Model Checking User Interfaces

Abigail Cauchi
Dept of Computer Science
University of Malta
acau0004@um.edu.mt

Gordon Pace
Dept of Computer Science
University of Malta
gordon.pace@um.edu.mt

Sandro Spina
Dept of Computer Science
University of Malta
sandro.spina@um.edu.mt

Abstract

User interfaces are crucial for the success of most software projects. As software grows in complexity there is a similar growth in the user interface complexity which leads to bugs which may be difficult to find by means of testing. In this paper we use the method of automated model checking to verify user interfaces with respect to a formal specification. We present an algorithm for the automated abstraction of the user interface model of a given system, which uses asynchronous and interleaving composition of a number of programs. This technique was successful at verifying the user interface of case study and brings us one step forward towards push button verification.

Index Terms

Model Checking, User Interfaces, Formal Verification, Temporal Logic, Human Computer Interaction

1. Introduction

The human component of software systems is often overlooked, but it is an extremely important factor in the success of a software project. A good number of system failures (60% to 90%) have been attributed the lack of handling of possible erroneous human actions which contribute to an unpleasant user experience [1]. Catering for avoiding the possibility of erroneous human actions is therefore crucial. One should also check the user interface for bugs. As the user interface grows larger, the possibility of introducing bugs, similarly becomes larger, possibbly becoming more difficult to find by means of testing.

From relevant work such as [2], [3], the missing block is the automation of the abstraction process; this is the main focus of this paper. In this paper we present a technique which can be used to automatically abstract the user interface of a given program and modeling user behavior. The resulting model is used for verification by model checking. Since we are dealing with reactive systems, models are built using a labeled transition system (LTS) and temporal logic is used to for formal specification. We present a tool which was implemented using this approach and show how it can be used in a practical scenario.

2. Background

Two main tasks required for model checking are building a formal model of the system to be verified and formalizing a specification which the model should satisfy. The model checking problem is whether the model satisfies the specification. A model checker's output indicates whether the specification is satisfied or not, and possibly generates a diagnostic to demonstrate the result. The diagnostic may then be visualized to help tracing a possible bug. The problem with model checking is that the process is exponential, however, implementing abstractions on a complex model might still render the model checking process to be feasible. This however, may introduce false positives, i.e. the model checker may verify that the property does not hold when it does.

User interfaces are *reactive systems*; these are described as systems which continuously interact with their environment and often do not terminate [4], [5]. Since termination is not guaranteed, we cannot reason about the system's input-output behavior, but we need to reason about the system's state which captures the values of variables at a particular instance of time [4]. In order to model our system, we need to describe how the state of the system changes when a particular action occurs. These state transitions may be described by transition relations between the state before an action occurs and the state after the action occurs [4]. To

formalize specifications about user interfaces, temporal logic is ideal since models contain several states and it has a dynamic notion of truth which allows a formula to hold in some states and not hold in others.

The regular alternation free mu-calculus is a temporal logic which allows us to specify safety, liveness and fairness properties about labeled transition systems (LTSs) [6]. The regular alternation free mu-calculus is made up of three types of formulas: action formulas, regular formulas and state formulas; the syntax in BackusNaur form (BNF) of each is given below.

Action formulas are made up of action formulas, labels and boolean operators. In the BNF below, A is an action formula and a is a label in the LTS. The boolean operators have the following semantics: a label is always said to hold, a label satisfies $\neg A$ iff it does not satisfy A; it satisfies $A_1 \vee A_2$ iff it satisfies A_1 or it satisfies A_2; it satisfies $A_1 \wedge A_2$ iff it satisfies both A_1 and A_2; it satisfies $A_1 \Rightarrow A_2$ iff it does not satisfy A_1 or it satisfies A_2; it satisfies $A_1 \equiv A_2$ iff either it satisfies both A_1 and A_2, or none of them.

$$A ::= a \mid \neg A \mid A \vee A \mid A \wedge A \mid A \Rightarrow A \mid A \equiv A$$

A regular formula R, as described in the BNF below, is made up of nil, the empty operator, . the concatenation operator, $|$ the choice operator, $*$ the transitive and reflexive closure operator, and $+$ the transitive closure operator. The semantics of each are the same as those for regular expressions.

$$R ::= a \mid nil \mid R.R \mid R \,|\, R \mid R* \mid R+$$

In he BNF below, S is a state formula, $< R > F$ and $[R]F$ are the possibility and necessity modal operators, $@(R)$ is the infinite looping operator, $\mu X.F$ and $\nu X.F$ are the minimal and maximal fixed point operators, and X is a propositional variable. The fixed point operators act as binders for the variables X in a way similar to quantifiers in first-order logic. In each meaningful $\mu X.F$ or $\nu X.F$ formula, X is supposed to have free occurrences inside F. State formulas are assumed to be syntactically monotonic (i.e., in each fixed point formula $\mu X.F$ or $\nu X.F$, free occurrences of X in F may appear only under an even number of negations and/or left-hand sides of implications) and alternation-free (i.e., without mutually recursive minimal and maximal fixed point variables). Syntactically, all binary operators on state formulas are left-associative.

$$S ::= true \mid false \mid \neg S \mid S \vee f \mid S \wedge S \mid S \Rightarrow S \mid S \equiv S \mid < R > S \mid [R]S \mid @(R) \mid X \mid \mu X.F \mid \nu X.F$$

2.1. The Model Checking Process

We shall demonstrate the model checking process with a text processor model. In table 1, the text processor is modeled and specified, after being verified, the counterexample generated may be seen in the same table. The text processor is modeled as a labeled transition system on which a specification written in the regular alternation free mu-calculus is verified. The text processor starts at the `Text Processing` state and when a help file is requested, it may either go to the `Idle` state or to the `Help File Display` state. From `Idle`, it may either display the help file, loop in `Idle` or continue processing. It is required that when the user requests a help file, it is always displayed at some later state. This property is formalized in table 1. From the LTS described, one can see that this property does not hold since the text processor may loop to infinity in the `Idle` state. After verifying the model, the model checker may produce a counterexample or a family of counterexamples to show why the property does not hold as shown in table 1.

2.2. Dealing with the State Explosion Problem

As mentioned previously, a problem one needs to deal with when model checking a system is the state explosion problem. In order to be able to verify complex systems, among other techniques (mentioned in [4]) abstraction may be used. When reasoning about reactive systems that involve data paths, the use of abstraction seems to be essential [4].

Given a programming or specification language, abstract interpretation consists in giving several semantics linked by relations of abstraction [7]. A semantic is a mathematical characterization of a possible behavior of a system. A precise abstraction describes the actual execution of the system very closely; this are called the concrete semantics. The concrete semantics of the type of system we would like to verify is a set of execution traces where each state corresponds to a line of code. This type of abstraction is the first step towards modeling the user interface.

Due to the state explosion problem of model checking, we are mostly interested in types of abstractions which under-approximate the system by removing states which are irrelevant to the types of properties being verified. There are several ways of doing this, the simplest way would be by retaining states which are directly relevant, such as lines of code which

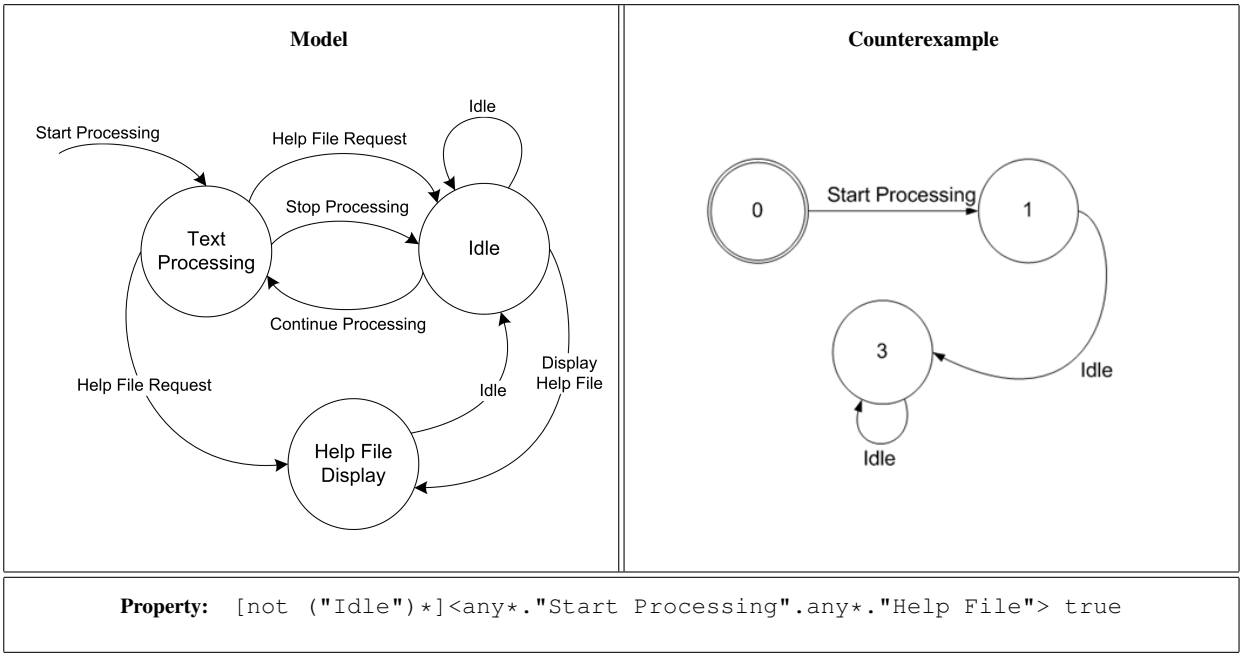

Table 1. The Model Checking Process

are directly related to user interfaces. Other under-approximations are described in [2] which include the grouping of related components and the consideration of independent subsystems which may be verified separately, such as different windows.

3. Human-Computer Interaction

HCI is the study of interaction between people and computers. It is often regarded as the intersection of computer science, behavioral sciences, design and several other fields of study. Interaction between users and computers occurs at the user interface. A sister field of research to HCI is MMI (Man-Machine Interaction) or HMI (Human Machine Interaction). In [8] a definition is given as follows: "Human-computer interaction is a discipline concerned with the design, evaluation and implementation of interactive computing systems for human use and with the study of major phenomena surrounding them."

In HCI we are mostly concerned with usability issues. In [9], Nielsen defines usability is as follows: "Usability is a quality attribute that assesses how easy user interfaces are to use. The word "usability" also refers to methods for improving ease-of-use during the design process." Alternatively, in [10], Bevan interprets usability as "quality of use". What Bevan points out as a flaw in Nielsen's definition of usability is that software can have a user interface which is easy enough to use making the software usable in Nielsen's sense, but the features the software might have might be useless and cannot be used as a utility. Therefore a more refined definition of usability would be how capable a software package is at helping users achieve their intended goals, providing a quick way of doing things by a good user interface design.

Common user interface design principles have been identified by various papers such as [9] and [11], some of which are listed below:

1) Strive for consistency
2) Enable frequent users to use shortcuts
3) Offer informative feedback
4) Design dialog to yield closure
5) Offer simple error handling
6) Permit easy reversal of actions
7) Support internal locus of control
8) Reduce short-term memory load

Bugs in coding may lead to other situations not mentioned; specifying a good design and implementing it make way for different types of errors. Apart from the need for assurance that a user interface is designed following good principles, one should also check whether it is implemented correctly.

4. Model Checking User Interfaces

From figure 1 we can get a general picture of the process of model checking user interfaces. A system is abstracted to obtain a model of its user interface which is translated and input to the model checker. Subsequently, properties are specified, formalized and input to the model checker. Finally, the model checker is run to obtain a result of whether the property holds or not, and possibly a counterexample is visualized.

Here we shall describe the technique for automating the process of abstracting the user interface model. The systems we are handling are a family of programs, consisting of the `Main` method and a number of event handlers which handle user interaction. To model user behavior, we use the asynchronous and interleaving composition of all these programs. This composition is an automated process handled by the model checker.

To abstract the user interface, we build an interprocedural control flow graph (ICFG) for each program using standard techniques [12]. At this stage, control flow issues, such as the handling of recursion and catering for the object oriented paradigm, should be considered. The next step is to discard all nodes which are irrelevant to the user interface, while retaining the correct control flow and anything contributing to the correctness of the properties we are verifying, such as loops (for the correctness of liveness properties).

Let us look at a method of an event handler and perform the abstraction to get a better idea of this procedure. Consider the event handler code in figure 2. The concrete graph is the first step towards abstraction; a complete ICFG of the event handler is built. The abstract graph is the resulting ICFG after the abstraction is performed. One can note that the only control structure retained here is the structure containing user interface relevant code, and the only line of code retained is the one directly related to the user interface (by the label).

5. Case Study and Evaluation

This technique was used to implement a tool for systems written in C# .NET 1.0 and 2.0 which produces graphs to be used as input to CADP's Evaluator [6] model checker. CADP provides tools for representing LTSs in a compact format, reducing and performing manipulations on them. Evaluator allows us to write properties in the regular alternation free mu-calculus described in section 2.

A case study was carried out on a system developed using C# .NET 2.0 consisting of 2, 631 lines of code. This system is a simulator for a fuzzy logic controller and it is made up of a main window with a main menu. Menu items are to be enabled and disabled when appropriate, and when selected, a new window is opened to allow the user to work on the selected item. 26 properties were defined which were of the following nature:

- **Properties related to the behavior of menu items** – Checking that they are enabled when they should be and that they behave as necessary.
- **Window disposal** – Since the user interface uses several windows, we have checked that when one is disposed, no others are disposed mistakenly.
- **Window resizing** – This system makes use of window resizing and we make sure that this behaves as specified.
- **Error and warning handling** – We checked that errors and warnings are displayed when necessary.
- **Component behavior** – General behavior of components such as buttons and labels were verified.

Popular classes of properties include safety (nothing bad ever happens), liveness (something good eventually happens) and fairness properties (considering only fair execution paths). We see an example of a property from each class.

- **Safety Property** – Two labels are to be enabled in a mutually exclusive fashion.
```
[any*.Label1Enabled.
(not label1Disabled)
*.Label2Enabled ] false
```
- **Liveness Property** – When a user closes a window, eventually it is closed.
```
<any*.CloseWindowCommand.
(any*.Error)*.any*.
WindowClosed> true
```
- **Fairness Property** – When the `Select` button is clicked and no error occurs, the window is resized.
```
[any*.SelectClicked.(not Error)*]
<(not WindowSizeChanged)*.
WindowSizeChanged> true
```

Our model consisted of a total 143 states with 73 different labels. The average time taken for properties to be verified is 45s. The property which took longest to be verified was the one which generated a counterexample, the time included its verification and counterexample generation.

6. Conclusions and Future Work

In this paper we presented a technique to automatically abstract a system's user interface for verification by model checking. The asynchronous and interleaving

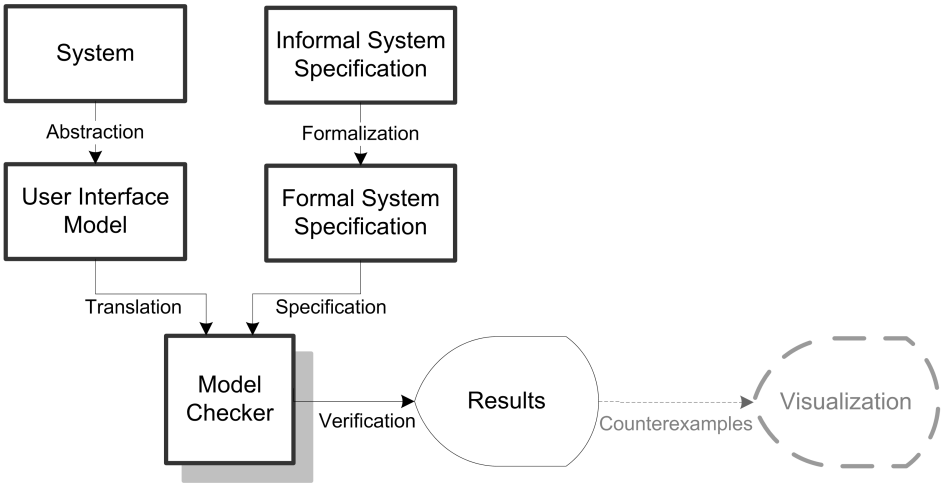

Figure 1. System Architecture

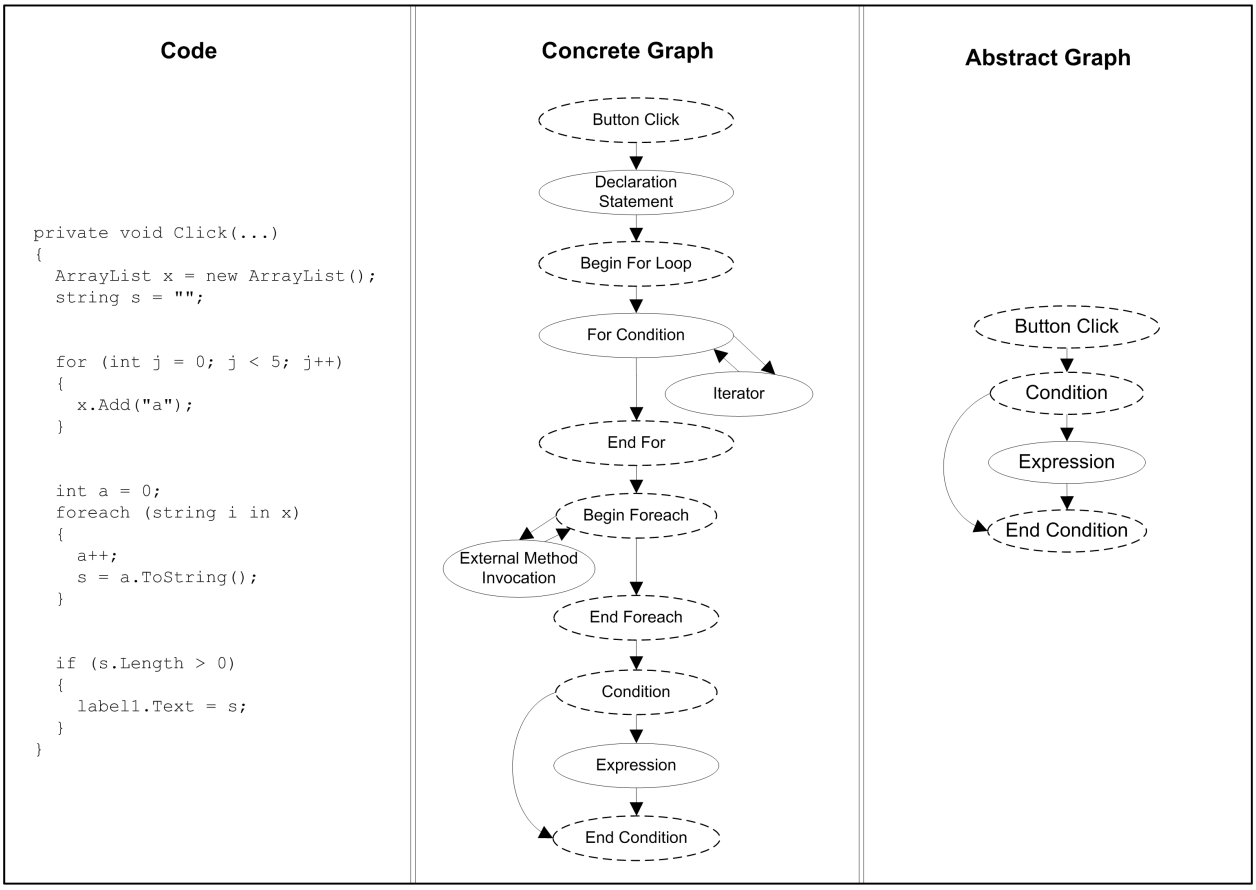

Figure 2. Abstraction Process

composition of a number of programs was used to model user behavior, where this process of composition was automated. A tool was developed implementing this technique and used on a case study to demonstrate how automatic user interface abstraction for verification by model checking may be applied.

A number of possibilities for future work have been identified. One of the most interesting issues is how abstraction is performed. In this paper we abstracted the user interface using an under-approximating abstraction, however one may look into precise approximations by program slicing [13]. Due to the precise approximation and concrete semantics of program slicing, this would work well on a system with a small state space, however, for more complex systems, further abstraction would be necessary. Another form of abstraction is property driven abstraction where if one wants to verify properties about one variable, all irrelevant details are discarded. Combining property driven abstraction with program slicing techniques could be a step towards a precise abstraction relevant to the property being verified. Additionally, in [2] a few possible abstractions on user interfaces are discussed which include grouping fields, considering independent subsystems and the application state.

On another note, web user interfaces also pose an interesting problem, their apparent simplicity may reveal several underlying issues. For example, to model the user interface of a simple HTML page, one should keep in mind client-server interaction. The techniques presented in this paper combined with techniques used to model check website may lead to an interesting analysis.

References

[1] J. Preece and Y. Rogers, *Human-Computer Interaction*. Addison-Wesley, 1994.

[2] M. B. Dwyer, V. Carr, and L. Hines, "Model checking graphical user interfaces using abstractions," in *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, M. Jazayeri and H. Schauer, Eds. Springer–Verlag, 1997, pp. 244–261. [Online]. Available: citeseer.ist.psu.edu/20430.html

[3] R. E. K. Stirewalt and G. D. Abowd, "Composition property analysis: a new strategy for model checking user-interface designs," Department of Computer Science, Michigan State University, East Lansing, Michigan, Tech. Rep. MSU-CSE-99-30, August 1999.

[4] J. Edmund M. Clarke, O. Grumberg, and D. A. Peled, *Model checking*. Cambridge, MA, USA: MIT Press, 1999.

[5] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data-flow programming language LUSTRE," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, September 1991.

[6] H. Garavel, F. Lang, and R. Mateescu, "An overview of CADP 2001," *European Association for Software Science and Technology (EASST) Newsletter*, vol. Volume 4, pp. Pages 13–24, August 2002.

[7] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. New York, NY, USA: ACM, 1977, pp. 238–252.

[8] Hewett, B. Card, Carey, Gasen, Mantei, Perlman, Strong, and Verplank, "Curricula for human-computer interaction," ACM SIGCHI, New York, NY, USA, Tech. Rep., 1992, chairman-Thomas T. Hewett.

[9] J. Nielsen, *Usability Engineering*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995.

[10] N. Bevan, "Usability is quality of use," in *Proceedings of the Sixth International Conference on Human-Computer Interaction*. Elsevier Science, 1995, pp. 349–354.

[11] B. Shneiderman, *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. ADDISON-WESLEY, 1998.

[12] F. Nielson and H. R. Nielson, "Interprocedural control flow analysis," in *European Symposium on Programming*, 1999, pp. 20–39. [Online]. Available: citeseer.ist.psu.edu/nielson99interprocedural.html

[13] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," in *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, vol. 23, Atlanta, GA, June 1988, pp. 35–46. [Online]. Available: citeseer.ist.psu.edu/horwitz90interprocedural.html