Proceedings of the International MultiConference of Engineers and Computer Scientists 2011 Vol I,
IMECS 2011, March 16 - 18, 2011, Hong Kong

Cartographer: Architecture of a Distributed System for Automated GUI Map Generation

Paweł Brach, Jacek Chrząszcz, Janusz Jabłonowski, Jakub Świątły

***Abstract*—Vast majority of contemporary computer programs come with a complex Graphical User Interface. To facilitate automatic GUI testing, we propose Cartographer, a system for automatic generation of GUI maps. In comparison to existing solutions Cartographer has many important improvements: it is easily adaptable to particular application and technology and it is highly scalable, as it is based on a distributed system of independent Workers.**

***Index Terms*—GUI, GUI map, automatic program testing, automatic GUI map generator.**

I. Introduction

Contemporary computer programs come with complex Graphical User Interfaces (GUIs) [1]. It makes them comfortable to use but at the same time complicated to analyze (for example to test them [2]). There are several reasons why GUIs are not so easy to analyze:

- They use mouse clicks and other kinds of users gestures (like drag and drop) for communication. That makes the communication language very complex. Even recording user actions is not so trivial—programs may recognize as actions even moving the mouse over some active parts of a window.
- The set of possible interactions between a GUI program and a user is extremely rich. Usually the user issues many commands within one program invocation, it may be as well thousands like with work within a spreadsheet or a word processor.
- There is no one single path through the program, usually at each window the user can make many actions in any order.

But GUIs are popular and will remain for long (until new era of user interfaces, maybe speech-based, will dethrone them). Therefore tools aimed at program analysis have to deal with them.

In this paper we described a tool, called Cartographer, capable of performing such analysis. By simulating user actions on the analyzed application it produces a GUI map—a graph of connections between windows. Having such a map is very useful, first of all, paths of the map can be used to design and automatically generate test suites. The tests can be enriched with checkpoints, verifying e.g. if specific GUI elements are present in the window. Design of test suites according to the map helps us achieve the largest possible coverage of the GUI. A similar approach to using a GUI map was presented in [3] and [4].

Map analysis can give many valuable data on the overall quality of the program GUI. The number of nodes and connections gives a metrics of the program complexity. The length of the longest path describes how easy (or difficult) it would be for the user to access all program functions. The average nodes degree gives clues about the complexity of the program from the user view point. Discovering parts which are closely or loosely connected may be used to improve GUI itself, as their existence suggests nontrivial navigation through the interface.

Not only having a program map is useful — also the process of generating it can bring some useful functionality. For example during map generation the program may execute some actions in each window, like storing its image or texts presented to the user to a file. Such collection of all windows images is a valuable tool for somebody, who wants to evaluate the GUI, for example to check if all windows have proper view, clear layout and no linguistic errors which is specially important for program localization teams.

There are several (but not so many) tools already developed before. One of them is developed within the GUITAR [5], [6] project. Its principal author, Atif Memon, one of the most active researcher in field of automatic GUI testing, came up with the idea of the event-flow model. His approach decomposes GUI window into set of events with a relation that specifies pairs of events that can be fired one after another. Having such a graph allows easy automatic generation of test cases. As a part of GUITAR project an attempt was made to create a reverse engineering tool to aid creation of the model. However Memon's concept, although elegant, does not suit our purpose of accumulating precise knowledge of visited GUI windows and developing an effective way to navigate between them.

Our solution, in contrast to that of Memon, does not use reverse engineering and is technology independent. The plug-in architecture of the presented system allows easy customization for various GUI technologies. The overall performance of Cartographer is achieved by distributing analysis subtasks among a pool of Workers.

II. Preliminaries

A map of a GUI is naturally a graph. In a coarse simplification, the nodes of the graph are windows and edges are user actions that lead from one window to another. But complex user interfaces require more fine-grained approach distinction. First of all a part of a user interface which is perceived by the user as a window, may have different sets of controls, depending of the context. For example, dialogs such as "Options" usually have many tabs and different set of controls on every one of them. On the other hand, the effect of a user action sometimes depends on the state of controls (checkboxes checked or unchecked, text entered in

The authors' affiliation is Institute of Informatics, University of Warsaw.
{p.brach, j.chrzaszcz, j.jablonowski}@mimuw.edu.pl,
j.swiatly@students.mimuw.edu.pl

the text areas etc.). Therefore, if the graph constructed by the cartographer is supposed to be complete, one needs a more fine-grained approach to the interface graph.

Another problem that one has to face is window identification. While this is intuitively obvious for a human being, strictly defining a window is not a trivial task. Indeed, almost all "distinctive" features such as title, size, the set of controls etc. can depend on the context and identification via window handle is only valid in one run of an application.

A. GUI graph definition.

We decided to conceptually split windows into *layers* and take them as nodes of our interface graph. The layers are identified by the set of active (visible and enabled) controls i.e. their kinds, sizes, labels, states, etc. — the set of characteristics that contribute to the control identification (as opposed to the ones that contribute to the control state) is defined for each kind of control. Layers are grouped together into *layer groups* that are identified by the set of visible controls. The distinction between layers and layer groups is depicted in Fig. 1.

The edges of interface graph are user actions that lead from one layer to another. User actions are identified as the identification of the control (within the layer) plus the particular action (left or right click, keyboard input etc.). Note that this includes edges consisting in e.g. making a button enabled by checking a checkbox or entering some text in a textarea.

Having defined layers and layer groups, we can define a *window* to be the set of layers such that it is possible to go from one layer to the other without changing the window handle. The UI map visualization should take into consideration the fact that the layers are part of the same window, but it should be possible to see particular layers in a more detailed view.

B. Graph representation.

The nodes and edges are internally represented using hashes of controls characteristics used to calculate a layer or a layer groups identification. Note the choice of characteristics of controls used to calculate a hash of a given layer or layer group is very important. If too few characteristics are chosen, different layers are considered to be one which may result in an impossibility to repeat an edge which is actually available only in one of the two unified layers.

On the other hand, if too many characteristics are given, e.g. including the status (checked/unchecked) of checkboxes the number of layers becomes exponentially large and the whole graph becomes impossible to explore.

C. Power User.

Even though the system is supposed to automatically build a GUI map, we are aware that certain parts of the interface may be extremely difficult to explore or even discover automatically. These includes for example parts of the interface accessible when a correct username and password is entered etc. Still it is important to somehow include these hard explorable parts in the final map in order to get the full advantages of a GUI map. Therefore we

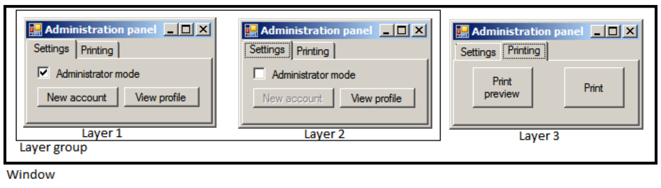

Figure 1. Windows and layers

decided to define a role of *Power User* to help the automatic algorithm discover layers and edges that would otherwise be impossible to explore.

Thanks to the distributed and multipass nature of the algorithm it is possible to find the first approximation of the UI graph automatically, than a Power User can add "secret" edges and during the second pass, the system can follow the newly added edges and explore new parts of the GUI.

Another role of the Power User is to merge windows or layers that are said to be different by the system of hashes, yet are really the same. The lack of unification may result from the fact that too many control characteristics is taken into account when identifying layers. The internal representation of such manually unified layers consists in the table of identification of layer hashes.

Dually, sometimes a mere analysis of widow controls may result in a false identification of windows that are actually different in the sense of results of actions performed on the controls. Think of a "Confirmation" dialog where the "Confirm" button really has different actions depending on what made the dialog appear. Here also, the Power User can decide that a dialog opened by one action is a different node graph from the same dialog opened by another action. In our implementations, the Map Repository service (see Section III below) is responsible for managing and storing hashes which identify different elements of GUI map i.e. windows, layers, controls, actions, tasks, etc. When the Power User wants to merge two different GUI map elements, it can create a new unique key which represents a group of hashes. Since that time, Cartographer uses the new key to identify those elements.

Some issues related to Power User hints are solved by using scripts. A *script* is a sequence of user actions performed on selected controls. In particular, the Power User is able to define starting scripts which set the application in a certain state, ready for GUI testing. More than one initial script can be given: for example one script sets the application into advanced mode, a second one into basic mode. Two different Workers could be engaged to explore these two GUI versions.

Other kinds of scripts may involve selecting a particular file from an "Open File" dialog or entering a valid username and password in a login window.

III. System Architecture

A. Background.

Here we provide a conceptual vision of the architecture of the distributed system for automated GUI map generation (Cartographer). The designed software architecture consists of modeling concepts for each self-contained module and its role in the whole system. In order to propose our design of the system, we considered existing system architectures (not only distributed systems) which satisfies functional and non-functional requirements. Shortly, we could describe our system as an automated GUI map generator, which take as its input binary application i.e. executable file and returns a finite graph as a kind of GUI map representation.

We made basic assumptions for Cartographer and attempted to predict future outcomes with a view to possible application of this kind of system. We realize that the problem is too complex and our solution is not going to cover 100% of the product. We have to be as accurate as possible and recognize a fundamental part of the GUI. The complexity of the problem, forces Cartographer to be “smart”. The brute force solutions are unacceptable which is result of the next assumption which says that the system must be fast and accurate.

The main function of such systems is not only to displace group of 10, 100 or even more people who can, just by clicking, manually create GUI map. This solution should also recognize not trivial connections between different parts of the GUI, difficult to be identified by human. The power of the system will be shown not on the one pass analysis, but in the later passes on the further versions of the product being analyzed. During the first pass, essentially big amount of work needs to be done to recognize application’s behavior. For example, when program being analyzed requires entering a password, the Power User must be consulted. In the further passes on new versions, Cartographer will supply the password by itself.

B. Design Research.

Design research investigates the process of designing in all its many fields. In this research, we have defined non-functional criteria specified for this kind of problems. These contains performance requirements which take into account measurable aspects of the system that govern overall speed and responsiveness. The next criterion describes that system has to be scalable. System has to manage very complicated software graphic interfaces, so we should take into considerations distributing of the product analysis.

As far as our findings are concerned, we’ve created Cartographer’s prototype which made us realize that we have to build distributed system. Review of existing models for parallel computation is provided in paper published in ACM journal [7]. The initial version of our solution was not parallel. One experiment has been fundamental for further evolution of development. We have tried to analyze probably one of the simplest Windows application – Notepad. It turned out to be too difficult for our single thread prototype where one Worker did almost whole job. Process of GUI map creation has not been completed in the expected time. After 24 hours, the returned GUI map covered less than 70% of the entire map (obtained after 40 hours of Worker’s work) in the sense of number of discovered states (nodes) and connections (edges). The resulted GUI map was really huge - there were 317 nodes and 1187 edges! The reason for that was windows like Font window, where there are hundreds of actions to perform which potentially are important for GUI map generation. As GUI maps are complex, we propose a distributed system to explore various fragments of GUI in parallel.

C. Detailed Description.

In this part of our paper we would like to give a high-level view of the architecture that we have proposed and implemented. We have decided to adapt one of the most popular technology founded by Microsoft. Windows Communication Foundation (WCF) [8] is a part of .NET Framework [9] that provides a unified programming model for rapidly building service-oriented applications. The WCF architecture is based on the design principles of service-oriented architecture (SOA) [10], [11] and gives us convenient and efficient level of abstraction in the development. The design of our project is closely related to the idea of cloud computing.

A WCF Service is a program that exposes a collection of endpoints which enable clients to communicate with desired data provider. A Client is a program that exchanges messages with one or more endpoints. A Client may also expose an endpoint to receive messages from a service in a duplex message exchange pattern.

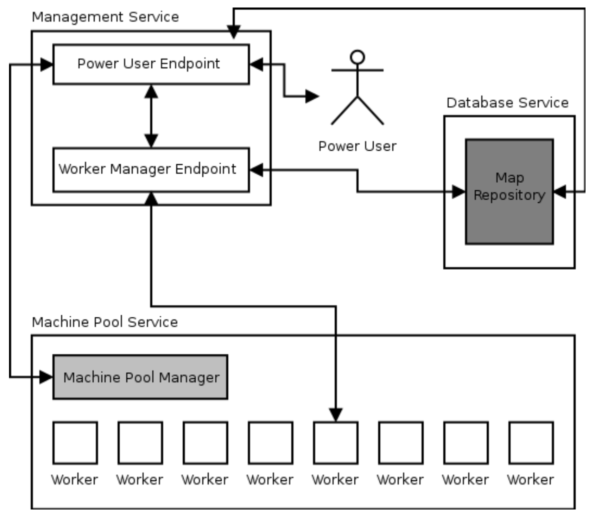

Figure 2. Cartographer System Architecture: Main components

In high-level view, in our system we single out 3 services: Management Service, Database Service and Machine Pool Service. Each of them exposes dedicated endpoints which are responsible for specific parts of functionality. We will describe each service with focus on Worker’s and Worker Manager’s parts. We intentionally disregard hardware issues because all services could be run on the same machine as well as on separated machines, what follows the cloud computing idea. We have tested our implementation on the network built on virtual machines which allows us to not care about hardware specific issues like tests reproducibility.

1) Management Service: The Management Service consists of Power User Endpoint and Worker Manager Endpoint. Power User Endpoint gives an interface for GUI part available for Power User. The main system control panel uses this endpoint, which redirect control to an appropriate element of the whole system. This is also the place where Power User is able to define new job, e.g. new application analysis, modifying system parameters, browsing through results, etc. The next component is Worker Manager Endpoint which is responsible for the fundamental part of the system logic. The Worker Manager is an entity which has the following tasks:

- managing pool of Workers in the sense of commissioning Workers to do some jobs (e.g. do Mission which

relies on analyzing the specified part of the considered application GUI),
- cooperate with Power User Endpoint,
- gathering the results from Workers (e.g. partial results, fragments of GUI map, information about incidents occurred during walking, etc.)
- merging GUI map from fragments derived from Workers,
- verification of doubtful parts of the map,
- transferring partial results to the Map Repository.

Surely we can say that Worker Manager is one of the most important elements of the Cartographer system. Worker Manager is some kind of the logic kernel responsible for final GUI map creation. One can say that Worker Manager Endpoint is a bottleneck of the Cartographer. We have taken steps to avoid that and have implemented Worker Manager in the multithreaded architecture. The WCF technology enables us to take advantage of multiple concurrency mode so if we provide appropriate hardware, there will be no issues with system capacity. Let us note that the most time consuming part of the system is Worker job (see Worker algorithm part).

2) Database Service: There will be no sense of doing analysis like that without giving ability to browse through the results. Database Service provides an interface for Map Repository where we store all GUI map fragments derived from Workers. We keep all information about specified stage of analysis, like pass id, environment settings, Worker id, incidents details, application description (revision number, version number), etc. Power User is able to compare outputs from different passes which may be useful for comparing two successive versions of the same application or just running next pass and analyzing new version of the given software based on results obtained in the previous stages.

3) Machine Pool Service: Virtual machines can help to build scalable, manageable and efficient Machine Pool infrastructure. The work proposed in this paper focuses on employing virtual machines for computing partial GUI maps. There are many advantages of using virtual machines to run Worker services on them. One of the advantages of having virtual Machine Pool is that one can instantly obtain the benefits of the enormous infrastructure without having to implement and administer it directly. It is very important to have possibility to configure every node in our machine network in the same way to make all experiments and calculations repetitive. We are able to run all Workers on machines with configuration based on the same disk image. What is more, we can add or remove machine dynamically while system is running, so Power User can decide whether to add or remove additional element from network.

Machine Pool is managed by Machine Pool Manager, which is available as a service cooperating with Power User Service. When new nodes are requested by Power User, Machine Pool Manager runs the configurator on each machine individually. After that application and Worker Service are run.

Worker Service gives an interface to a computation unit run on every node in the Cartographer system. The Worker fulfills an important role. Its function is to aide in the GUI map generation by executing Missions specified by Worker Manager. The common Mission for Worker is to analyze some new windows or layers of given application and look for new outgoing edges from the current state. The results are directly transferred to Worker Manager.

D. Application Environment.

We have investigated two possible approaches to the definition of nodes and edges of the GUI map. In the first, coarse-grained, nodes are windows of the application and edges correspond to transitions between them. In the second approach, fine-grained one, a node is a window together with values stored in its controls. The benefit of the first approach is the reduced number of nodes in the graph. Our research has shown that a hybrid solution will work best. We have introduced the notion of a window layer. As mentioned before a layer encompasses the window and some subset of values of its controls. The main idea is to observe that not all controls typically influence the GUI map. For example a value of a control displaying current time will typically not enable or disable some possible user navigations through the GUI. On the other side a checkbox is quite likely to enable or disable some functionalities of the program. Hence a layer is a window together with values of important controls.

For that reason, we have investigated which user interface elements may influence GUI map. We introduced items called “Configuration Controls” - those controls, whose value change alters (directly or indirectly) the GUI map. In other words these are exactly those controls, whose value change is necessary for traversing the entire GUI. But the identification of them is far from trivial. From practical point of view most difficult to find are those of them, whose change influences controls in other than their own windows. Unfortunately such controls do exist, for example in Options or Configuration windows of various programs.

The interaction of the user with such elements causes a change in application state. And this may influence which windows will be accessible in the future. These are potentially controls like menu, button, text box, radio button, check box, spin box, list, tab, scroll-bar and others. Sometimes we are not able to test all possible actions that could be done with controls. There are GUI controls which cause a need for manual entering data during GUI map generation because of too large number of possible values like text box, spin box, list (e.g. multi-select one). We have implemented a module called Task Generator to reduce situations when Power User action is needed. The Task Generator will be described in detail in the next parts. These all controls will be called interactive elements here.

The next group of controls, called GUI static elements have no connected actions to be performed. These are labels, groups of elements, graphics, etc. There are used by our algorithm only to support windows identification.

On the other hand, we have identified application environment elements that may have impact on GUI map. Many of them are related to a operating system, system registry and the level of utilization of accessible resources: virtual memory, disk storage, processes and threads, communication ports, network connections etc. There is also impact from other programs running concurrently including the system ones, configuration files, attached external devices. We have tried to minimize the influence of this part by using virtualization and providing the same configuration for all machines.

IV. GUI MAP GENERATION

The purpose of this section is to describe the general idea of the Cartographer algorithm. It does not delve deep into implementation details but gives a bird's eye view of the main part of the Cartographer system.

The algorithm generates a graph, with window layers as nodes and edges for transitions between them. Layer represents the part of the given window with visible and enabled (not grayed) controls on it. Different window views should be represented by different layers. Each edge has its own identifier. Additionally edges are labeled with actions, which made the transition between layers/windows. There is in general no possibility for making return transitions, hence the only way to visit a node more than once is in going forward. Therefore full exploration of a node will demand multiple openings of the corresponding window.

Each window is identified by its properties, like set of controls it contains, its size etc. For each visit in a node we store a structure, describing this visit, containing among others the identifier of the edge used to leave the layer. Each node holds a reference to the current visit structure.

A. *Setting new job.*

The GUI map generation process starts when Power User specifies new application for analysis through the Power User Endpoint. In the beginning every Worker is in the idle state and is waiting for messages from Worker Manager.

Worker gets from Worker Manager messages with the following types:

- NEW_MISSION - it means that Worker has to run specified application, and then execute initialization script (if one exists). Initialization script is responsible for preparing application for further analysis (e.g. configuring application, omitting splash screen, etc.),
- PLAY_SCRIPT - Worker has to do script (walk through the given path in the UI map) and start discovering GUI map,
- CONTINUE_FROM_CURRENT_PLACE - Worker should just start walking from current place and discovering GUI map,

In the beginning Worker Manager usually sends NEW_MISSION message to every Worker in the system with initialization script provided by Power User.

B. *Worker Manager algorithm.*

We can divide Worker Manager part into 2 main modules: Mission Generator and Map Validator.

1) Mission Generator.: This part of Worker Manager is responsible for generating new Missions for Workers. Every node in GUI map is tagged with number which quantities the percentage of completed tasks (it says how much we have analyzed the considered node). The Mission Generator is looking for the least analyzed nodes and commissions idle Workers to analyze these parts of GUI map. In the Worker Manager algorithm we optimize the Worker's job and commission them to analyze the node which is the closest to the current Worker place. From bird's eye view we can say that process of building GUI map is similar to breadth-first search (BFS) algorithm.

2) Map Validator.: This module is responsible for verification of doubtful parts of the GUI map. The GUI map is built from fragments derived from Workers. In the easiest case, when there are no conflicts between two parts of the map derived from different Workers, process of merging them is simple. Unfortunately, sometimes we have to deal with a situation when conflicts exist. The conflicted parts are marked with "doubtful" flag. Worker Manager has to verify these parts and resolve conflicts. Sometimes, when connections between two nodes are nondeterministic, it is not possible. In this case, we keep the appropriate flag.

Worker Manager is constantly communicating with Database Service and stores partial results into database.

C. *Worker algorithm.*

We assumed that there are instances of the following analyzers:

- TaskGenerator - generates tasks for the given layer,
- TaskSelector - selects task with the highest priority.

Task here represents compound task which contains set of simple tasks (e.g. right-click on the button). Window configuration is a kind of overlay on real window structure in operating system readable for Cartographer. Walk stack represents whole Worker knowledge about real windows stack (should represent stack of windows which are visible on the desktop in a given moment). When Worker gets new Mission from Worker Manager, it plays attached initialization script if one exists. After this step Worker is able to start discovering GUI map from current place. Below we present pseudocode of main Worker algorithm part (discovering).

D. *Task Generator and Task Selector logic.*

On its mission Worker has to decide which actions to take and in what order. Sometimes this decision is critical, because GUI design forces the user to perform a specific sequence of events to reveal a path to some window or layer. This problem is solved by an entity called Task Generator (TG). An input for TG is the tree of GUI elements, and its output is a collection of compound tasks with priorities assigned. Compound task is a sequence of simple tasks, which are primitive GUI actions such as setting a value in textbox or checking a checkbox. TG uses two main approaches.

First one is a naive generation of all available actions in the current state. All compound task generated are composed of just one simple task and have the same priority. Using this approach ensures that every active GUI element is used at least once. The shortcoming is that it cannot handle a situation when a specific sequence of actions is needed. This requirement is met with the second approach.

The second approach is based on generic, manually declared rules. Generally speaking, these rules describe patterns of GUI window structure and tell what actions to take when such a pattern is found. Rules are declared in an external xml file, which contains four main sections.

1) Selectors: Selector is essentially a filter that can be applied to GUI elements tree. Selector takes a stream of elements as its input and leaves only elements that satisfy defined constraints. Selector can make use of other streams of elements in its constraints, but it cannot filter them. Streams

```
if (WalkStack.Count > 0) {
  // Get foreground window configuration
  var conf = WalkStack.Pop();
  // Calculate active (foreground) layer
  var layer = conf.GetActiveLayer();
  // Generate compound tasks for Worker
  var tasks = TaskGenerator(layer);

  while (var task = TaskSelector(tasks).GetNext()) {
    // Do all steps of the current compound task
    while (task.DoStep()) {
      // Check whether window is still foreground
      if (conf.IsForeground()) {
        // Update window configuration (look for changes)
        conf.Update();

        // Check whether layer is still foreground
        if (layer != conf.GetActiveLayer()) {
          // Report to Worker Manager that there is
          // new edge between layers on
          // the same window
          Report();

          WalkStack.Push(conf);

          // End of walking, Worker has to wait
          // for new Mission from Worker Manager
          return;

        }//if(layer!=...

      }//if conf.IsForeground

      else { // new window

        // Get new window configuration
        var actConf = getActiveWindow();

        if (WalkStack.Contains(actConf)) {
          // Reports that task causes switching
          // windows on the walk stack
          Update(WalkStack);
          Report();
          return;

        }// if (WalkStack.Contains(actConf))

        else {
          // Reports that there is new edge between
          // different windows
          Update(WalkStack);
          WalkStack.Push(actConf);
          Report();
          return;
        }
      }// else
    }// while task.DoNext()
  }// while(...TaskSelector(tasks)...)
}// if (WalkStack.Count...
else {
  Error("Nothing to do");
  return;
}
```

Figure 3. Worker algorithm

are defined in the first section of selector definition. Stream can be defined as a whole elements tree (TG uses tree built-in iterator and doesn't make any assumptions about the order of nodes that it returns), or the elements tree filtered by some other selector (cycles are not allowed). Some constraints have already been implemented, but the architecture allows easy enhancements. Some of the implemented constraints are: checking a value of an element property, i.e. text or type, checking an existence of other element in defined neighborhood, checking a relation to other element in the tree (child, grandparent, etc.) checking if the element is a window. Constraints can be linked by conjunctions and disjunctions and modified by negation.

2) Forbidden elements: If for some reason we do not want the Worker to use specific elements from GUI we can put the selectors that describe them in the forbidden elements section. Entries in this section contain nothing more than selector names.

3) Tasks: In the tasks section one can define compound tasks. Every definition contains three sections. The first one is a definition of the element streams, which is similar to the one in selector declaration. Second section is optional and it allows specifying additional constraints that these streams must satisfy. The third section contains simple tasks definitions that are composed of action name, serialized arguments and stream from which the element should be taken. The semantics are as follows: TG divides selectors into two groups. The first group contains elements that are used in simple tasks definition, and second one those that are not. Then it removes forbidden elements (see Forbidden elements above) from the first group and checks if every stream (in both groups) enumerates to at least one element. If not, the definition is omitted. Otherwise it creates a cartesian product of the streams and generates compound task for each tuple that it contains. Note that the purpose of the streams from the second group is to force that some specific controls exist on the window, even though we don't want to use them.

4) Priorities: In this section priorities can be assigned to compound tasks created with definitions from the tasks section. Each entry contains task definition name and a number that represents priority. TG uses these two approaches simultaneously. Task Selector simply gets the task with the highest priority.

V. Conclusions

We have presented an architecture of a system to automatically build a map of the GUI, based on an semi-automatic interaction with the windows and GUI controls. Experiments conducted on an implemented prototype are very promising. They show the power of the chosen distributed architecture of the system. When a complete system is built it will become an invaluable help in quality assurance of GUI applications.

References

[1] A. Dix, J. E. Finlay, G. D. Abowd, and R. Beale, *Human-Computer Interaction*. Prentice Hall, 2003.

[2] P. Gerrard, "Testing GUI Applications," in *EuroSTAR '97*, November 1997. [Online]. Available: http://www.gerrardconsulting.com/GUI/TestGui.html

[3] A. M. Memon, "Automatically repairing event sequence-based GUI test suites for regression testing," in *ACM Transactions on Software Engineering and Methodology*, 2008.

[4] X. Yuan and A. M. Memon, "Using GUI run-time state as feedback to generate test cases," in *ICSE '07 Proceedings of the 29th international conference on Software Engineering*, 2007.

[5] A. M. Memon, "Using reverse engineering for automated usability evaluation of GUI-based applications," in *Software Engineering Models, Patterns and Architectures for HCI*. Springer-Verlag London Ltd, 2009.

[6] ——, "Gui ripping: Reverse engineering of graphical user interfaces for testing," in *In Proceedings of The 10th Working Conference on Reverse Engineering*, 2003, pp. 260–269.

[7] D. T. David B. Skillicorn, "Models and languages for parallel computation," in *ACM Computing Surveys (CSUR)*. ACM, 1998.

[8] J. Smith, *Inside Windows Communication Foundation*. Microsoft Press, 2007.

[9] A. Troelsen, "Exploring the .NET universe using curly brackets," in *Pro C# 2010 and the .NET 4 Platform*. Apress, 2010.

[10] H. Carr, "The PEPt service oriented architecture," in *2nd International Conference on Service Oriented Computing New York City*. ICSOC, 2004.

[11] P. R. Reed, *Reference architecture: The best of best practice*. Published on IBM Website, 2004.