Graphtransformationen zur Systemstabilitätsanalyse

Stephan Epp

11. Januar 2026

Inhaltsverzeichnis

1 Einführung

1.1 Problemstellung

Die Analyse komplexer Softwaresysteme erfordert Methoden zur Modellierung und Verifikation von Systemzuständen und deren Übergängen. Traditionelle Ansätze stoßen bei der Identifikation von Stabilitätszuständen, Zyklen und Ruhelagen an ihre Grenzen, insbesondere wenn Systeme durch strukturelle Transformationen beschrieben werden.

Die zentrale Problemstellung dieser Arbeit ist: **Wie lassen sich Systemzustände effizient vergleichen und analysieren, um Stabilitätseigenschaften, Ruhelagen und zyklische Verhalten in transformationsbasierten Systemen zu identifizieren?**

1.2 Motivation

Graphtransformationen sind eine Methode zur Modellierung von Zustandsübergängen in Softwaresystemen. Dabei werden:

- Systemzustände als Graphen $G = (V, E)$ dargestellt
- Zustandsübergänge durch farbcodierte Transformationsregeln $L \to R$ beschrieben
- Kontextelemente (schwarz), zu löschende Elemente (rot) und neue Elemente (grün) unterschieden

Ein praktisches Beispiel ist die Modellierung einer Ampelkreuzung, bei der jeder Ampelzustand (Grün, Gelb, Rot, Rot-Gelb) als Graph repräsentiert wird und Übergänge zwischen den Zuständen als Transformationsregeln definiert sind.

Die Systemstabilitätsanalyse ermöglicht es:

- Ruhelagen zu identifizieren (Zustände, die sich nicht mehr ändern)
- Zyklen zu erkennen (wiederkehrende Zustandsmuster)
- Längste Subgraph-Sequenzen zu finden (monotone Systemerweiterungen)
- Aussagen über Performance und Sicherheit zu treffen

Die Effizienz dieser Analyse hängt maßgeblich vom verwendeten Subgraph Algorithmus ab, der in $O(n^3)$ Zeit entscheiden kann, ob ein Graph in einem anderen enthalten ist.

2 Grundlagen

2.1 Definitionen

Definition 1 (Farbcodierter Graph)**.** Ein farbcodierter Graph ist ein Tripel $G = (V, E, c)$ mit einer Knotenmenge $V = \{v_1, v_2, \ldots, v_n\}$, einer Kantenmenge $E \subseteq V \times V$ und einer Farbfunktion $c : (V \cup E) \to \{\text{schwarz, rot, grün}\}$.

Die Farbcodierung hat folgende Bedeutung: **Schwarz**: Kontextelemente, die unverändert bleiben, **rot**: Elemente, die gelöscht werden, **grün**: Elemente, die neu erzeugt werden.

Definition 2 (Graphtransformation)**.** Eine Graphtransformation ist eine Regel $t : L \to R$ mit:

- Linke Seite L (Vorbedingung): enthält schwarze und rote Elemente
- Rechte Seite R (Nachbedingung): enthält schwarze und grüne Elemente
- Schwarze Elemente in L und R sind identisch

Definition 3 (Systemzustand)**.** Ein Systemzustand S_i zum Zeitpunkt i ist charakterisiert durch:

$$S_i = (i, t_i, G_i, T_{i-1}, A_i)$$

wobei i die Schrittnummer, t_i der Zeitstempel, G_i der Graph des Zustands, T_{i-1} die angewendete Transformation und A_i die Adjazenzmatrix von G_i sind.

Definition 4 (Subgraph-Sequenz)**.** Eine Subgraph-Sequenz ist eine Folge von Systemzuständen

$$\text{Seq} = (S_i, S_{i+1}, \ldots, S_j)$$

mit der Eigenschaft, dass $G_k \subseteq G_{k+1}$ für alle $k \in \{i, \ldots, j-1\}$.

2.2 Graphtransformationen

2.2.1 Anwendung einer Transformation

Die Anwendung einer Transformation $t : L \to R$ auf einen Graph G erfolgt in drei Schritten:

Algorithm 1 Anwendung einer Graphtransformation

Require: Graph G, Transformation $t = (L, R)$
Ensure: Transformierter Graph G'

```
1: Schritt 1: Prüfe ob L mit G übereinstimmt
2: if nicht übereinstimmend then
3:     return Fehler
4: end if
5: Schritt 2: G' ← Kopie von G
6: Schritt 3: Entferne alle roten Knoten aus L in G'
7: Schritt 4: Entferne alle roten Kanten aus L in G'
8: Schritt 5: Füge alle grünen Knoten aus R zu G' hinzu
9: Schritt 6: Füge alle grünen Kanten aus R zu G' hinzu
10: return G'
```

2.2.2 Matching-Bedingung

Eine Transformation $t : L \to R$ ist auf Graph G anwendbar, wenn:

$$\forall v \in V_L^{\text{schwarz}} : v \in V_G \tag{1}$$
$$\forall v \in V_L^{\text{rot}} : v \in V_G \tag{2}$$
$$\forall e \in E_L^{\text{schwarz}} : e \in E_G \tag{3}$$
$$\forall e \in E_L^{\text{rot}} : e \in E_G \tag{4}$$

Beispiel 1 (Ampeltransformation: Grün → Gelb)**. Linke Seite** L**:** Schwarze Knoten: `ampel`, `nord`, `süd`, rote Knoten: `grün`, rote Kanten: `ampel`→`grün`, `grün`→`nord`, `grün`→`süd`. **Rechte Seite** R**:** Schwarze Knoten: `ampel`, `nord`, `süd`, grüne Knoten: `gelb`, grüne Kanten: `ampel`→`gelb`, `gelb`→`nord`, `gelb`→`süd`.

3 Systemstabilitätsanalyse

In diesem Kapitel wird die Idee der Systemstabilitätsanalyse vorgestellt. Es werden Algorithmen zur Identifikation von Systemeigenschaften beschrieben.

Die Systemstabilitätsanalyse basiert auf der Beobachtung, dass Systeme durch wiederholte Anwendung von Transformationen bestimmte Verhaltensmuster zeigen:

1. **Monotone Erweiterung**: Jeder neue Zustand erweitert den vorherigen (Subgraph-Beziehung)
2. **Stabile Zustände**: Zustände, die sich nicht mehr ändern (Ruhelagen)
3. **Zyklisches Verhalten**: Zustände, die sich periodisch wiederholen

Für eine Sequenz von n Systemzuständen $S_0, S_1, \ldots, S_{n-1}$ wird eine Vergleichsmatrix $M \in \{0,1\}^{n \times n}$ berechnet:

$$M_{ij} = \begin{cases} 1 & \text{falls } G_i \subseteq G_j \\ 0 & \text{sonst} \end{cases} \tag{5}$$

Die Berechnung von M_{ij} erfolgt mittels des Subgraph Algorithmus in $O(n^3)$ Zeit.

Satz 1 (Komplexität der Vergleichsmatrix)**.** Die Berechnung der vollständigen Vergleichsmatrix für n Systemzustände erfordert $O(n^5)$ Zeit.

Beweis. Es sind n^2 Paare von Zuständen zu vergleichen. Jeder Vergleich mittels Subgraph Algorithmus benötigt $O(n^3)$ Zeit. Gesamtkomplexität: $O(n^2 \cdot n^3) = O(n^5)$. □

Satz 2 (Korrektheit der Sequenzberechnung)**.** Der Algorithmus 2 findet die längste zusammenhängende Subgraph-Sequenz in $O(n^2)$ Zeit.

Beweis. Die dynamische Programmierung garantiert, dass $dp[i]$ die Länge der längsten Sequenz speichert, die bei Zustand i endet. Die Rekonstruktion über die *parent*-Zeiger liefert die tatsächliche Sequenz. Die Zeitkomplexität ergibt sich aus den zwei verschachtelten Schleifen: $O(n^2)$. □

Die Identifikation der längsten Subgraph-Sequenz erfolgt durch dynamische Programmierung. Algorithmus 2 beschreibt die Arbeitsweise zur Ermittlung der längsten Subgraph-Sequenz.

Algorithm 2 Längste Subgraph-Sequenz

Require: Vergleichsmatrix $M \in \{0,1\}^{n \times n}$
Ensure: Längste Subgraph-Sequenz

$dp[i] \leftarrow 1$ für alle $i \in \{0, \ldots, n-1\}$
$parent[i] \leftarrow -1$ für alle $i \in \{0, \ldots, n-1\}$
for $i = 1$ to $n-1$ **do**
    for $j = 0$ to $i-1$ **do**
        if $M[j][i] = 1$ und $j \neq i$ **then**
            if $dp[j] + 1 > dp[i]$ **then**
                $dp[i] \leftarrow dp[j] + 1$
                $parent[i] \leftarrow j$
            end if
        end if
    end for
end for
$max\_length \leftarrow \max(dp)$
Rekonstruiere Sequenz über *parent*-Zeiger
return Sequenz

Definition 5 (Ruhelage)**.** Ein Zustand S_i ist stabil (Ruhelage), wenn:

$$G_i = G_{i+1} \quad \text{und} \quad M[i][i+1] = 1 \quad \text{und} \quad M[i+1][i] = 1 \tag{6}$$

Algorithmus 3 beschreibt die Arbeitsweise zur Identifikation von stabilen Zuständen.

Algorithm 3 Identifikation stabiler Zustände

Require: Vergleichsmatrix M, Systemzustände $\{S_0, \ldots, S_{n-1}\}$
Ensure: Menge stabiler Zustände

$stable \leftarrow \emptyset$
for $i = 0$ to $n-2$ **do**
    if $M[i][i+1] = 1$ und $M[i+1][i] = 1$ **then**
        if $A_i = A_{i+1}$ **then** ▷ Adjazenzmatrizen sind identisch
            $stable \leftarrow stable \cup \{i\}$
        end if
    end if
end for
return *stable*

Definition 6 (Zyklus)**.** Ein Zyklus liegt vor, wenn ein Zustand S_i zu einem späteren Zeitpunkt S_j $(j > i)$ identisch wiederkehrt:

$$\text{Zyklus}(i,j) \Leftrightarrow G_i = G_j \quad \text{für } j > i \tag{7}$$

Die Zykluslänge ist $l = j - i$.

Algorithmus 4 beschreibt die Arbeitsweise zur Identifikation von Zyklen im System.

Algorithm 4 Identifikation von Zyklen

Require: Systemzustände $\{S_0, \ldots, S_{n-1}\}$
Ensure: Menge von Zyklen (i, j)

```
cycles ← ∅
for i = 0 to n − 2 do
    for j = i + 1 to n − 1 do
        if A_i = A_j then                    ▷ Adjazenzmatrizen sind identisch
            cycles ← cycles ∪ {(i, j)}
        end if
    end for
end for
return cycles
```

4 Subgraph Algorithmus

In diesem Kapitel wird auf den Subgraph Algorithmus eingegangen und verdeutlicht, wie dieser Algorithmus bei der Systemstabilitätsanalyse eingesetzt wird.

Der Subgraph Algorithmus ist das zentrale Element der Systemstabilitätsanalyse. Er entscheidet in einer Laufzeit von $O(n^3)$, ob eine Subgraph-Beziehung $G_i \subseteq G_j$ existiert. Die Analyse der Effizienz hängt direkt von der Effizienz des Subgraph Algorithmus ab:

- **Vergleichsmatrix**: Die Ermittlung der Vergleichsmatrix erfordert $O(n^2)$ Aufrufe des Subgraph Algorithmus.
- **Gesamtlaufzeit**: Der Subgraph Algorithmus hat eine Laufzeit von $O(n^3)$, damit ergibt sich eine Gesamtlaufzeit von $O(n^5)$.

Um den Subgraph Algorithmus zu nutzen, müssen die erforderlichen Datenstrukturen für die Anwendung des Subgraph Algorithmus bereit gestellt werden. Die Graphklasse bietet Methoden zur Konvertierung zwischen Graph-Repräsentation und Adjazenzmatrix:

Listing 1: Graph-Adjazenzmatrix Konvertierung

```python
def to_adjacency_matrix(self) -> Tuple[np.ndarray, Dict[str, int]]:
        """Konvertiert Graph in Adjazenzmatrix."""
        node_list = sorted(self._nodes.keys())
        n = len(node_list)
        node_to_idx = {node_id: idx for idx, node_id in enumerate(node_list)}

        matrix = np.zeros((n, n), dtype=int)
        for edge in self._edges:
                i = node_to_idx[edge.from_node]
                j = node_to_idx[edge.to_node]
                matrix[i][j] = 1

        return matrix, node_to_idx
```

5 Algorithmus

In diesem Kapitel wird der Algorithmus zur Systemstabilitätsanalyse beschrieben.

5.1 Arbeitsweise

Es folgt der vollständige Algorithmus zur Systemstabilitätsanalyse.

Algorithm 5 Systemstabilitätsanalyse

Require: Anfangszustand G_0, Transformationen $\{t_1, \ldots, t_m\}$, Schritte k
Ensure: Analyseergebnisse

$states \leftarrow [S_0]$ mit $S_0 = (0, \text{now}(), G_0, \text{null}, A_0)$
for $i = 1$ to k **do**
    $t \leftarrow t_{((i-1) \mod m)+1}$ ▷ Zyklische Transformation
    $G_i \leftarrow t.\text{apply}(G_{i-1})$
    $S_i \leftarrow (i, \text{now}(), G_i, t, A_i)$
    $states \leftarrow states \cup \{S_i\}$
end for

$M \leftarrow \text{Matrix}(|states| \times |states|)$
for $i = 0$ to $|states| - 1$ **do**
    for $j = 0$ to $|states| - 1$ **do**
        if $i = j$ **then**
            $M[i][j] \leftarrow 1$
        else
            $M[i][j] \leftarrow \text{Subgraph}(A_i, A_j)$
        end if
    end for
end for

$sequences \leftarrow \text{FindLongestSequences}(M)$
$stable \leftarrow \text{FindStableStates}(M, states)$
$cycles \leftarrow \text{FindCycles}(states)$

return $(M, sequences, stable, cycles)$

Satz 3 (Laufzeit)**.** Die Systemstabilitätsanalyse für Transformationsschritte mit Graphen der Größe n hat eine Laufzeit von $O(n^5)$, die von der Anzahl der Transformationsschritte abhängt.

Beweis. Der Beweis geht direkt aus der Analyse der Effizienz aus Kapitel 4 hervor. □

6 Zusammenfassung

Die vorliegende Arbeit beschreibt die formale Methode zur Modellierung von Systemzuständen durch farbcodierte Graphen. Außerdem beschreibt sie einen effizienten Algorithmus zur Systemstabilitätsanalyse basierend auf dem Subgraph Algorithmus.

6.1 Ausblick

Für die Weiterentwicklung der Systemstabilitätsanalyse durch Graphtransformationen ergeben sich vielfältige Forschungsrichtungen, die sich in drei Bereiche gliedern lassen: Optimierungen des bestehenden Algorithmus, konzeptionelle Erweiterungen sowie neue Anwendungsgebiete.

6.1.1 Optimierungen

Im Bereich der Optimierungen bietet die Parallelisierung der Vergleichsmatrix-Berechnung erhebliches Potenzial. Da die einzelnen Subgraph-Vergleiche voneinander unabhängig sind, können sie problemlos auf mehrere Prozessorkerne verteilt werden. Dies würde bei modernen Multi-Core-Systemen eine nahezu lineare Beschleunigung ermöglichen und die Gesamtlaufzeit deutlich reduzieren.

Ein weiterer vielversprechender Ansatz ist die inkrementelle Analyse. Anstatt bei jedem neuen Systemzustand die gesamte Vergleichsmatrix neu zu berechnen, könnten nur die Vergleiche des neuen Zustands mit allen bestehenden Zuständen durchgeführt werden. Dies würde die Laufzeit für jeden zusätzlichen Schritt reduzieren und die Analyse von langfristigen Systemverläufen erheblich beschleunigen.

Für sehr große Systeme mit mehr als tausend Zuständen könnten approximative Verfahren eingesetzt werden. Diese würden einen bewussten Trade-off zwischen Genauigkeit und Laufzeit eingehen, indem sie beispielsweise nur eine Stichprobe von Zuständen vergleichen oder heuristische Abbruchkriterien verwenden. Dadurch ließe sich die Analyse auch auf Systeme anwenden, für die eine vollständige Berechnung zu zeitaufwändig wäre.

6.1.2 Erweiterungen

Konzeptionell eröffnen sich zahlreiche Erweiterungsmöglichkeiten des Grundansatzes. Die Integration gewichteter Graphen würde es ermöglichen, quantitative Eigenschaften von Systemzuständen zu modellieren. Kantengewichte könnten beispielsweise Wahrscheinlichkeiten, Kosten oder Kapazitäten repräsentieren. Der Subgraph Algorithmus müsste entsprechend erweitert werden, um nicht nur strukturelle Gleichheit, sondern auch die Verträglichkeit von Gewichten zu prüfen.

Attributierte Graphen stellen eine weitere wichtige Erweiterung dar. Dabei würden Knoten und Kanten mit Attributen versehen, die bei den Vergleichen berücksichtigt werden müssen. Dies ist besonders relevant für die Modellierung realer Systeme, bei denen Objekte durch vielfältige Eigenschaften charakterisiert sind. Der Vergleichsalgorithmus müsste dann sowohl strukturelle als auch attributbasierte Übereinstimmungen prüfen.

Probabilistische Transformationen würden die Modellierung nicht-deterministischer Systeme ermöglichen. Anstatt deterministischer Regeln könnten Transformationen mit Wahrscheinlichkeiten versehen werden, was die Analyse stochastischer Systeme erlaubt. Die Stabilitätsanalyse müsste dann um statistische Methoden erweitert werden, um Aussagen über die erwartete Systemdynamik treffen zu können.

Die Unterstützung hierarchischer Graphen würde Systeme mit mehreren Abstraktionsebenen zugänglich machen. Knoten könnten selbst wieder Graphen repräsentieren, was die Modellierung komplexer Systemarchitekturen erheblich vereinfachen würde. Der Subgraph Algorithmus müsste rekursiv auf verschiedenen Hierarchieebenen operieren können.

Schließlich wäre die Integration von Echtzeitaspekten für viele praktische Anwendungen essentiell. Durch die Berücksichtigung von Zeitbedingungen und Deadlines bei

Transformationen könnten zeitkritische Systeme analysiert werden. Die Stabilitätsanalyse müsste dann auch temporale Eigenschaften wie maximale Antwortzeiten oder Periodizität untersuchen.

6.1.3 Anwendungsgebiete

Die entwickelte Methodik eröffnet vielfältige neue Anwendungsgebiete. Im Bereich der Cyber-Physical Systems könnten eingebettete Systeme und IoT-Anwendungen analysiert werden. Die Wechselwirkung zwischen physikalischen Prozessen und Software-Steuerung ließe sich durch Graphtransformationen elegant modellieren, wobei die Stabilitätsanalyse kritische Systemzustände identifizieren könnte.

Ein besonders aktuelles Anwendungsfeld ist die Blockchain-Technologie. Die Verifikation von Smart Contract Zustandsübergängen stellt eine große Herausforderung dar, da Fehler in Contracts zu erheblichen finanziellen Verlusten führen können. Durch Modellierung der Contract-Zustände als Graphen und der Transaktionen als Transformationen könnte die Stabilitätsanalyse unerwünschte Zustände oder Endlosschleifen frühzeitig erkennen.

In der Systembiologie würde die Modellierung von Gen-Regulationsnetzwerken profitieren. Gene und ihre Interaktionen lassen sich natürlich als Graphen darstellen, während Regulationsprozesse durch Transformationen beschrieben werden können. Die Stabilitätsanalyse könnte helfen, stabile Expressionsmuster oder oszillierende Genaktivitäten zu identifizieren.

Schließlich bieten soziale Netzwerke ein interessantes Anwendungsgebiet für die Analyse von Netzwerkdynamiken und Informationsfluss. Die Evolution sozialer Strukturen durch das Hinzufügen oder Entfernen von Verbindungen ließe sich als Folge von Graphtransformationen modellieren. Die Stabilitätsanalyse könnte stabile Communities identifizieren oder die Ausbreitung von Informationen durch das Netzwerk verfolgen.

6.2 Fazit

Die Kombination von Graphtransformationen und dem Subgraph Algorithmus bietet eine leistungsfähige Methode zur Systemstabilitätsanalyse. Die Effizienz des Ansatzes ermöglicht die Analyse realistischer Systeme, während die formale Fundierung korrekte und nachvollziehbare Ergebnisse garantiert.

Die praktische Anwendbarkeit wird durch die Implementierung in Python und einem Beispiel einer Ampelkreuzung demonstriert. Die Analyse liefert wertvolle Einblicke in Systemverhalten, Stabilitätseigenschaften und potenzielle Fehlerquellen.

Der Subgraph Algorithmus ist dabei das zentrale Element, das durch seine Effizienz und Korrektheit die gesamte Analyse erst praktikabel macht. Ohne diese effiziente Subgraph-Erkennung wäre die paarweise Vergleichsmatrix-Berechnung für realistische Systemgrößen nicht durchführbar.

6.3 Implementierung

Der Python Code für die Implementierung den Algorithmus der Systemstabilitätsanalyse und die Tests wurden mit Claude AI generiert. Der Code ist verfügbar unter: `https://github.com/hjstephan/graph-trans`. In diesem Repository befindet sich auch der generierte Code Coverage Report im HTML Format.