

# Statische Analyse von graphischen Oberflächen

Stefan Staiger  
Institut für Softwaretechnologie  
Universität Stuttgart  
staiger@informatik.uni-stuttgart.de

**Zusammenfassung:** Dieser Beitrag beschreibt eine neue statische Analyse von Programmen mit graphischer Oberfläche (GUI). Die Analyse aus dem Bauhaus-Projekt extrahiert aus dem Quellcode die Widget-Hierarchien der GUI, GUI-Ereignisse und deren Reaktionen. Einige Anwendungen, beispielsweise eine Migration zu einem GUI-Builder, werden kurz vorgestellt. Testresultate untermauern die Tragfähigkeit unseres Ansatzes.

## 1 Einleitung

Viele Anwendungen besitzen eine graphische Oberfläche (GUI), die einen großen und wichtigen Teil ihrer Architektur ausmacht. Heute stehen Werkzeuge zur Verfügung (sogenannte GUI-Builder), um die Fenster der GUI, deren hierarchischen Aufbau aus Widgets wie Menüs, Schaltern und Boxen und einiges mehr auf eine komfortable Weise zu spezifizieren. Dennoch ist die Zahl der Programme, deren GUI handgeschrieben ist, nach wie vor groß. Bei diesen ist es ungleich schwieriger, die einzelnen GUI-Elemente im Quellcode zu lokalisieren und umgekehrt anhand des Quellcodes den Aufbau der GUI nachzuvollziehen.

Für diese Art von Programmen hilft unsere statische Analyse. Wir können aus dem Quellcode die Widget-Hierarchien extrahieren, aus denen die Fenster der GUI bestehen, und einige Informationen mehr zu den GUI-Elementen liefern. Wir können ermitteln, welche Ereignisse von den Widgets ausgelöst werden und welche Funktionen daraufhin als Reaktion durchlaufen werden. Ferner unterstützen wir eine Untersuchung der Programmarchitektur, da wir diejenigen Codeteile erkennen und vom Rest trennen, die zur GUI beitragen.

Unsere Analyse hilft bei verschiedenen Aufgaben. Einige Anwendungen, wie beispielsweise die Migration zu einem GUI-Builder, skizzieren wir in Abschnitt 2. Darauf folgen Abschnitte, die den Algorithmus hinter unserer statischen Analyse erläutern: Zunächst die Erkennung der für die GUI relevanten Codeteile in Abschnitt 3.1, dann das Aufspüren von Widgets, ihren Hierarchien sowie der Ereignisse und Reaktionen in Abschnitt 3.2. Anschließend schildern wir unsere Tests und diskutieren die Ergebnisse, die wir erhalten haben, in Abschnitt 4. Eine Literaturliste ermöglicht schließlich den Zugang zu weiterführenden Informationen und rundet den Artikel ab.

## 2 Anwendungen

Unsere Analyse unterstützt verschiedene Anwendungsszenarien. Wir wollen hier als Motivation und Abgrenzung der Problemstellung auf einige davon kurz eingehen.

*Redokumentation:* Die Informationen zu den Fenstern, deren Aufbau in Form von Widget-Hierarchien und das Wissen über die Reaktionen auf Ereignisse sind eine wichtige Dokumentation der GUI (sowie eines Teils des Verhaltens) der Anwendung.

*Migration zu einem GUI-Builder:* Die Migration von einer ausprogrammierten GUI zu einem GUI-Builder, bei dem die Oberfläche spezifiziert wird, erfordert unter anderem die Kenntnis der Fenster, der Widget-Hierarchien sowie der Ereignisse und deren Reaktionen. Unsere Analyse liefert diese Informationen; mit einem geeigneten Backend können damit die Fenster der GUI automatisch in der Sprache des GUI-Builders spezifiziert werden.

*Architektur-Erkennung:* Die GUI ist ein großer und wichtiger Teil der Anwendungen, die mittels einer graphischen Oberfläche dem Benutzer gegenüber treten. Daher ist die Separierung der GUI-Teile vom Rest ein wichtiger Beitrag zur Architektur-Erkennung. Weiterhin verleihen wir dem GUI-Teil der Architektur noch dadurch Struktur, dass wir die darin enthaltenen Fenster samt deren Aufbau erkennen. Auch über den Zusammenhang zwischen den Fenstern können wir Informationen liefern und somit auch Verbindungen und Abhängigkeiten zwischen den erkannten Architektur-Teilen erkennen.

*Programmverstehen:* Die Analyse zeigt zu jedem GUI-Ereignis (z.B. dem Auswählen eines Menüpunktes), welche Funktionen als Reaktion darauf angesprungen werden. Dies hilft dem Wartungsingenieur, die Benutzerinteraktion und die dadurch verursachten Abläufe im Programm zu verstehen, denn hierzu geben übliche Mittel wie der Aufrufgraph keine Information. Ferner können wir das Programmverstehen erleichtern durch eine Abbildung der bei einer Ausführung sichtbaren Programmteile auf Quellcode (und umgekehrt).

## 3 Die statische Analyse

Eine statische Analyse für graphische Oberflächen muss einige Probleme bewältigen. Da eine typische GUI-Anwendung umfangreich ist, muss die Analyse sehr effizient sein. Außerdem gebrauchen derartige Anwendungen gerne viele Zeiger, was nach einer statischen Zeigeranalyse verlangt. Zeigeranalysen sind jedoch schwierig und kostspielig. Schließlich sei als weiteres Problem noch speziell für die Kategorie der Anwendungen, die in C oder C++ geschrieben sind, die Vielfalt der Sprachdialekte erwähnt. So benutzt beispielsweise die bekannte GTK-Bibliothek, Grundlage für GNOME-Programme, den GNU-Dialekt der Sprache C. Ein Werkzeug zur Analyse solcher Anwendungen benötigt darum ein starkes Frontend, das mit diesen Dialekten umgehen kann.

Unsere neuartige Analyse, die ins Bauhaus-Projekt [RVP06] integriert wurde, kann mit diesen Herausforderungen umgehen. So sind wir in der Lage, verschiedene Dialekte von C und C++ zu analysieren und verfügen über ein breites Spektrum an Analysen: Verschiedene Zeigeranalysen [And94, Das00, Ste96], eine globale Kontrollfluss-Analyse und

eine interprozedurale SSA-Form (ähnlich zu der von Liao beschriebenen [Lia00]) bilden die Grundlage für die GUI-Analyse. Unsere GUI-Analyse selbst besteht aus zwei Phasen: Phase 1 erkennt die für die GUI relevanten Codeteile und Phase 2 sucht darin nach Widget-Hierarchien, Ereignissen und Reaktionen. Dabei lässt sich die Analyse mit einer Konfigurationsdatei auf besondere Fälle anpassen. Im Folgenden skizzieren wir kurz, wie diese Phasen arbeiten. Eine ausführlichere Beschreibung findet sich in einem aktuellen Konferenzbeitrag [Sta07]. Erweiterungen der Analyse sind noch in Arbeit, beispielsweise die Erkennung von Widget-Attributen und (im Rahmen des Entscheidbaren) deren Initialwerten, was die Unterstützung für eine Migration zu einem GUI-Builder sowie der Redokumentation weiter verbessert.

### 3.1 Phase 1: GUI-relevanten Code erkennen

Unsere Analyse bestimmt zunächst diejenigen Teile des Quellcodes, die zur Implementierung der GUI beitragen. Es sollen also für die GUI relevante Datentypen, Variablen, Funktionen und Dateien von den übrigen getrennt werden. Dazu nutzen wir die Tatsache, dass fast alle GUI-Anwendungen auf einer Bibliothek aufbauen, welche die Widgets zur Verfügung stellt (z.B. GTK oder Qt). Die Bibliotheksanteile lassen sich leicht daran erkennen, dass sie aus den Include-Verzeichnissen der Bibliothek stammen müssen. Alternativ können wir damit starten, die relevanten Basistypen aus der Bibliothek zu markieren (z.B. `QWidget` oder `_GtkWidget`). Dazu ist lediglich minimaler Konfigurationsaufwand durch den Anwender nötig, wobei sich die Konfiguration zudem auf weitere Programme übertragen lässt, die dieselbe GUI-Bibliothek nutzen.

Auf diesen Grundelementen aufbauend erkennen wir auch die anwendungsspezifischen GUI-Teile, denn diese benutzen (zumindest transitiv) die Elemente der Bibliothek. Wird z.B. eine Klasse von einer Klasse aus der Bibliothek abgeleitet, so ist die Chance groß, dass diese abgeleitete Klasse ebenfalls mit der GUI in Verbindung steht und vielleicht gar ein spezielles Widget definiert. Allgemeiner gesprochen betrachten wir den gerichteten Graphen, den die Datentypen mit ihren Beziehungen aufspannen, und klassifizieren einen Typ als für die GUI relevant, wenn die von ihm ausgehenden Kanten im Graphen auf ebensolche Typen verweisen. Hierbei kann man die Relevanz des Typs noch gewichten, indem der Anteil und die Art der Kanten zu GUI-relevanten Typen sowie deren jeweiliges Gewicht in Betracht gezogen wird. Für die Traversierung der Typen benutzen wir im Wesentlichen Tarjans Algorithmus zum Auffinden maximaler starker Zusammenhangskomponenten [Tar72] (eine Postorder-Traversierung mit geeigneter Behandlung von Zyklen).

Hat man Datentypen auf diese Weise klassifiziert, lassen sich darauf aufbauend auch Aussagen über den Beitrag von Variablen, Funktionen und Dateien zur GUI machen. So kann beispielsweise dem Wartungsingenieur beim ersten Kontakt mit dem System schnell gezeigt werden, welche Codeteile er in Betracht ziehen muss, wenn es um die Wartung der GUI oder der Anwendungslogik geht. Die Ergebnisse werden aber auch in Phase 2 der Analyse eingesetzt: Dort ist für uns interessant, welche Datentypen für Widgets in Frage kommen, denn dies ist ein nützliches Kriterium bei der Untersuchung, ob eine Funktion z.B. ein Widget erzeugt.

### 3.2 Phase 2: Extraktion von Widget-Informationen

Unsere Analyse entnimmt dem Quellcode folgende Informationen:

- Welche Widgets besitzt die Oberfläche und wo werden sie erzeugt?
- Welche Fenster gibt es und aus welchen Widget-Hierarchien bestehen diese?
- Welche Ereignisse können von den Widgets ausgelöst werden und welche Funktionen reagieren darauf?
- Wo werden Widgets ineinander eingebettet und wo werden Verbindungen zwischen Ereignissen und Reaktionen aufgebaut?

Als Grundlage werden die Ergebnisse der im vorigen Abschnitt beschriebenen GUI-Erkennung sowie Datenfluss-Informationen benutzt. Die Datenfluss-Informationen dienen dazu, Ausdrücke zu erkennen, die dasselbe Widget bezeichnen.

Die Analyse arbeitet nun in den folgenden Schritten: Zunächst werden Funktionsaufrufe gesucht, die relevant erscheinen. Eine gerufene Funktion ist dabei relevant, wenn sie ein Widget erzeugt, Widgets ineinander einbettet oder ein Ereignis mit einer Reaktion verbindet. Anschließend werden die Argumente (im Falle von Widget-Konstruktoren auch die Rückgabe) inspiziert und Ausdrücke für die jeweiligen Rollen identifiziert. Beispielsweise müssen bei einer Widget-Einbettung Ausdrücke für die Rolle des Vater-Widgets und des Kind-Widgets bestimmt werden. Mittels der Datenfluss-Informationen erkennen wir die Ausdrücke, die dasselbe Widget bezeichnen und können so Widget-Hierarchien aufbauen und Ereignisse zu Elementen dieser Hierarchien zuordnen.

Abbildung 1 zeigt den Algorithmus. Die Erkennung der GUI-Teile haben wir als Phase 1 skizziert. Die Hilfsfunktionen zur Erkennung von z.B. Widget-Konstruktoren unterstützen neben der Information, welche Typen für Widgets in Frage kommen, die Angabe von Namensmustern (als reguläre Ausdrücke) in der Konfigurationsdatei, die von den relevanten Funktionen erfüllt werden müssen. Haben wir einen relevanten Funktionsaufruf entdeckt, so können wir zunächst Ausdrücke für die beteiligten Widgets aus den beteiligten Argumenten ablesen.

Für die Bestimmung der Widgets, die mit diesen Ausdrücken bezeichnet werden, nutzen wir abschließend unsere SSA-Form wie folgt: Die Ausdrücke, mit denen beispielsweise bei einer Einbettung die beteiligten Widgets angegeben werden, entsprechen *Use*-Knoten in der SSA-Form. Zu diesen bestimmen wir (durch Verfolgung der Kanten der SSA-Form) die zugehörigen *Def*-Knoten, also Zuweisungen aller Art, welche den Wert gesetzt haben könnten, der beim *Use* benutzt wird. Jedes *Def*, das einem Aufruf an einen Widget-Konstruktor entspricht, wird schließlich als Widget angesehen. Ein Ausdruck kann somit an mehrere Widgets gebunden werden. Im Allgemeinen ergeben sich durch konservative Abschätzungen dieser Art gerichtete Graphen als Widget-Hierarchien, nicht zwingend Bäume.

```

procedure GUI_Analysis is
begin
  Load_Config_File;
  — GUI-Teile bestimmen (Phase 1)
  Find_GUI_Types;
  Find_GUI_Variables;
  Find_GUI_Functions;
  Find_GUI_Files;
  — Widget-Hierarchien samt Ereignissen extrahieren (Phase 2)
  forall  $c \in$  Function_Calls loop
    if  $\exists f \in c.Targets : Is\_Widget\_Constructor(f)$  then
      New_Widget( $c$ );
    end if;
    if  $\exists f \in c.Targets : Is\_Widget\_Embedder(f)$  then
      New_Embedding(Get_Parent( $c$ ), Get_Child( $c$ ));
    end if;
    if  $\exists f \in c.Targets : Is\_Event\_Connector(f)$  then
      New_Event_Connection
        (Get_Sender( $c$ ), Get_Event( $c$ ),
         Get_Receiver( $c$ ), Get_Handler( $c$ ));
    end if;
  end loop;
  — Mit Datenfluss-Information Ausdrücke für gleiche Objekte bestimmen
  forall  $w \in$  Widget_Expressions loop
    Bind_Expression( $w$ );
  end loop;
  — Ergebnisse ausgeben (Text, dot, Bauhaus-Tools, ...)
  Print_Results;
end GUI_Analysis;

```

Abbildung 1: Die GUI-Analyse (vereinfacht)

## 4 Tests und Ergebnisse

Wir haben unsere Analyse mit den in Tabelle 1 genannten Anwendungen getestet. Die beiden wichtigen GUI-Bibliotheken GTK (für GNOME) und Qt (für KDE) wurden hierbei unterstützt. Die Tabelle zeigt, dass wir selbst große und bekannte Programme wie `gimp` in kurzer Zeit untersuchen konnten. Die Laufzeit der Analyse hängt dabei unter anderem von der Dateigröße ab, die unsere Zwischendarstellung für das jeweilige Gesamtsystem benötigt; für C++ ist diese Darstellung zur Zeit recht umfangreich, weswegen die Qt-Programme eine längere Laufzeit benötigten.

Die bisherigen Testresultate sind ermutigend (vgl. Tabelle 2). Wir können bereits große Widget-Hierarchien und zugehörige Ereignisse und Reaktionen erkennen. Für das Programm `codebreaker` wurde beispielsweise der Aufbau des Hauptfensters sehr gut bestimmt. Ein kleineres Problem sind Widgets, die in Schleifen erzeugt werden, denn hier können wir nicht die korrekte Anzahl darstellen.

Für viele Widgets wird außerdem ein hilfreicher String gefunden, beispielsweise die Beschriftung von Buttons und Menü-Einträgen. Dieser hilft, die sichtbaren Elemente der GUI mit den Analyse-Resultaten und damit mit den richtigen Quellcode-Stellen in Verbindung zu setzen. Durchschnittlich etwa 85% aller Widget-Ausdrücke konnten an Widgets gebunden werden, was für eine geringe Zahl an false positives sowie eine gute Qualität unserer Basisanalysen spricht.

Programm	Sprache	Bibliothek	SLoc	Analyse-Zeit	User-Time	Dateigröße
codebreaker	C	GTK	1,217	0.11s	0.27s	0.7 MB
jlfractal	C++	Qt	3,060	7.49s	18.80s	88 MB
sgrviewer	C++	Qt	3,327	16.28s	38.30s	184 MB
qspacehulk	C++	Qt	18,886	39.957s	105.86s	588 MB
euler	C	GTK	24,103	0.62s	3.51s	32 MB
qtroadmap	C,C++	Qt	25,843	9.69s	26.79s	140 MB
bluefish	C	GTK	40,765	1.66s	3.85s	19 MB
gqview	C	GTK	52,998	2.03s	5.72s	32 MB
qtads	C++	Qt	82,016	20.01s	57.63s	371 MB
dia	C	GTK	124,825	2.94s	8.25s	52 MB
gimp	C	GTK	568,826	14.88s	48.72s	377 MB

Tabelle 1: Test-Programme, Laufzeit und Dateigröße

Programm	Widgets	Einbettungen	Ereignisse	Größte Hierarchie
codebreaker	88	71	20	39
jlfractal	67	71	26	26
sgrviewer	403	122	26	15
qspacehulk	920	347	55	41
euler	122	98	45	32
qtroadmap	113	51	8	8
bluefish	1428	1126	148	56
gqview	1261	829	252	76
qtads	361	274	36	103
dia	836	721	220	22
gimp	4000	2244	880	21

Tabelle 2: Erkannte Widgets, Ereignisse and Hierarchien

## 5 Verwandte Arbeiten

Überraschenderweise scheint die *statische* Analyse von graphischen Oberflächen in bisherigen Publikationen kaum behandelt worden zu sein. Dynamische Analysen dagegen wurden bereits eingesetzt. So setzen z.B. Chan et al. [CLM03] auf eine dynamische Analyse, um Benutzeraktionen in einer Datenbank zu protokollieren, welche anschließend dabei hilft, den Entwurf der Anwendung zu verstehen. Michail [Mic02] skizziert einen Ansatz, der mit einfachen lexikalischen Mitteln GUI-Nachrichten der Menüs von KDE-Applikationen erkennt. Unsere Analyse dagegen ist nicht auf diese Nachrichten und Programme beschränkt und basiert auf mächtigeren Analysen, die z.B. auch Zeiger und deren Ziele in Betracht ziehen. Weitere Arbeiten zum Reverse-Engineering von Benutzerschnittstellen wurden beispielsweise von Memon et al. [MBN03] und Merlo et al. [MGK+93] publiziert, betrachten jedoch andere Szenarien und setzen andere Mittel ein.

## 6 Schluss

Wir haben in diesem Artikel einen kurzen Blick auf eine neue statische Analyse mit unmittelbarem Nutzen für den Anwender geworfen: Die Untersuchung graphischer Oberflächen. Neben einer skizzenhaften Beschreibung der Analyse wurden einige wichtige Anwendungen genannt, die von der Analyse unterstützt werden. Für unsere Implementierung im Bauhaus-Projekt haben wir über ermutigende Testresultate berichtet. Laufende Erweiterungen der Analyse werden den Nutzen weiter erhöhen.

## Literatur

- [And94] L.O. Andersen. *Program Analysis and Specialization for the C Programming Language*. Dissertation, University of Copenhagen, 1994.
- [CLM03] Keith Chan, Zhi Cong Leo Liang und Amir Michail. Design Recovery of Interactive Graphical Applications. In *Proceedings of the 25<sup>th</sup> International Conference on Software Engineering (ICSE)*, Seiten 114 – 124, 2003.
- [Das00] Manuvir Das. Unification-based Pointer Analysis with Directional Assignments. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, Seiten 35–46, 2000.
- [Lia00] Shih-Wei Liao. *SUIF Explorer: An Interactive and Interprocedural Parallelizer*. Dissertation, Stanford University, 2000.
- [MBN03] Atif Memon, Ishan Banerjee und Adithya Nagarajan. GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing. In *Proceedings of the 10<sup>th</sup> Working Conference on Reverse Engineering (WCRE)*, Seiten 260 – 269, 2003.
- [MGK<sup>+</sup>93] E. Merlo, J.-F. Girard, K. Kontogiannis, R. Panangaden und R. De Mori. Reverse Engineering of User Interfaces. In *Proceedings of the 1<sup>st</sup> Working Conference on Reverse Engineering (WCRE)*, Seiten 171 – 179, 1993.
- [Mic02] Amir Michail. Browsing and Searching Source Code of Applications written using a GUI Framework. In *Proceedings of the 24<sup>th</sup> International Conference on Software Engineering (ICSE)*, Seiten 327 – 337, 2002.
- [RVP06] Aoun Raza, Gunther Vogel und Erhard Ploedereder. Bauhaus – A Tool Suite for Program Analysis and Reverse Engineering. In *Reliable Software Technologies, Ada Europe 2006 (LNCS 4006)*, Seiten 71 – 82, 2006.
- [Sta07] Stefan Staiger. Static Analysis of Programs with Graphical User Interface. In *CSMR '07: Proceedings of the 11<sup>th</sup> European Conference on Software Maintenance and Reengineering*, Seiten 252 – 261. IEEE Computer Society, 2007.
- [Ste96] B. Steensgaard. Points-to Analysis in Almost Linear Time. In *Proceedings of the 23<sup>rd</sup> ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Seiten 32 – 41, 1996.
- [Tar72] Robert Endre Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal of Computing* 1 (2), Seiten 146 – 160, 1972.