edit}, \mathtt{Next}, \mathtt{Suspend}, \mathtt{Resume}, \mathtt{Select}, \mathtt{Start}, \mathtt{MACE}\}$,
- $X_{AM} = \{\mathtt{Edited1}, \dots, \mathtt{Edited7}, \mathtt{EditedResults}, \mathtt{Suspended}, \mathtt{Selected}, \mathtt{UnitInfo}\}$

E_{AM} is visually represented in Figure 2, somewhat simplified for readability, and v_0 is discussed below.

In the EFSM model, states represent the following subset of AHLTA-Mobile screens relevant to MACE. MACE is comprised of eight screen states, MACE 1 through MACE 7, and MACE Results. Each screen is a form that has to be completed before the next may be displayed. The Start Screen is the initial screen where the application begins after the user has logged in and a patient has been selected. The Exam Index is a menu from which the user can navigate to MACE, and the Resume Screen is a menu from which the user may resume a suspended exam.

The alphabet of this EFSM consists of the following actions available within MACE.

- Edit completes the required fields in the current screen.
- Next clicks Next to navigate to the next screen.
- Suspend clicks Suspend to suspend the evaluation.
- Resume clicks Resume to resume the evaluation.
- Select selects the appropriate exam to resume.
- Start clicks Exam Index on the initial screen.
- MACE clicks MACE within the Exam Index.

Each action would label a transition in the EFSM representation of the mental model of MACE. Note that the Edit action is the only one that does not correspond to the invocation of a particular user interface element. In our approach, we do not model the contents of MACE forms. Instead, we capture only the fact that some editing has to be performed before the user can move to the next screen.

The variables of the EFSM model have been introduced to capture conditional execution of user actions as specified in the user manual. For example, the action Resume may only be executed from the Start Screen state if the value of the boolean variable Suspended is *false*, indicating that the exam has been previously suspended. Other variables include UnitInfo, which indicates whether unit information has been previously specified for the patient, Selected, which indicates whether an exam has been selected in the Resume Screen state to resume, and Edited1...EditedResults indicate whether the required fields in

the MACE screens MACE 1...MACE Results have been completed. Initial values of Suspended, Selected, and Edited1...EditedResults are all *false*.We have separately considered initial valuations with UnitInfo either *true* or *false*. The reason for this is that the patient's unit information is set in another part of the AHLTA-Mobile system, which has been excluded from the case study.
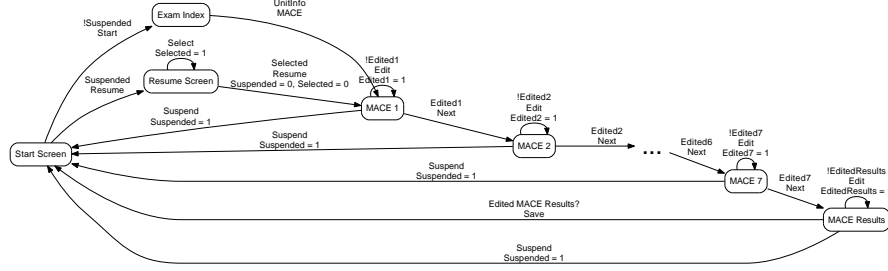

Fig. 2. Expected behavior the MACE exam

3.3 Model Program Representation of the Mental Model

After creating a formal representation of the mental model we needed to translate it into a *model program* that could be used by NModel to test the AHLTA-Mobile application. Model programs, executable specifications written in C# using the NModel library, are action oriented. They define which actions in an application may be taken in what circumstances.

A model program contains a set of variables that captures the state of the model program, and a collection of methods that represent actions. For each action a, the model program contains two methods, a() that represents the action itself, and aEnabled() that, based on the current state of the model program, determines whether a is enabled. The body of a() is a collection of cases that update the variables of the model program when the action method is invoked. Given an EFSM $M = \langle Q, \Sigma, X, E, q_0, v_0 \rangle$ that defines states as screens, we mechanically translate it into a model program as follows:

1. Declare the Model class, which references the System and NModel libraries.
2. Within Model, create all variables in X and initialize them according to v_0. Add the string variable current that stores the label of the current state of M, initially q_0.
3. For each a in Σ:
   (a) Create an action skeleton:
   ```
   [Action("a")]
   static public void a() {}
   static bool aEnabled() { return false; }
   ```
   (b) For each $t = \langle q, g, a, u, q' \rangle$ in E:

i. Add the following lines to `a()`:
```
if (current.Equals("q") && g)
{
  current = "q'";
  update(u);
}
```
where `update(u)` updates all variables according to u. Given our sequential interpretation of u in the definition of EFSM, the assignments of u can be syntactically transcribed into C# statements.

ii. Add the following line to the beginning of `aEnabled()`:
```
if (current.Equals("q") && g) return true;
```

The described translation is mechanical and can be easily made automatic. However, in the case study, we followed the described procedure manually. As an example of this translation, consider the action MACE in Figure 3. The action may be taken when the Exam Index screen is displayed and may lead to No Unit or MACE 1 depending on the value of `UnitInfo`. This fragment of the EFSM yields the model program shown below it.
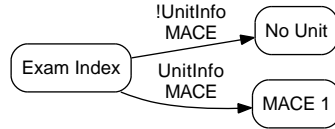

```
[Action("MACE")]
static public void MACE()
{
    if (unitinfo) current = "MACE 1";
    else current = "No Unit";
}
static bool MACEEnabled()
{
    return current.Equals("Exam Index");
}
```

Fig. 3. Representing the `MACE()` action in a model program

After writing the model program representation of the MACE mental model, we used `mpv` to produce an FSM. We then used `otg` to automatically generate a test suite for MACE.

4   Writing a Test Harness

NModel requires the use of a *stepper*, a test harness that invokes an instance of the implementation to be tested and causes the appropriate actions to be

executed when invoked by `ct`. For simple applications, like the samples provided on the NModel website[6], when `ct` requests for an action to be executed, the stepper is written to simply call a corresponding method that exists within the implementation. In AHLTA-Mobile this was not possible: our actions did not directly correspond to single methods provided in the application but instead to multiple methods triggered by user input events, like keystrokes and mouse clicks.

Attempting to associate input actions with existing methods, like callbacks for buttons, was problematic for a few reasons. Since callback methods are normally private, code needed to be modified in many places for them to be used; an inelegant solution that presented many possibilities for errors to be introduced. We also needed to know which instance of any object we were manipulating, requiring further additions to the source code. This method also required detailed knowledge about how the implementation worked, which was both tedious and, as we found, unnecessary.

Instead of using callbacks and related methods in order to simulate actions, we inserted actual keystroke and mouse click events into the application's message loop. We did this in AHLTA-Mobile by retrieving object handles from the C# message loop within the application and sending our user input events directly to the appropriate handles via the message loop. This method allowed us to add code in only one part of the application, making it simpler to work with and reducing the opportunities to introduce errors into the application.

5   Testing AHLTA-Mobile

Once a stepper is written we can run `ct` with the coupled implementation and stepper and the test suite generated by `otg` as arguments, as shown previously in Figure 1.

Running the conformance tester quickly revealed an error: Suspending an exam does not lead to the Start Screen as expected but instead to the Exam Index. This resulted in a timeout, a function included in NModel in case an implementation does not behave as expected and must be terminated. Since the Start Screen and thus Resume button never appeared, it was never clicked and no exam could be selected, causing the application to stall. The test trace that caused the error is given below.

```
TestResult(0, Verdict("Failure"), "Action timed out",
    Trace(
        Test(0), Start(), MACE(), Suspend(), Resume(), Select()
    )
)
```

6   Related Work

The use of state machines for specifying user interfaces has been explored as early as mid-1980s in [17]. At that time, however, state machines were applied

to textual user interfaces, which are much simpler to model and analyze (for example, they do not involve callbacks). With the advent of flexible, dynamically modifiable GUI systems research in the human-computer interface (HCI) area has focused primarily on dynamic aspects of GUI-based systems, where state machines appear to be less useful. However, in the domain of GUI-driven handheld devices considered in our case study, EFSMs are quite appropriate and yield high-level and accurate models of user expectations of the system.

Model-based testing of GUI programs is also explored in [9], where the authors use randomized online testing instead of providing offline tests that achieve transition coverage of the model. The paper presents a very different modeling approach based on labeled transition systems with concurrency. The approach involves two levels of models. A high-level model describes the various user-level actions that may be performed. A user level action may require several GUI operations, such as popping up a menu and then selecting an item in the menu. A low-level model then describes how these actions are accomplished. We believe that the approach of [9] is targeted towards systems with dynamically created and manipulated GUI screens. In our case, their multi-level approach would be an overkill.

Several other research works focus on different aspects of model-based testing. [13] mentions the use of model based test case generation for fault detection, and employs hierarchical predicate transition Petri Nets as a formalism. [18] discusses and compares several testing methodologies toward open source software using model based testing.

In [12], the authors present extensions to the Spec Explorer tool to automate testing based on Spec# specifications. A GUI mapping tool allows the tester to associate actions with physical objects that appear on the GUI display. The tool generates C# code with methods that have the same signature as those specified and actions are performed externally according to tests generated by Spec Explorer. [14] specializes the task modelling notation to ConcurTaskTrees.

7   Conclusions and Discussion

In this paper we presented an approach for the behavioral modeling of GUI-driven handheld devices. We illustrated how the NModel methodology can be applied for the model-based testing of this class of devices. We discussed the challenges we faced in applying this approach and our way of overcoming them. Finally, we presented the results of our case study of the AHLTA-Mobile application, demonstrating an inconsistency between the observed behavior and the behavior described in the user manual. We believe that our approach is applicable to most GUI-driven handheld devices, offering a viable method of establishing conformance between system operation and user expectations for these types of reactive systems.

While this work is an encouraging step forward, it is still far from the comprehensive methodology needed for the analysis of user-centric GUI-driven devices

that we envision. Several aspects of such a methodology remain open problems, as discussed below.

Realistic mental models. In this paper, we constructed the mental model based on the contents of a user manual. Clearly, perception of the appropriate use for a device by a user is formed through other factors as well and can be quite different from the literal representation of the user manual in formal notation [11,10]. Empirically constructed mental models, capturing probabilistic information about observed user behaviors, are used in testing literature under the name of usage models or usage profiles [4]. A predictive way to construct such mental models is needed, especially for new kinds of devices. A practical mental modeling methodology should build on both cognitive science and computer science.

Detecting and managing underspecification. A big part of the challenge in constructing mental models is that natural-language documents describing a system are never complete and users interpret them by making assumptions based on their knowledge and prior experience with similar systems. The problem here is that these assumptions are so natural for the reader that it is often hard to detect that an implicit assumption has been made.

Representation of alternatives would require us to apply a different modeling and testing approach. A possible way to capture alternatives is to use non-deterministic EFSMs, where different transitions labeled by the same symbol would correspond to different alternatives. During testing, as long as the implementation offers a behavior corresponding to one alternative, a test should succeed. A slight complication here is the need to ensure *consistency*: if an alternative has been resolved in some way during a test execution then later in the same execution it has to be resolved the same way. We also will have to rely on a different tool to generate and execute tests, since NModel operates on input-deterministic model programs.

Soft vs. hard inputs. Many approaches to model-based testing require that the model does not restrict the tester from performing an action. Technically, this corresponds to the notions of weak input enabledness [15] or input completeness [8]. In our case, this requirement should be relaxed, because the system interacts with its environment via what we call soft inputs, such as GUI buttons on the screen. Such a button may not be present on some screens, and in that case the tester should actually be prevented from invoking that input.

References

1. R. Alur, D. Arney, E. L. Gunter, I. Lee, J. Lee, W. Nam, F. Pearce, S. Van Albert, and J. Zhou. Formal specifications and analysis of the computer-assisted resuscitation algorithm (CARA) infusion pump control system. *Software Tools for Technology Transfer*, 5(4):308–319, 2004.
2. AHLTA-Mobile fact sheet. Medical Communications for Combat Casualty Care Web Site. `https://www.mc4.army.mil/AHLTA-Mobile.asp`.
3. D. Arney, R. Jetley, P. Jones, I. Lee, and O. Sokolsky. Formal methods based development of a PCA infusion pump reference model: the generic infusion pump (GIP) project. In *Joint Workshop on High-Confidence Medical Devices, Software*

    and Systems and Medical Device Plug-and-Play Interoperability, pages 23–33, July 2007.

4. P. Brooks and A. M. Memon. Automated GUI testing guided by usage profiles. In *Proceedings of the 22nd IEEE International Conference on Automated Software Engineering (ASE '07)*, November 2007.

5. K.T. Cheng and A.S. Krishnakumar. Automatic functional test generation using the extended finite state machine model. In *Proceedings of the 30th international conference on Design automation (DAC '93)*, pages 86–91, June 1993.

6. Microsoft Corporation. NModel website, 2009. `http://www.codeplex.com/NModel`.

7. J. Jacky, M. Veanes, C. Campbell, and W. Schulte. *Model-based Software Testing and Analysis with C#*. Cambridge University Press, 2008.

8. C. Jard and T. Jéron. TGV: theory, principles and algorithms. a tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Software Tools for Technology Transfer*, 2004.

9. A. Kervinen, M. Maunumaa, T. Pääkkönen, and M. Katara. Model-based testing through a GUI. In *Formal Approaches to Software Testing*. Springer, 2006.

10. P. Legrenzi and V. Girotto. Mental models in reasoning and decision making. In A. Garnham and J. Oakhill, editors, *Mental models in cognitive science*, pages 95–118. 1996.

11. C. Lewis. A model of mental model construction. In *Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '86)*, pages 306–313, 1986.

12. A. Paiva, J. Faria, N. Tillmann, and R. Vidal. A model-to-implementation mapping tool for automated model-based GUI testing. *Formal Methods and Software Engineering*, pages 450–464, 2005.

13. Hassan Reza, Sandeep Endapally, and Emanuel S. Grant. A model-based approach for testing gui using hierarchical predicate transition nets. In *ITNG*, pages 366–370, 2007.

14. José L. Silva, José Creissac Campos, and Ana C. R. Paiva. Model-based user interface testing with spec explorer and concurtasktrees. In *Electronic Notes in Theoretical Computer Science*, volume 208, pages 77–93. 2008.

15. J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17(3):103–120, 1996.

16. U.S. Army Medical Research & Material Command, Mobile Computing Group, Telemedicine and Advanced Technology Research Center, Fort Detrick, Maryland. *AHLTA-Mobile User Manual, v2.2.61*.

17. A.I. Wasserman. Extending state transition diagrams for the specification of human-computer interaction. *IEEE Transactions on Software Engineering*, 11(8):699–713, August 1985.

18. Qing Xie and Atif Memon. Model-based testing of community-driven driven open source GUI applications. In *22nd International Conference on Software Maintenance (ICSM'06)*, pages 145–154, 2006.