[Downloaded from www.aece.ro on Thursday, March 28, 2024 at 15:48:50 (UTC) by 44.200.210.43. Redistribution subject to AECE license or copyright.]

Formal Model for Data Dependency Analysis between Controls and Actions of a Graphical User Interface

Dejan SKVORC, Ivan ZUZAK, Sinisa SRBLJIC
University of Zagreb, School of Electrical Engineering and Computing, HR-10000, Zagreb, Croatia
dejan.skvorc@fer.hr, ivan.zuzak@fer.hr, sinisa.srbljic@fer.hr

***Abstract*—End-user development is an emerging computer science discipline that provides programming paradigms, techniques, and tools suitable for users not trained in software engineering. One of the techniques that allow ordinary computer users to develop their own applications without the need to learn a classic programming language is a GUI-level programming based on *programming-by-demonstration*. To build wizard-based tools that assist users in application development and to verify the correctness of user programs, a computer-supported method for GUI-level data dependency analysis is necessary. Therefore, formal model for GUI representation is needed. In this paper, we present a finite state machine for modeling the data dependencies between GUI controls and GUI actions. Furthermore, we present an algorithm for automatic construction of finite state machine for arbitrary GUI application. We show that proposed state aggregation scheme successfully manages state explosion in state machine construction algorithm, which makes the model applicable for applications with complex GUIs.**

***Index Terms*—computer aided software engineering, formal specifications, graphical user interfaces, programming environments, user centered design.**

I. Introduction

Graphical user interfaces, or GUIs for short, are a predominant technique of interaction with current software-driven systems. A GUI is a type of user interface which allows people to interact with electronic devices by exposing the inputs and outputs of the device as a set of visually represented controls and enabling user-driven actions upon these controls. A GUI uses a combination of technologies and devices to provide an environment the user can interact with, in order to gather or produce information. The most common form of GUI in the field of personal computers is the WIMP paradigm [1]. This paradigm uses physical input devices to control the system's data input as well as the position of a cursor, and outputs the information on to a display device. Available commands to control the target system are compiled together in windows, menus, and icons and acted upon through physical input devices, such as keyboard and mouse.

In recent years, especially with the advent of Web 2.0, automation of GUI applications has been more actively researched and a number of GUI automation tools have been developed. Such tools allow users to build customized programs that automate operations over the applications' GUI and integrate content from multiple sources into a coherent whole. The particular purpose of GUI automation tools ranges from web automation and testing to customization and even application development.

Although the development of user programs in majority of such tools still relies on low-level technologies like parsing *HTML*, scripting with *JavaScript*, and styling with *CSS*, recent tools like *Chickenfoot* [2], *Marmite* [3], *Geppeto* [4], *Selenium* [5], and *Sikuli* [6] operate at a GUI level. In this type of programming paradigm, programming primitives describe GUI operations like filling in forms, clicking on links and buttons, or selecting items from drop-down menus. Because of their closeness to human perception of the GUI, these characteristics make such tools appropriate even for non-programmers.

However, in order to verify the correctness of such GUI programs, models that describe the behavior of applications at the GUI level are required. Furthermore, the definition of correct usage of GUIs must be appropriately defined with respect to GUI dynamics and integrated into models. Additionally, the practical applicability of models depends on both their simplicity and expressiveness as requirements for supporting automatic construction of application descriptions and effective verification.

Although many models of GUI applications have been previously developed with the specific purpose of GUI testing, our work has complementary motivation and requirements. In GUI testing [7], [8], GUI applications are being verified and the premise is that GUI-level programs that test them are correct, whereas our work focuses on cases where GUI applications are known to be correct and GUI-level programs require verification.

In this paper, we present a formal model for describing data dependencies among controls and operations of a graphical user interface. Since data dependencies at a GUI level dictate the temporal ordering of GUI operations for reaching the desired state of the application, we use this as the basis for the definition of correct GUI usage. The formal model for describing data dependencies is constructed in a form of a finite state machine [9], where states describe the state of GUI controls, while transitions describe GUI operations. We describe how such finite state machines may be efficiently generated automatically based on a simple description of GUI structure and operation-level data dependencies. Furthermore, we explain how the formal model may be used both to guide the construction of GUI-level programs and to verify temporal characteristics of GUI-level programs prior to their execution.

This work was supported by the Croatian Ministry of Science, Education, and Sports under Grant 036-0362980-1921.

Digital Object Identifier 10.4316/AECE.2012.01011

1582-7445 © 2012 AECE

[Downloaded from www.aece.ro on Thursday, March 28, 2024 at 15:48:50 (UTC) by 44.200.210.43. Redistribution subject to AECE license or copyright.]



Figure 1. A sample web application GUI

The rest of the paper is organized as follows. In Section 2, we introduce basic concepts of data dependency at the GUI level. In Section 3, we construct a finite state machine describing data dependencies for a single GUI element, while in Section 4 we give rules for construction of application-level finite state machines. In Section 5, we outline the algorithm for construction of application-level finite state machines. In Section 6 and 7, we discuss the efficiency of the presented model and algorithm, and discuss model usage. In Section 8, we compare our research to the work done in the field of GUI testing, while Section 9 concludes the paper.

II. Data Dependencies at GUI Level

The algorithm for construction of a formal model in a form of a finite state machine is exemplified through a sample web application GUI shown in Fig. 1. The purpose of the given application is to support online translation of natural languages. The GUI consists of two text areas, two drop-down menus, and two button controls. The input text area is used to enter the text in the source language, while the output text area shows the translated text. The drop-down menus are used to select the source and target languages, while buttons are used to start the translation of source text and pronunciation of translated text.

To automate the usage of the web application shown in Fig. 1, the user may construct a program consisting of primitives that mimic the human operation on GUI elements. Typing text into the input text area, selecting an item from the *From* menu, selecting an item from the *To* menu, clicking the *Translate* button, clicking the *Pronounce* button, and copying text from the output text area into some other input element are six different operations that can be performed over the given set of GUI elements.

```
G = ( E, O )

E = { Input Text,
      From, To,
      Translate, Pronounce,
      Output Text
    }

O = { write Input Text,
      select From,
      select To,
      click Translate,
      click Pronounce,
      read Output Text
    }
```

Figure 2. Abstraction of a web application GUI with a set of GUI elements and GUI operations

To achieve a certain task, the user program will perform a sequence of the listed operations which will guide the application to the desired state. If the GUI is used manually by an end-user, using a keyboard and pointing device, the proper ordering of operations is inferred by the user herself from the application semantics, as well as visual layout, labeling, and grouping of the GUI controls. For example, there is no sense in clicking the *Translate* button if the input text area is empty or any of the languages is not selected. Even if a faulty interaction occurs, manual operation allows the user to correct herself through multiple trial-and-error cycles.

However, to check the correctness of a user program automatically, we need to check whether the ordering of programming primitives is a correct and sensible usage of the GUI. For this reason, GUI data dependencies may be used to determine the correct temporal ordering of GUI operations for reaching the desired state of the application.

To construct a finite state machine for describing the correct GUI usage, information regarding both the structure and the dynamics of the GUI is needed. On a higher level of abstraction, a GUI can be modeled as an ordered pair consisting of a finite set of GUI elements and a finite set of GUI operations. Fig. 2 shows the abstract model for the web application GUI shown in Fig. 1, where G represents an ordered pair consisting of a finite set of GUI elements E and finite set of GUI operations O. As shown in Fig. 2, the GUI of the given web application consists of six elements which support six different operations to be performed by an end-user.

TABLE 1. DOMAINS AND CO-DOMAINS OF GUI OPERATIONS

Operation	Domain	Co-domain
write *Input Text*	∅	*Input Text*
select *From*	∅	*From*
select *To*	∅	*To*
click *Translate*	*Input Text*, *From*, *To*	*Output Text*
click *Pronounce*	*Output Text*	∅
read *Output Text*	*Output Text*	paste target element

From a GUI-level data dependency analysis perspective, GUI elements are elementary units, while GUI operations are complex structures which can be further described with two sets, the domain and the co-domain of the operation. The domain of an operation is a set of GUI elements which are used as inputs for the given operation. On the other hand, the co-domain of an operation is a set of GUI elements where the results of the operation are written to. Table 1 shows the domain and co-domain for each of the GUI operations of the web application shown in Fig. 1.

III. Element-Level Finite State Machine

GUI elements of an application can be divided into two groups. The first group, which we call *content-sensitive elements*, consists of elements that contain application-generated or user-generated content. For example, elements named *Input Text*, *From*, and *To* contain user-generated content since user defines the values contained in these elements. On the other hand, *Output Text* contains

[Downloaded from www.aece.ro on Thursday, March 28, 2024 at 15:48:50 (UTC) by 44.200.210.43. Redistribution subject to AECE license or copyright.]

application-generated content since the content of this element is generated by internal application logic. The remaining elements are called *content-free elements*. For example, control buttons *Translate* and *Pronounce* are content-free elements because they are used only to start certain operations and do not change their content over time.

We aggregate all possible states of each content-sensitive GUI element during the execution of the application into three abstract states. The content of the GUI element can be either *defined* or *undefined*. If the content of the element is defined, we can further make a distinction between *refreshed* and *used* content. We introduce three labels

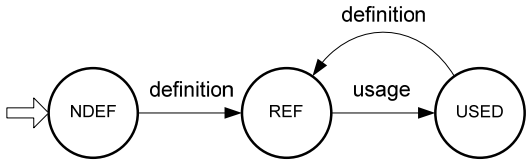


Figure 3. Finite state machine describing the dynamics of a single GUI element

representing the three basic states of the GUI elements: *NDEF*, *REF*, and *USED*. *NDEF* denotes that the content of the element is not yet defined, *REF* denotes that the element contains refreshed content, while *USED* denotes that the content of the element was used by an operation.

We model the transitions of GUI elements between the three basic states by the finite state machine shown in Fig. 3. Initially, elements have no content defined and therefore are in the *NDEF* state. Once the content of the element is defined by an operation, the element changes its state to *REF*. Once another operation uses the previously refreshed content, the element enters the *USED* state. After usage, the content of the element can be redefined, which lead the element back to the *REF* state. Other transitions between the described states are considered as incorrect usage of the GUI element. For example, using the element with undefined content is not allowed.

The execution of particular GUI operation impacts the content of GUI elements that belong to its domain and co-domain. If the element belongs to the domain of the given operation, then the operation uses the content of the element. Otherwise, if the element belongs to the co-domain of the given operation, the operation refreshes the content of the element. Hence, we can distinguish between two types of operations, *definition* and *usage*, to model the effect of any GUI operation. The finite state machine shown in Fig. 3 uses these two abstract types of GUI operations to model the transitions between GUI element states.

IV. Application-Level Finite State Machine

The finite state machine introduced in Section 3 models the transitions of a particular GUI element in response to different types of operations performed over that element. In order to model the GUI dynamics of an entire application, we extend the element-level state machine to the application-level state machine. The application-level FSM is a composition of element-level FSMs which respects data dependencies of all GUI elements within an application and guides the ordering of GUI operations.

A. Input data

In order to construct state machines describing GUI element dynamics, the structure of the application GUI and the dynamics of operations must be known. The structure of a GUI, as exemplified in Fig. 2, is given with expression (1):

$$G = (E,\ O) \tag{1}$$

where E is the set of GUI element names and O is the set of operation names. Also, as exemplified in Table 1, each operation Op must be defined by its domain and co-domain sets D_{op} and C_{op}:

$$Op = (D_{op},\ C_{op}), D_{op} \subset E, C_{op} \subset E \tag{2}$$

The set of content-sensitive elements CE may be computed from (1) and (2) using expression (3):

$$CE(e,\ O) : \exists x \in O \mid (e \in D_x \vee e \in C_x) \tag{3}$$

which means that GUI element e is considered content-sensitive if it appears as either domain or co-domain of at least one GUI operation.

B. FSM state construction

The basic finite state machine described in Section 3 shows how a particular GUI element changes states in response to GUI operations. However, in a general case, the GUI of an application contains more than one GUI element. Hence, the first step in construction of the application-level finite state machine is to change the way we construct the FSM states in order to represent the state of the entire GUI, instead of a single GUI element.

The states of the application-level FSM are constructed as multi-dimensional vectors. Each vector has N dimensions, where N is the number of content-sensitive GUI elements. Each dimension of the multi-dimensional FSM state represents the state of exactly one GUI element. For example, each state of the application-level FSM for sample application shown in Fig. 1 has four dimensions, since the GUI of the given application has four content-sensitive elements. The first dimension of the four-dimensional state represents the state of the *Input Text* textual area, the second dimension represents the state of the *From* drop-down menu, the third represents the state of the *To* drop-down menu, while the fourth dimension represents the state of the *Output Text* textual area.

Each dimension of the multi-dimensional FSM state can be assigned one of three possible values: *NDEF*, *REF*, or *USED*. In the case of the web application shown in Fig. 1, the FSM state [*NDEF*, *NDEF*, *NDEF*, *NDEF*] represents the GUI state where all content-sensitive GUI elements are not yet defined. This represents the state of the application at the beginning of execution. Similarly, FSM state [*REF*, *REF*, *REF*, *NDEF*] represents the GUI state where input text is entered into *Input Text* textual area, and the source and target languages are selected from *From* and *To* drop-down menus, but the content of the *Output Text* textual area is not yet defined. Such a state occurs immediately before the

[Downloaded from www.aece.ro on Thursday, March 28, 2024 at 15:48:50 (UTC) by 44.200.210.43. Redistribution subject to AECE license or copyright.]


```
Σ = O = { write Input Text,
          select From,
          select To,
          click Translate,
          click Pronounce,
          read Output Text
        }
```

Figure 4. The alphabet of an application-level finite state machine for the web application shown in Fig. 1

Translate button is ready to be pressed.

Since each application has a finite set of GUI elements and each dimension of the multi-dimensional FSM states has exactly three possible values, we can calculate the maximum number of states required to construct the application-level FSM. The maximum number of states is given by expression (4):

$$|Q| = 3^{|CE|} \tag{4}$$

where Q represents the set of states of the FSM, while $|CE|$ is the number of content-sensitive GUI elements. Since the maximum number of FSM states is finite for any given application, expression (4) confirms the feasibility of construction of the application-level FSM using multi-dimensional states.

C. FSM alphabet construction

Once the set of states of the application-level FSM is created, the second step is construction of the alphabet symbols to manage the transitions between states. Since the states of the application-level FSM reflect the state of the content-sensitive GUI elements, while the states of GUI elements are impacted by executing GUI operations, the alphabet of the application-level FSM is derived from these operations. The alphabet of the application-level FSM corresponds to the set of GUI operations for the given application. As shown in Fig. 4, the FSM alphabet for the web application shown in Fig. 1 consists of six symbols since the given application has six different GUI operations.

D. FSM transitions construction

Given the set of states and set of alphabet symbols, the construction of the transitions of the application-level state machine is based on four rules described in the following sections.

Domain definition rule

The domain definition rule requires that the content of each element that belongs to the domain of a given operation has to be defined before the operation is performed. Domain definition rule asserts that performing an operation does not make sense if one or more input elements for that operation are undefined. Expression (5) describes the domain definition rule:

$$R_{D_{DEF}}(s,\ op):\ \forall d \in D_{op}\ |\ s[d] = REF \vee s[d] = USED \tag{5}$$

where s denotes a particular state of the application-level FSM, op denotes an operation, D_{op} denotes the set of elements comprising the domain of operation op, and $s[d]$ denotes the value of the state s for a GUI element d. According to expression (5), to satisfy the domain definition rule, each content-sensitive GUI element belonging to the domain of an operation has to be either in *REF* or *USED* state in order to perform an operation.

Domain refreshness rule

The domain refreshness rule requires that the content of at least one element that belongs to the domain of a given operation must be refreshed before the operation is performed. Domain refreshness rule asserts that there is no sense in performing the same operation multiple times if at least one input element has not been redefined in the meantime. Expression (6) describes the domain refreshness rule:

$$R_{D_{REF}}(s,\ op):\ \exists d \in D_{op}\ |\ s[d] = REF \tag{6}$$

According to expression (6), to satisfy the domain refreshness rule, at least one content-sensitive GUI element belonging to the domain of an operation has to be in *REF* state in order to perform that operation.

Co-domain non-definition rule

Co-domain non-definition rule requires that the content of each element that belongs to the co-domain of a given operation has to be undefined before the operation is performed. Co-domain non-definition rule enables a cold-start of GUI applications when the content of output elements is still undefined. Expression (7) formally describes the co-domain non-definition rule.

$$R_{C_{NDEF}}(s,\ op):\ \forall d \in C_{op}\ |\ s[d] = NDEF \tag{7}$$

where C_{op} denotes the set of elements comprising the co-domain of operation op. According to expression (7), to satisfy the co-domain non-definition rule, each content-sensitive GUI element that belongs to the co-domain of the given operation has to be in *NDEF* state before the operation is performed.

Co-domain usage rule

The co-domain usage rule requires that the content of at least one element belonging to the co-domain of a given operation must be used before the operation is performed. Co-domain usage rule prevents that effects of previously executed operations are overwritten before their usage in cases when co-domains of two operations overlap fully or partially. Expression (8) describes the co-domain usage rule.

$$R_{C_{USED}}(s,\ op):\ \exists d \in C_{op}\ |\ s[d] = USED \tag{8}$$

According to expression (8), to satisfy the co-domain usage rule, at least one content-sensitive GUI element belonging to the co-domain of an operation has to be in the *USED* state.

[Downloaded from www.aece.ro on Thursday, March 28, 2024 at 15:48:50 (UTC) by 44.200.210.43. Redistribution subject to AECE license or copyright.]

Composite rule

The composite rule defines how four basic FSM transitions construction rules are applied to construct the transitions of the application-level finite state machine. To perform a given GUI operation, we require that the content of all GUI elements belonging to the operation domain is defined (domain definition rule) and that the content of at least one GUI element is refreshed (domain refreshness rule). Furthermore, we require that the content of all GUI elements belonging to the operation co-domain is either still undefined (co-domain non-definition rule) or, if the content of some elements are already defined, then the content of at least one such element has to be used (co-domain usage rule). The composite rule is described using expression (9):

$$R_{COMP}(s,\ op):\ R_{D_{DEF}}(s,\ op)\ \wedge\ R_{D_{REF}}(s,\ op)\ \wedge \left(R_{C_{NDEF}}(s,\ op)\ \vee\ R_{C_{USED}}(s,\ op)\right) \tag{9}$$

If the composite rule applies for a given FSM state and a given GUI operation, then that operation may be applied as a transition to transfer the GUI from the given state to the next state. Thus, we may construct a transition from given state to the next state, according to the basic FSM shown in Fig. 3. The algorithm for constructing the set of transitions for application-level FSM is given in Section 5.

E. FSM initial state construction

The initial state of the application-level FSM depends on the initial state of application's GUI. For example, if the application GUI consists of four content-sensitive GUI elements, each of which is empty at the beginning of application execution, then the initial FSM state is [*NDEF*, *NDEF*, *NDEF*, *NDEF*]. On the other side, if first two GUI elements have predefined content, while the rest do not, then the initial FSM state is [*REF*, *REF*, *NDEF*, *NDEF*].

V. Algorithm for Application-Level FSM Construction

The algorithm for constructing an application-level FSM is presented in Fig. 5. The first input for the algorithm is a data structure **model** containing the model of the application GUI as defined with expressions (1) and (2), and exemplified in Fig. 1 and Fig. 2. The data structure contains a list of GUI elements and a list of GUI operations, where the domain and co-domain of each operation are represented as two hash tables. The second input for the algorithm is a data structure **initialGUIState** containing the initial state of the application as a list of GUI elements which are defined at the start of the application. Both input data structures for the algorithm are prepared by developers of the GUI application. The algorithm returns the sets **states** and **transitions** which contain the states and transitions of the constructed FSM.

The algorithm computes the output through three phases. First, the set of content-sensitive elements **dimensions** is computed from the **model** as described in expression (3) (lines 04 through 06). Second, the initial state of the FSM **initialFSMstate** is computed as described in Section 4.5 (lines 08 through 13). Third, the **states** and **transitions** sets are iteratively computed by checking the composite rule defined in expression (9) for each operation, starting from the initial state of the FSM (lines 15 through 28). If the composite rule does apply for a given state and given operation, the algorithm constructs a target state and makes a transition from current state to target state. The transition is labeled with the given GUI operation.

```
01 FSM_construct(model, initialGUIState):
02   define states, transitions, dimensions, reachableStates as empty sets
03
04   for each GUI element e in model:
05     if CE(e, model) does apply:
06       add e to dimensions
07
08   define state initialFSMState as new array[CE.length]
09   for each dimension d in dimensions:
10     if d ∈ initialGUIState:
11       set initialFSMState[d] to "REF"
12     else:
13       set initialFSMState[d] to "NDEF"
14
15   add initialFSMState to reachableStates
16   while reachableStates is not empty:
17     define s1 as any state removed from reachableStates
18     for each GUI operation op in model:
10       if RCOMP(s1, op) does apply:
20         define state s2 as a copy of s1
21         for each dimension d in dimensions:
22           if d ∈ Dop:
23             set s2[d] to "USED"
24           else if d ∈ Cop:
25             set s2[d] to "REF"
26         add transition from s1 for op to s2 to transitions
27         add s2 to reachableStates
28     add s1 to states
29
```

Figure 5. Application-level FSM construction algorithm

[Downloaded from www.aece.ro on Thursday, March 28, 2024 at 15:48:50 (UTC) by 44.200.210.43. Redistribution subject to AECE license or copyright.]

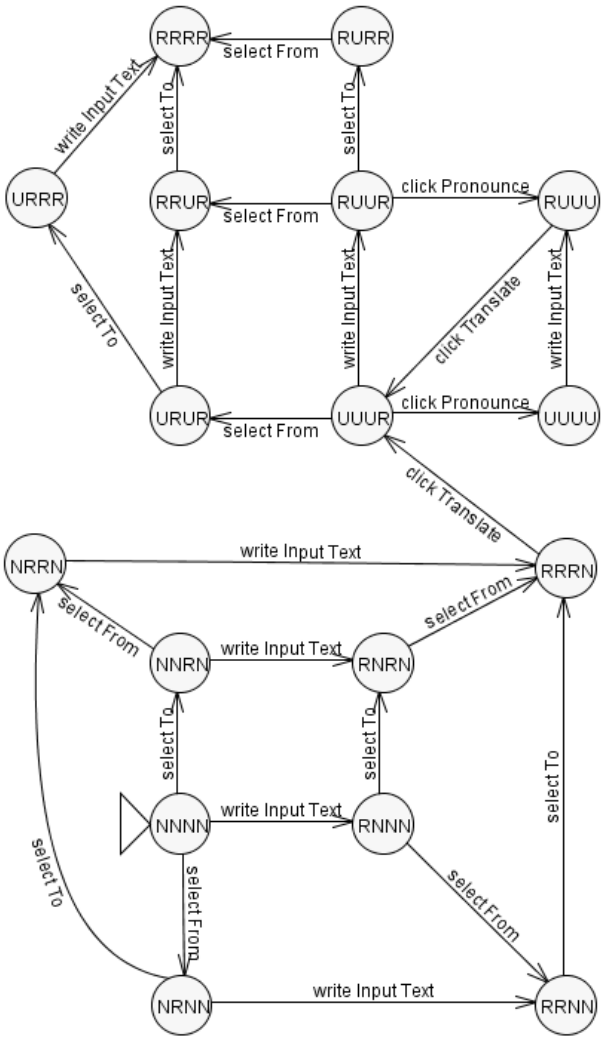


Figure 6. An excerpt of FSM describing the GUI of web application shown in Fig. 1

An excerpt of a complete FSM for application shown in Fig. 1 is shown in Fig. 6. For the sake of readability, we shown the first 17 states only and use *N* for *NDEF*, *R* for *REF*, and *U* for *USED* when labeling FSM states.

VI. Model Efficiency

In this section, we discuss the efficiency of the presented model and algorithm. The algorithm described in Section 5 has an upper bound complexity given with expression (10):

$$O\left(N_E * N_O + N_E + 3^{N_E} * N_O * N_E^2\right) \quad (10)$$

where N_E is the number of GUI elements of the application, while N_O is the number of GUI operations.

The first augend (N_E*N_O) represents the complexity of computing the set of content-sensitive elements; all GUI elements must be checked by examining the domain and co-domain of each operation. The second augend (N_E) represents the complexity of constructing the initial FSM state. The third augend ($3^{N_E}*N_O*N_E^2$) represents the complexity of constructing the states and transitions of the FSM. As defined by expression (4), the maximum number of states is $3^{|CE|}$, which in the worst case is 3^{N_E}. For each state, the composite rule must be checked for each operation, which has an upper bound complexity of $O(N_O*N_E)$. If the composite rule does apply, a new FSM state is constructed by defining each dimension according to the operations' domain and co-domain, which has an upper-bound complexity of $O(N_E)$. Expression (10) may be approximated with an upper-bound complexity of $O(3^n*n^3)$ where n represents the sum of the numbers of GUI elements N_E and operations N_O.

Although the algorithm for construction of the application-level FSM has super-exponential upper-bound complexity, the average complexity and size of the constructed FSM will be significantly smaller.

First, although the FSM constructed for an application with N content-sensitive GUI elements may have a maximum of 3^N states, many of these states will be unreachable. The algorithm presented in Fig. 5 acknowledges this by incrementally constructing only reachable states, while unreachable states are not constructed. For example, the FSM that describes the application shown in Fig. 1, which has four content-sensitive elements, is reduced from maximum of 81 states to 24 states reachable from initial [*NDEF*, *NDEF*, *NDEF*, *NDEF*] state.

Second, although each state of the FSM has different application-level semantics, many states are identical with respect to FSM operation. In the proposed model, GUI operations do not differentiate *NDEF* from *USED* state of a particular element since both enable the same set of operations to execute. In essence, the *NDEF* and *USED* states, although different in semantics, are equivalent from automata theory point of view [10]. Therefore, two states of the application-level FSM which for each dimension have either the same values or different values *NDEF* and *USED*, are equivalent and may be reduced to a single FSM state. This property may be acknowledged either by modifying the algorithm to construct only non-equivalent states or by merging equivalent states a-posteriori.

Lastly, since the model assumes that the domains and co-domains of GUI operations are non-changeable during application execution, which is true for most GUI applications, the constructed FSM is deterministic.

In conclusion, these two properties, reduced number of states and FSM determinism, enable both efficient construction of application-level FSMs and their efficient usage.

VII. Model Usage

In order to demonstrate the practical use of the model, we describe two possible usage scenarios within the *Geppeto* framework [4]. *Geppeto* is a consumer-oriented framework for programming application-level workflows over widgets. Programming in *Geppeto* is achieved using a *programming-by-demonstration* technique, while programs consist of sequences of GUI operations over widget GUI elements. The presented model may be similarly applied to other web applications enabling GUI automation [2], [3], [6].

First, if each widget was modeled with an application-level FSM, the *Geppeto* framework could use the FSM to guide users through the programming process in order to reach a certain goal. The specification of a goal is done by defining a goal state, based on which a path towards that state consisting of GUI operations may be computed. For

[Downloaded from www.aece.ro on Thursday, March 28, 2024 at 15:48:50 (UTC) by 44.200.210.43. Redistribution subject to AECE license or copyright.]

example, if the translation widget presented in Fig. 1 is being used to program another application, the framework would visually guide the user to specify a command for entering the input (*write Input Text*), then a command to specify the source and target languages (*select From*, *select To*), after which a command for clicking the translate button (*click Translate*), and lastly a command to consume the output translation (*read Output Text*).

Second, if each widget was modeled with an application-level FSM, the *Geppeto* framework could verify the correctness of consumer-defined programs before their execution. For example, if the translation widget presented in Fig. 1 is used in the program and if the consumer specified multiple operations for entering the input (*write Input Text*) before consuming the output (*read Output Text*), the framework may alert the user that the output was not consumed. Similarly, if the consumer specified commands for entering input text (*write Input Text*) and specifying source and target languages (*select From*, *select To*) but didn't specify commands for clicking the translate button (*click Translate*) and consuming the results (*read Output Text*), the framework may alert the user that the program flow was not completed.

VIII. Related Work

As we indicated in Section 1, development of the models of GUI applications has been a research focus of GUI testing [7], [8]. In GUI testing, the object of verification is a GUI application under development, while the verification is achieved using series of correct test. In contrast, our work focuses on cases where GUI applications are known to be correct and GUI-level programs require verification.

Most GUI testing techniques are based on model-based testing [11] in which the GUI is modeled as a finite set of states and a set of transitions between those states [12]. Based on the model, a large number of tests are automatically generated [13] that check the model for various properties, like reachability and ordering of application-specific goal states.

There are two key challenges in developing effective GUI testing techniques: state explosion of the model and model extraction. First, the number of states in a finite-state model may be unmanageably large even for simple applications due to a large number of possible states for the content of each GUI element. As a result, the number of tests required to verify the application may become unacceptably large due to the high execution time of running all tests. This state explosion problem is usually approached by aggregating states representing specific content into abstract states which represent multiple different contents of a single element [14], [15]. Nevertheless, the aggregation must be expressive enough to model key system properties which are being verified.

Creating an effective model has also been a goal of our research, even though we do not generate tests but use the application-level finite state machine for verification of GUI automation programs. Therefore, we introduced a novel aggregation scheme which combines all states of a single GUI element into three abstract states representing non-defined, refreshed, and used content. As we have shown, this scheme is expressive enough to formalize correct usage of GUI interfaces.

Another approach used in GUI testing techniques for constructing tests in cases of large state sets is plan generation. With plan generation [16], some states are defined as goal states and tests are constructed as sequences of operations leading from the initial to the goal state. As we described in the previous section, a form of plan generation may be applied to our model in order to guide end-users in the construction of GUI automation programs.

The second challenge in GUI testing techniques is extraction of a model from existing applications which do not have one. For such cases, several reverse engineering approaches have been proposed. In [17], a *GUI ripping* is introduced as a process in which the software's GUI is automatically "traversed" by opening all its windows and extracting all their widgets, properties, and values. In contrast, the reverse engineering approach presented in [15] is based on parsing the source code of Java applications.

The challenge of model extraction is important for our work also. In order to construct an application-level FSM, sets of GUI elements used as inputs and outputs must be known for every operation. Since for most applications these sets are not known, a system for automatically determining these sets for an arbitrary GUI application is required and both GUI ripping and source code parsing approaches may be used.

IX. Conclusion

In this paper, we introduce a formal model for describing data dependencies between controls and operations of GUI applications. The proposed model is based on the formalism of finite state machines, where states describe the possible state of GUI controls, and transitions denote GUI operations. The set of possible states is obtained by a novel state aggregation scheme, which combines all states of a single GUI element into three abstract states representing non-defined, refreshed, and used content. We show how this aggregation scheme is still expressive enough to describe correct GUI usage. Furthermore, we give a set of simple rules and an algorithm for automatic construction of finite state machines for arbitrary GUI applications. Lastly, we show how the model may be applied for reasoning about GUI dynamics which is often used in applications for web automation, testing, customization, and personalization.

The presented aggregation scheme and algorithm enable both efficient construction and efficient usage of the state machines. First, although the algorithm constructs the FSM with super-exponential upper bound complexity, we show that on average this complexity is significantly reduced due to many unreachable and equivalent states. Consequently, the size of the constructed FSM is reduced with respect to the number of states and transitions. Second, the aggregation scheme produces a deterministic finite state machine which enables efficient usage.

Our results present several beneficial directions for future work. First, a system for automatic extraction of GUI operation domains and co-domains is required in order to apply the model to existing applications. Although the presented model is not tied to a specific programming language or development framework, our interests are focused on web applications. Second, in order to evaluate

[Downloaded from www.aece.ro on Thursday, March 28, 2024 at 15:48:50 (UTC) by 44.200.210.43. Redistribution subject to AECE license or copyright.]

the performance of program verification and end-user satisfaction of program construction guidance, we plan to integrate the model into the *Geppeto* framework. Third, our model assumes that domains and co-domains of operations are non-changeable during application execution, which in some cases is not true. For example, if two text fields represent a username-password login form, the co-domain of login operation may vary depending on the correctness of the supplied user credentials. Therefore, the model needs to be extended in order to cover this non-determinism.

REFERENCES

[1] S. Staiger, "Static analysis of programs with graphical user interface", *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, Amsterdam, Netherlands, 2007, pp. 252-264. [Online]. Available: http://dx.doi.org/10.1109/CSMR.2007.44

[2] M. Bolin, M. Webber, P. Rha, T. Wilson, R. C. Miller, "Automation and customization of rendered web pages", *Proceedings of the 18th Annual ACM Symposium on User Interface Software and Technology*, Seattle, WA, USA, 2005, pp. 163-172. [Online]. Available: http://dx.doi.org/10.1145/1095034.1095062

[3] J. Wong, J. Hong, "Making mashups with marmite: towards end-user programming for the web", *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'06)*, Montreal, Quebec, Canada, 2006, pp. 1541-1546. [Online]. Available: http://dx.doi.org/10.1145/1240624.1240842

[4] S. Srbljic, D. Skvorc, D. Skrobo, "Widget-oriented consumer programming", *Automatika: Journal for Control, Measurement, Electronics, Computing and Communications*, Vol. 50, No. 3-4, December 2009, pp. 252-264. [Online]. Available: http://hrcak.srce.hr/index.php?show=clanak&id_clanak_jezik=73263

[5] A. Holmes, M. Kellogg, "Automating functional tests using Selenium", *Proceedings of the Conference on AGILE 2006*, Minneapolis, Minnesota, USA, 2006, pp. 270-275. [Online]. Available: http://dx.doi.org/10.1109/AGILE.2006.19

[6] T. Yeh, T. H. Chang, R. C. Miller, "Sikuli: using GUI screenshots for search and automation", *Proceedings of the 22nd Annual ACM Symposium on User Interface Software and Technology*, Victoria, BC, Canada, 2009, pp. 183-192. [Online]. Available: http://dx.doi.org/10.1145/1622176.1622213

[7] A. M. Memon, "Advances in GUI testing", *Advances in Computers*, No. 58, Academic Press, August 2003, pp. 149-202. [Online]. Available: http://dx.doi.org/10.1016/S0065-2458(03)58004-4

[8] J. C. Silva, J. Saraiva, J. C. Campos, "A generic library for GUI reasoning and testing", *Proceedings of the 2009 ACM Symposium on Applied Computing*, Honolulu, Hawaii, 2009, pp. 121-128. [Online]. Available: http://dx.doi.org/10.1145/1529282.1529307

[9] A. Gill, *Introduction to the theory of finite state machines*, McGraw-Hill, New York, 1962.

[10] J. E. Hopcroft, "An *nlogn* algorithm for minimizing the states in a finite automaton", *The Theory of Machines and Computations*, Academic Press, New York, 1971. [Online]. Available: ftp://db.stanford.edu/pub/cstr/reports/cs/tr/71/190/CS-TR-71-190.pdf

[11] V. Chinnapongse, I. Lee, O. Sokolsky, S. Wang, P. L. Jones, "Model-based testing of GUI-driven applications", *Proceedings of the 7th IFIP WG 10.2 International Workshop on Software Technologies for Embedded and Ubiquitous Systems*, Newport Beach, CA, USA, 2009, pp. 203-214. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-10265-3_19

[12] F. Belli, "Finite state testing and analysis of graphical user interfaces", *Proceedings of the 12th International Symposium on Software Reliability Engineering*, Hong Kong, China, November 2001, pp. 34-43. [Online]. Available: http://dx.doi.org/10.1109/ISSRE.2001.989456

[13] V. Santiago, N. L. Vijaykumar, D. Guimaraes, A. S. Amaral, E. Ferreira, "An environment for automated test case generation from statechart-based and finite state machine-based behavioral models", *IEEE International Conference on Software Testing Verification and Validation Workshop*, 2008, pp. 63 72. [Online]. Available: http://dx.doi.org/10.1109/ICSTW.2008.7

[14] M. B. Dwyer, V. Carr, L. Hines, "Model checking graphical user interfaces using abstractions", *ACM SIGSOFT Software Engineering Notes*, Vol. 22, No. 6, New York, NY, USA, November 1997, pp. 244-261. [Online]. Available: http://dx.doi.org/10.1145/267896.267914

[15] A. M. Memon, "An event-flow model of GUI-based applications for testing", *Software Testing, Verification & Reliability*, Vol. 17, No. 3, John Wiley and Sons, September 2007, pp. 137-157. [Online]. Available: http://dx.doi.org/10.1002/stvr.v17:3

[16] A. M. Memon, M. E. Pollack, M. Lou Soffa, "Plan generation for GUI testing", *Proceedings of The Fifth International Conference on Artificial Intelligence Planning and Scheduling*, April 2000, pp. 226-235. [Online]. Available: http://www.aaai.org/Papers/AIPS/2000/AIPS00-024.pdf

[17] A. M. Memon, I. Banerjee, A. Nagarajan, "GUI ripping: reverse engineering of graphical user interfaces for testing", *10th Working Conference on Reverse Engineering*, 2003, pp. 260-269. [Online]. Available: http://dx.doi.org/10.1109/WCRE.2003.1287256