Effizientes Auffinden von Fehlern in Verteilten Systemen mit Symbolischer Ausführung

Oscar Soria Dustmann
Communication and Distributed Systems (ComSys)
RWTH Aachen University
oscar.dustmann@comsys.rwth-aachen.de

***Zusammenfassung*—Zum Testen verteilter bzw. vernetzter Systeme existieren zahlreiche Werkzeuge, etwa [1]–[3], welche für das Rapid Prototyping und die Fehler-Rekonstruktion unerlässlich sind. Durch verschieden stark abstrahierte Modelle lassen sich auf diese Weise, bereits früh in der Entwicklung verteilter Systeme, Aussagen über Performanz, Konsistenz und Korrektheit von Netzwerkprotokollen treffen. Diese Verfahren arbeiten auf explizit für den Testvorgang entworfenen Modellimplementierungen, welche zwar teilweise Quelltext mit der Produktionssoftware gemein haben, aber viele Aspekte und potentielle Fehlerquellen einer tatsächlichen Software unberücksichtigt lassen.**

In diesem Artikel werden die Grundlagen von Testparadigmen beschrieben, welche die Ausführung unmodifizierter Software erlauben und dabei ermöglichen, eine hohe Zuversicht in die getesteten Systeme zu gewinnen. Der Fokus liegt dabei auf der Optimierung von Algorithmen um die redundante Ausführung des Systems zu minimieren, wodurch sich die zum Testen benötigte Zeit um Größenordnungen verringert.

I. Einführung

Software für verteilte Systeme zu testen ist ein aufwendiges und zeitraubendes Problem. Zum einen sind einzelne, spezielle Testfälle eines verteilten Systems manuell schwierig zu reproduzieren, da die Umweltbedingungen – wie beispielsweise bei einem Sensornetzwerk – bei wiederholten Ausführungen zu unterschiedlichem Verhalten des Systems führen können. Zum anderen treten Fehler oft erst nach dem Austausch vieler Datenpakete zwischen Netzwerkknoten auf, was bei der simulierten bzw. automatischen Ausführung auch trotz hochentwickelter Verfahren zu langen Ausführungszeiten führt.

A. Was bedeutet Effizienz?

Infolge dieser Schwierigkeiten muss das effiziente, automatische Auffinden von Fehlern in einem verteilten System mit zwei orthogonalen Metriken beurteilt werden. Zunächst muss die Laufzeit eines zur Fehlersuche eingesetzten Werkzeugs minimiert werden (*Performanz*). Zusätzlich ist die von einem Menschen zu verrichtende Arbeit, um die Fehlersuche mithilfe des Werkzeuges zu ermöglichen, maßgebend (*funktionale Anwenderfreundlichkeit*).

Es ist zu beachten, dass sich diese Metriken auf den Erfolg der Fehlersuche beziehen. Hierbei wird Erfolg entweder in der Anzahl und Relevanz konkret nachgewiesener Fehlverhalten oder in der Zuversicht in deren Abwesenheit gemessen.

B. Wie kann Effektivität gemessen werden?

Bei nicht-verteilten Systemen kann Zuversicht in die Abwesenheit von Fehlern erreicht werden, indem die Abdeckung von Quelltextzeilen maximiert wird. Das heißt, dass das System nicht nur einmal ausgeführt wird, sondern wiederholt mit verschiedenen Eingaben. Beispielsweise können zwei zufällig gewählte Eingaben dazu führen, dass bei einer Fallunterscheidung im Quelltext die Programmausführung für die erste Eingabe den Quelltext für einen der Fälle ausführt, während für die zweite Eingabe die Ausführung den Quelltext des anderen Falls verfolgt. Bei einer solchen Situation, bei der sich ein *Ausführungspfad* in zwei unabhängige Pfade aufspaltet, spricht man von einer *Verzweigung* und entsprechend bei einer Menge von Ausführungspfaden von einem Baum.

Selbst eine vollständige Abdeckung des gesamten Quelltexts garantiert jedoch keine Fehlerfreiheit, da das Auftreten von Fehlern in einer bestimmten Quelltext-Zeile von der Wertebelegung der involvierten Variablen abhängt (beispielsweise bei einer Division abhängig davon, ob der Divisor gleich Null ist oder nicht). Daher ist es häufig zweckmäßiger, anstatt die Abdeckung von Quelltext-Zeilen zu betrachten, über die Abdeckung von einzelnen Pfaden (*Pfadabdeckung*) zu sprechen. Diese erlaubt eine feinere Aussage über das zu testende Programm, obschon auch eine vollständige oder hohe Pfadabdeckung kein Garant für die Abwesenheit von Fehlern ist.

Durch die inhärente Parallelität eines verteilten Systems muss die Zuversicht von der Quelltext- oder Pfadabdeckung bezogen auf mehrere Netzwerkknoten abhängen. Hierbei ist vollständige Abdeckung schwieriger zu definieren als im nicht-verteilten Fall: Etwa ist das Kreuzprodukt der lokalen Abdeckung aller Knoten quantitativ ungeeignet, da es eine Vielzahl von Kombinationen enthält, welche unerreichbar sind. Dennoch eignet sich dieses Kreuzprodukt, um eine qualitative Aussage über Verfahren zur Fehlersuche zu treffen. Insbesondere erlaubt es den Vergleich zweier Algorithmen, welche die gleiche Abdeckung erzielen, hinsichtlich ihrer Performanz.

C. Inhalt

Dieser Artikel gibt eine Einführung in die Anwendung eines Verfahrens – der Symbolischen Ausführung – auf verteilte Systeme, welches auf nicht-verteilten Systemen bewiesen hat, sehr hohe Abdeckung zu erzielen [4]. Dabei werden verschiedene Ansätze diskutiert, welche die selbe Pfadabdeckung (und

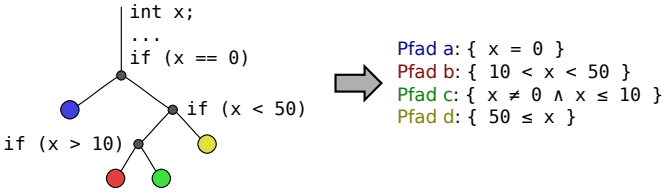


Abbildung 1. Ausführung eines Programms, welches vier verschiedene Verhalten aufweist. Folglich ist es ausreichend, vier Pfade mit symbolischen Werten auszuführen.

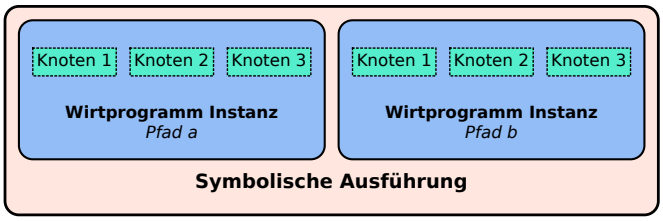


Abbildung 2. Ausführung eines auf drei Knoten verteilten Systems mithilfe eines einhüllenden Wirtprogramms. Das Ausführungswerkzeug hat zu diesem Zeitpunkt zwei Pfade entdeckt: Pfad a und Pfad b. Die verschachtelte Ausführung des Gastsystems geschieht opak.

damit auch die selbe Quelltext-Abdeckung) in kürzerer Zeit erzielen und damit eine bessere Performanz aufweisen.

II. Symbolische Ausführung

Betrachtet man auf einem Sim- oder Emulator eines einzelnen Netzwerkknoten – bzw. eines nicht-verteilten Systems – die Ausführung zweier verschiedener Ausführungspfade, so muss es eine Verzweigung geben, ab der sich diese Pfade unterscheiden. Nehmen wir an, dass beide Pfade nach k Instruktionen terminieren und die Verzweigung nach $i < k$ Instruktionen auftritt. Bei randomisiertem, sequentiellem Testen, würden zunächst k Instruktionen des ersten Pfades abgearbeitet, und anschließend die k Instruktionen des zweiten. Von diesen insgesamt $2 \cdot k$ Instruktionen sind jedoch jeweils die ersten i identisch. Die erneute Ausführung ist daher redundant, da der *Zustand* des Systems für beide Pfade der gleiche ist.

Redundante Ausführung kann systematisch eliminiert werden, indem Äquivalenzklassen von Pfaden zu einem einzigen Pfad zusammengefasst werden. Hierzu muss die Bearbeitung eines Pfades jedoch alle möglichen Wertebelegungen zulassen. Die Repräsentation solcher Wertemengen wird als *symbolischer Wert* bezeichnet. Erreicht die Ausführung eines Pfades eine Fallunterscheidung, welche von einem symbolischen Wert abhängt, muss der Ausführungspfad für alle *erfüllbaren* Fälle in einzelne Pfade verzweigen. Zusätzlich wird der symbolische Wertebereich für die jeweiligen Pfade auf die jeweils gültigen Wertebelegungen eingeschränkt. In Abbildung 1 wird dies anhand eines Programms mit drei Fallunterscheidungen demonstriert. Das Verhalten dieses Systems für x gleich 42 unterscheidet sich nicht vom Verhalten für x gleich 47. Daher können diese durch die Ausführung eines einzigen Pfades, nämlich Pfad b, untersucht werden.

Diese Jahrzehnte alte Idee [5] konnte erst nach der Jahrtausendwende breite praktische Anwendung finden [4]: Zum einen sind die Anforderungen an Arbeitsspeicher und Prozessor für die bloße Ausführung bereits sehr hoch, und zum anderen erfordern die hierbei entstehenden NP-vollständigen Probleme (etwa Erfüllbarkeitsprüfung) hochoptimierte Algorithmen [6].

III. Symbolische Ausführung verteilter Systeme

Aus praktischer Sicht kann ein Werkzeug, welches Symbolische Ausführung unterstützt, nicht unmittelbar auf ein Programm für ein verteiltes System angewandt werden. Der wesentliche Unterschied liegt in der Nebenläufigkeit des Systems, welche unabhängige Prozessoren erfordert. Um das Problem, ein solches System symbolisch auszuführen, auf die nicht-verteilte Version zu reduzieren, für die es bereits Ansätze gibt [4], bietet es sich an, ein Steuerprogramm zu implementieren. Dieses sequenziert, ähnlich wie ein Netzwerk-Simulator, die Parallelität des Systems mithilfe einer geeigneten Strategie; Beispielsweise unter Verwendung von Warteschlangen für diskrete Ereignisse (DES[1]) oder mit instruktionsweiser Ausführung des Gastsystems.

Solch ein in sich abgeschlossenes Programm lässt sich direkt mit vorhandenen Werkzeugen ausführen (siehe Abbildung 2). Hierbei stellt ein Zustand des Wirtprogramms einen logischen Zustand des gesamten verteilten Systems dar. Ein solcher logischer Zustand wird als *Szenario* bezeichnet: Er beschreibt eine zulässige Momentaufnahme des gesamten Netzwerks.

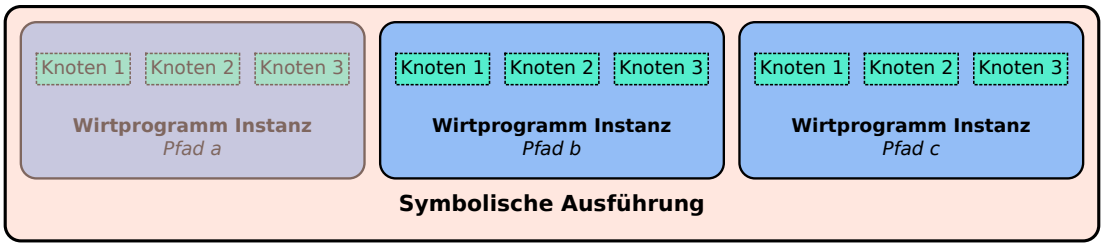
Dieser Ansatz, obwohl konzeptionell hilfreich, ist aus zwei Gründen problematisch. Zunächst ist es in der Praxis nicht trivial, ein solches Steuerprogramm zu implementieren. Das Wirtprogramm müsste den Maschinencode der Gastprogramme wie ein Emulator interpretieren, was sich negativ auf die Ausführungsgeschwindigkeit niederschlüge. Des Weiteren skaliert dieser Ansatz nicht, wie nachfolgend untersucht.

A. Redundanz

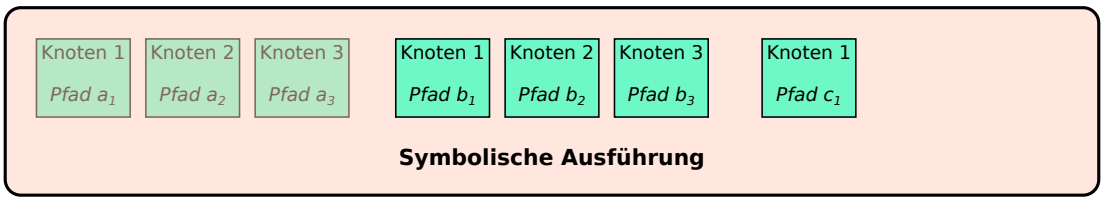
Bereits bei kleinen Systemen können Zustandsräume mit hohen Redundanzen entstehen, welche praktisch nicht handhabbar sind. Diese zentrale Beobachtung ist Kern der nachfolgenden *Symbolischen, Verteilten Ausführung* (SDE) [7]. Um uns dies zu verdeutlichen, betrachten wir eine exemplarische Verzweigung eines Pfades in Abbildung 2 näher.

Angenommen das einhüllende Programm von Pfad b schreitet um eine Instruktion voran und die verschachtelte Ausführung selektiert den eingebetteten Quelltext von, beispielsweise, Knoten 3. Dazu liest das Wirtprogramm seinerseits die nächste Instruktion des zu Knoten 3 passenden Maschinencodes und führt diese aus. Falls der Zustand von Knoten 3 in dieser Instruktion eine von einem symbolischen Wert abhängende Fallunterscheidung durchführt, muss dies auch auf eine Fallunterscheidung im einhüllenden Programm hinauslaufen. Folgerichtig führt diese zu einer Verzweigung

[1] *Discrete Event Simulation*



(a) Bei Verwendung eines Wirtprogramms sind für die symbolische Ausführungseinheit nur drei Pfade sichtbar. Jeweils ein Pfad pro Szenario.



(b) Wird das verteilte System direkt ausgeführt, entstehen mehrere leichtgewichtige Pfade. Die Knoten sind dabei jeweils unabhängig.

Abbildung 3. Symbolische Ausführung dreier Szenarien mit verschiedenen Ansätzen. Ausgegraut ist dabei jeweils das erste Szenario.

der Ausführungspfade des Wirtprogramms. In der internen Darstellung des Ausführungswerkzeuges führt das dazu, den mit Pfad b assoziierten Zustand zu duplizieren und einen neuen Pfad c zu instanziieren. Pfad b und Pfad c beschreiben verschiedene Szenarien, siehe Abbildung 3(a).

Betrachten wir jedoch die nachfolgende Instruktion des Wirtprogramms, welche eine Instruktion von Knoten 1 ausführt: Die Ursache, die zur Verzweigung der beiden Pfade geführt hat, liegt im verschachtelten Zustand von Knoten 3. Folglich ist die auf Pfad b nachfolgend ausgeführte Instruktion von Knoten 1 identisch mit der von Pfad c ausgeführten. Allgemeiner lässt sich feststellen, dass jeder Zustand in zumindest drei Teile partitioniert werden kann: Nämlich jeweils die eingebettete Ausführung der jeweiligen Knoten. Daher müssen ab dieser Verzweigung Pfad b und c mindestens zu $^2/_3$ exakt das gleiche ausführen.

B. Architektur

Durch Elimination der Zwischenschicht kann diese Redundanz vermieden werden. Hierzu werden in der Ausführungseinheit für jeden Knoten unabhängige Pfade verfolgt. Da ein Ausführungspfad zu einem gegebenen Zeitpunkt durch einen Zustand modelliert wird, bedeutet das, dass zu Beginn der Ausführung für jeden Knoten ein Wurzel-Zustand erzeugt wird. In der Praxis wird dies durch Verzweigung des ersten existierenden Zustands erreicht, sodass technisch gesehen stets ein Baum vorliegt, logisch jedoch ein Wald mit einem Baum je Netzwerkknoten. Die nebenläufige Ausführung des Netzwerks stellt kein Hindernis dar, da die Ausführungseinheit bereits dafür ausgelegt ist, verschiedene Versionen eines Programms getrennt voneinander zu untersuchen.

Betrachten wir nun das vorherige Beispiel erneut, fällt auf, dass eine Verzweigung auf einem der Knoten keinen Einfluss auf die Ausführung der restlichen Knoten hat. Die Szenarien, welche zuvor von den zwei Pfaden b und c modelliert wurden, werden nun durch die Pfade b_1, c_1, b_2 und b_3 modelliert – es gibt jedoch keine Pfade c_2 und c_3, siehe Abbildung 3(b). Die Komplexität dieser Pfade entspricht den zuvor beschriebenen Partitionen des Wirtprogramms, sodass obwohl es in dieser Architektur mehr Pfade gibt als zuvor, insgesamt weniger Arbeit vorliegt, da die redundanten Partitionen in b und c als einzelne Pfade b_2 und b_3 vorliegen und daher nicht doppelt ausgeführt werden müssen.

C. Kommunikation

Ein wichtiger Nebeneffekt der Beseitigung redundanter Ausführung ist die Entstehung des *State Mapping* Problems: Sendet ein Zustand ein Paket über das Netzwerk, muss nun die Ausführungseinheit dieses Paket abfangen und anschließend an den Empfängerknoten zustellen. Im Allgemeinen gibt es jedoch mehrere Zustände, welche den Empfängerknoten modellieren. Auch die Identifikation einzelner Szenarien ist nicht mehr eindeutig: Beispielsweise repräsentiert der zu Pfad b_2 gehörende Zustand in Abbildung 3(b) gleich zwei Szenarien, nämlich diejenigen die in Abbildung 3(a) durch die Pfade b und c modelliert wurden.

IV. State Mapping: Auflösung kommunikationsbedingter Inkonsistenzen

Um die globale Konsistenz der Ausführung zu wahren, muss Kommunikation so aufgelöst werden, dass die fehlende Redundanz für das ausgeführte System unsichtbar bleibt. Beispielsweise sendet der rot markierte Knoten in Abbildung 4 ein Paket an den zweiten Knoten. Da sich die einzelnen Knoten unabhängig voneinander verzweigt haben, ist zunächst unklar, wie das Paket zuzustellen ist.

Dieses Beispiel zeigt 5 zu Knoten 1 gehörende Pfade (orange/rot), 6 zu Knoten 2 (türkis) und 4 zu Knoten 3 (violett). Diese 15 Zustände modellieren jedoch $5 \cdot 6 \cdot 4 = 120$ echte Szenarien, da jede Kombination eines orange-roten, eines

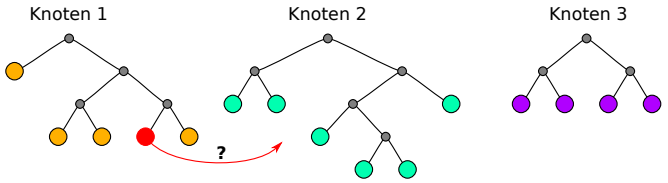


Abbildung 4. Zustellung eines Pakets von Knoten 1 an Knoten 2. Problematisch: Das Paket wird nur in einem Ausführungspfad gesendet (rot), während die anderen Pfade des selben Knotens (orange) kein Paket senden.

türkisfarbenen und eines violetten Zustands einen zulässigen Zustand des Gesamtsystems, bzw. eines gedachten Wirtprogramms, darstellen. Die hierdurch im Vergleich zum Verfahren in III-A entstehende Zustandsreduktion ist sehr hilfreich, um die Performanz des Testwerkzeuges zu erhöhen, welches in diesem Fall nur $^{15}/_{120}$ der Arbeitslast leisten muss – bei der gleichen Abdeckung.

Da jedoch nur ein Zustand auf Knoten 1 existiert, welcher ein Paket versendet, wird dieses Paket folgerichtig nur in $1{\cdot}6{\cdot}4 = 24$ Szenarien versandt, in den restlichen 96 aber nicht. Betrachtet man einen beliebigen der Zustände von Knoten 2 (türkis), fällt auf, dass dieser $5 \cdot 1 \cdot 4 = 20$ Szenarien repräsentiert, nämlich alle Kombinationen von orange-roten mit violetten Zuständen. Von diesen 20 Szenarien wird allerdings nur in denjenigen ein Paket versandt, welche den roten Zustand enthalten, also in $1 \cdot 1 \cdot 4$. Das bedeutet, dass jeder dieser türkisfarbenen Zustände sowohl Szenarien modelliert, in denen eine Paketübertragung stattfindet (und zwar 4), als auch solche, in denen keine Übertragung stattfindet (16).

Ein Zustand kann nicht gleichzeitig ein Paket empfangen und es nicht empfangen. Dieser Widerspruch kann auf verschiedene Weise behoben werden: Die nachfolgenden *State Mapping Algorithmen* leisten genau dies.

A. „Copy-on-Write“: Verallgemeinerte Szenarien

Ein Szenario ist eine Menge von Zuständen, welche dadurch charakterisiert ist, dass sie je Knoten exakt einen Zustand enthält; Und zwar genau solche Zustände, die eine zulässige Momentaufnahme des gesamten verteilten Systems modellieren. Für die algorithmische Lösung des State Mapping Problems ist es hilfreich, das Konzept eines Szenarios mit Mengen zu verallgemeinern, welche mehr als einen Zustand je Knoten enthalten dürfen. Solche Mengen werden *DStates* genannt [7]. DStates enthalten ausschließlich miteinander kompatible Zustände, also Zustände, deren knotenweises Kreuzprodukt eine Menge gültiger Szenarien ergibt.

Das Beispiel in Abbildung 4 besitzt zunächst einen einzigen DState, da keine inkompatiblen Zustände existieren. D. h. für jedes beliebige Paar von Zuständen existiert ein Szenario, welches diese enthält.

Für eine gegebene Übertragung werden Zustände des Empfängerknotens, die mindestens ein Szenario mit dem Sender gemeinsam haben, als *Ziel-Zustände* oder *Ziele* bezeichnet. Alle vom Sender verschiendenen Zustände des Senderknotens hingegen, welche gemeinsame Szenarien mit Zielen haben, werden als *rivalisierende Zustände* oder *Rivalen* bezeichnet, da

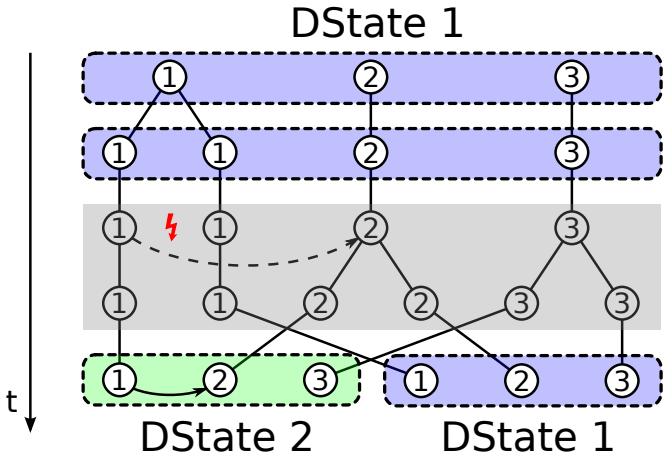


Abbildung 5. Copy-on-Write: Rivalisierte Übertragung eines Pakets von Knoten 1 zu Knoten 2 innerhalb eines DStates (blau). Im schattierten Schritt hat der Algorithmus Kopien der Zustände von Knoten 2 und 3 und einen neuen DState angelegt (grün), in welchem die Übertragung letztendlich durchgeführt wird.

sie mit dem Sender um Ziele konkurrieren. Dies liegt in dem zuvor beschriebenen Widerspruch begründet, dass ein (Ziel-) Zustand nicht gleichzeitig ein Paket empfangen und es nicht empfangen kann.

Bei der Abwicklung der Paketzustellung wird der eindeutige DState d_1 des sendenden Zustands identifiziert, welcher einen Rückschluss auf die Menge aller Ziele zulässt. Ist die Zustellung unrivalisiert, existieren also keine Rivalen, so kann das Paket unmittelbar an alle Ziele zugestellt werden. Andernfalls ist eine unmittelbare Zustellung nicht möglich, ohne die Abdeckung zu verringern. Der „Copy-on-Write“ Algorithmus erzeugt daher zunächst einen neuen DState d_2, weist diesem den sendenden Zustand zu, und entfernt diesen aus dem alten DState d_1. Die hier zugrundeliegende Intuition ist, dass d_2 fortan alle Szenarien enthält, in denen die Paketübertragung stattgefunden hat, während d_1 diejenigen enthält, in denen die Paketübertragung nicht stattgefunden hat. Zu diesem Zweck werden alle Zustände in d_1, die nicht zum Knoten des Senders gehören, kopiert, wobei die Kopie jeweils in d_2 eingefügt wird, während das Original in d_1 bleibt – siehe Beispiel in Abbildung 5.

Dieser Algorithmus holt die zuvor eingesparten Verzweigungen nach, um die auf dem Senderknoten (Knoten 1) getroffene Fallunterscheidung (Senden oder Nicht-Senden) in das Netzwerk propagieren zu können (Empfangen oder Nicht-Empfangen). Seine Implementierung ist vergleichsweise einfach und linear in der Anzahl der Ziel-Zustände, weil die Umsetzung von DStates mit geeigneten Datenstrukturen erlaubt, in konstanter Zeit alle Ziele zu finden sowie in konstanter Zeit zu entscheiden, ob Rivalen existieren.

B. SDS: Verallgemeinerte DStates

Der Nachteil des oben beschriebenen Algorithmus ist, dass redundante Zustände entstehen: Die bei der Auflösung einer Paketübertragung erzeugten Zustände fallen in zwei Kategorien: Ziele (Abbildung 5: Knoten 2) und *Zuschauer*

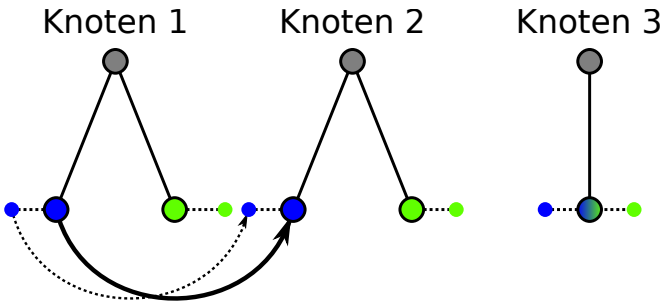


Abbildung 6. Fünf Zustände – große Kreise – mit insgesamt sechs virtuellen Zuständen – kleine Kreise – in zwei DStates: blau und grün. Der Zustand des dritten Knotens befindet sich in beiden DStates (blau/grün schattiert).

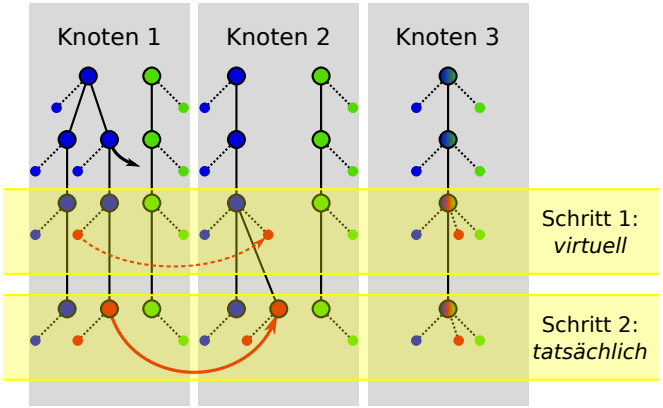


Abbildung 7. Fortführung des Beispiels aus Abbildung 6. In der ersten Zeile verzweigt ein Zustand auf Knoten 1, folglich wird auch sein virtueller Zustand verzweigt (virtuelle Verwandschaftsbeziehung nicht eingezeichnet). Nach einem Sendeversuch in Zeile 2 wird auf der virtuellen Ebene durch den eingebetteten „Copy-on-Write“ Algorithmus in Schritt 1 ein neuer DState angelegt (rot). Zuletzt wird in Schritt 2 die Verzweigung aus der virtuellen Ebene auf die tatsächlichen Zustände propagiert, indem der Zustand auf Knoten 2 durch den einbettenden Algorithmus explizit verzweigt wird.

(Abbildung 5: Knoten 3). Die Zuschauer sind dabei alle zu dem entsprechenden DState, aber weder zum Sender- noch Empfängerknoten gehörenden Zustände. Im Erwartungswert überwiegen sie zahlenmäßig, da typischerweise Netzwerke mit mehr als drei Knoten untersucht werden.

Im Gegensatz zu Ziel-Zuständen unterscheiden sich Zuschauer-Zustände nicht von ihren Kopien, da an sie in keinem Fall ein Paket zugestellt wird. Das bedeutet, dass die Ausführung dieser Zustände und ihrer Kopien redundant ist und ihre durch den State Mapping Algorithmus erzwungene Verzweigung künstlich bleibt. Redundanz dieser Art ist besonders problematisch, wenn große Netzwerke mit viel lokaler Kommunikation untersucht werden. Es ist jedoch wichtig anzumerken, dass die Performanzeinbuße durch die redundante **Ausführung** entsteht, nicht durch die Berechnung des Algorithmus selbst.

Auf Grundlage dieser Einsicht kann ein redundanzfreier Algorithmus entworfen werden. Der Grund, dass die Verbesserung von „Copy-on-Write“ nicht trivial ist, liegt an der Definition von DStates: Um effizient Ziele und Rivalen finden zu können, gehört jeder Zustand zu genau einem DState, was das Anlegen redundanter Zuschauer erfordert. Das kann umgangen werden, indem für den Algorithmus eine zusätzliche Abstraktionsschicht eingeführt wird, welche jedem Zustand eine nicht-leere Menge *virtueller Zustände* zuordnet. Virtuelle Zustände sind sehr kleine Datenstrukturen, welche Modelle für echte Zustände darstellen. Auf diesen lässt sich wiederum der „Copy-on-Write“ Algorithmus anwenden, welcher anstatt echte Zustände zu duplizieren dann virtuelle Zustände dupliziert. Dabei werden die neu angelegten Kopien demselben Zustand zugeordnet wie das Original, sodass ein Zustand mehrere virtuelle Zustände besitzen und sich dadurch indirekt in mehr als einem (virtuellen) DState befinden kann. Beispielsweise besitzt der Zustand von Knoten 3 in Abbildung 6 zwei virtuelle Zustände, die sich im grünen bzw. im blauen DState befinden.

Wird nun ein Paket von einem Zustand an einen Empfängerknoten k verschickt (vgl. Abbildung 7), löst dies ein State Mapping aus, welches seinerseits für alle virtuellen Zustände eine virtuelle Übertragung nach k simuliert. Der eingebettete „Copy-on-Write“ Algorithmus löst daraufhin sämtliche virtuellen Konflikte auf und dupliziert ggf. virtuelle Zustände. Dies hat zunächst keinen Einfluss auf tatsächliche Zustände, da neue, virtuelle Zustände stets dem selben Zustand zugewiesen werden zu denen auch die virtuellen Zustände gehören, von denen sie verzweigt wurden.

Nach dem eingebetteten State Mapping können allerdings neue DStates entstehen, falls virtuelle Zustände mit Rivalen existieren. Unabhängig davon sind alle den Sender enthaltenden DStates D solche, in denen eine Übertragung stattfindet. Folglich sind alle zu Knoten k gehörenden Zustände potentielle Ziele. All diese Zustände können unabhängig voneinander untersucht werden, um die Übertragung abzuschließen: Falls die DStates eines Ziel-Zustands z eine Teilmenge von D sind, bedeutet das, dass z zu *keinem* DState gehört, in dem *keine* Übertragung stattfindet. Es gibt folglich auf dieser Ebene keine Rivalen bezüglich z und das Paket kann unmittelbar an z zugestellt werden. Falls z aber einen DState enthält, in dem keine Übertragung stattfindet, existieren tatsächlich Rivalen. Daher muss z in zwei Zustände verzweigen: Einen Zustand z^+, der ein Paket erhält und einen Zustand z^-, an den – zu diesem Zeitpunkt – kein Paket übertragen wird. Die vorhandenen virtuellen Zustände von z müssen nun auf z^+ und z^- verteilt werden.

Da virtuelle Zustände nur zu genau einem DState gehören, ist es möglich zu entscheiden welche virtuellen Zustände zu einem DState gehören, in welchem eine Übertragung stattfindet. Folglich werden alle zu D gehörenden virtuellen Zustände von z an z^+ zugewiesen und alle anderen an z^-.

Das Propagieren der Änderungen aus der virtuellen Domäne in den Zustandsraum beschränkt sich also darauf, ggf. einzelne Ziel-Zustände zu verzweigen und deren virtuelle Zustände umzusortieren.

V. Diskussion

In diesem Artikel wurden die wesentlichen Ideen aus [8] und [7] zusammengefasst. Der Fokus liegt dabei auf der Optimierung der Symbolischen, Verteilten Ausführung selbst,

während technische Details aus Platzgründen ausgelassen wurden. Diese Ansätze werden am Lehrstuhl für Kommunikation und Verteilte Systeme der RWTH Aachen University in einem Werkzeug umgesetzt [10], das mit einer Open-Source Lizenz heruntergeladen werden kann [11].

Nachfolgend werden Evaluationsergebnisse zusammengefasst, sowie ein Ausblick auf zukünftige Forschung gegeben.

A. Evaluation

Zur Evaluation der vorgestellten Algorithmen existieren verschiedene Ansätze. Zunächst können die Algorithmen selbst mithilfe einer synthetischen Simulation von Sende- und Verzweigoperationen verglichen werden. Eine solche Simulation ist nicht an tatsächlichen Quelltext gebunden, sondern generiert zufällige Anfragen an den Algorithmus und protokolliert dessen Performanz. Hierbei ergeben sich, bereits nach wenigen Sekunden Gesamtlaufzeit, Unterschiede in der Laufzeit und in der Anzahl der erzeugten Zustände von mehreren Größenordnungen – bei der exakt gleichen Pfadabdeckung [8].

Zusätzlich ist es zweckmäßig, die Effizienz der Fehlersuche im Kontext Symbolischer Ausführung echter verteilter Systeme, wie etwa Contiki Netzwerken [9], zu untersuchen. Bei einem Testaufbau mit einem System bestehend aus 100 Knoten, das über eine Dauer von 10 simulierten Sekunden ausgeführt wurde, und in dem mehrere Knoten wiederholt verzweigen und Pakete versenden, ließ sich eine vollständige Abarbeitung in Minuten anstatt Stunden durchführen [7].

B. Weitere Forschung

Der in IV-B vorgestellte Algorithmus reduziert die Anzahl auszuführender Zustände auf das Minimum, sodass beweisbar keine redundanten Zustände existieren [8]. Dies gilt unter der Annahme, dass sich Pakete verschiedener Zustände stets unterscheiden, also verschiedene Zustände nicht zur gleichen Zeit das gleiche Paket verschicken. Das ist in der Praxis bei Datenpaketen häufig der Fall, da diese symbolische Werte enthalten können. Folglich ist die Äquivalenz der Lösungsmengen zweier Pakete nicht effizient entscheidbar[2], weshalb es besser ist, anzunehmen, Pakete unterschieden sich stets.

Allerdings existieren wichtige Fälle, in denen die Paketdaten vollständig konkret sind, ihre Gleichheit also mit geeigneten Datenstrukturen mit einem nur konstanten Mehraufwand entschieden werden kann. Ein Beispiel hierfür sind periodische Beacon-Pakete, welche in drahtlosen Netzwerken als Broadcast ausgesendet werden und typischerweise feste Werte beinhalten. Solche Anwendungsfälle zerstören den Vorteil gegenüber der simpleren „Copy-on-Write“ Strategie. Es werden derzeit auch für diese Fälle Ansätze erforscht, um den Zustandsraum trotz solcher Pakete optimal zu halten.

Des Weiteren wird die Anwendbarkeit der hier vorgestellten Algorithmen auf andere Gebiete, wie etwa der Netzwerksimulation, untersucht, mit der sich mehrere Simulationsläufe parallel durchführen ließen. Dies stellt einen orthogonalen Ansatz zu PDES[3] Ansätzen dar, welche die Ausführung einzelner Szenarien beschleunigen – siehe den Beitrag „*Behavior-aware Probabilistic Synchronization in Parallel Simulations and the Influence of the Simulation Model*“ in diesem Heft.

[2] falls $P \neq NP$

[3] *Parallel Discrete Event Simulation*

Anmerkungen und Danksagungen

Dieser Artikel basiert zu großen Teilen auf den Ergebnissen meiner Bachelorarbeit, die ich im Herbst 2010 an der RWTH Aachen University schrieb [8]. Sie baut auf der Idee Raimondas Sasnauskas' auf, Verteilte Systeme symbolisch auszuführen [12].

Ich danke Klaus Wehrle und Raimondas Sasnauskas für ihre Unterstützung bei sowohl o. g. Arbeit als auch nachfolgenden Projekten.

Literatur

[1] G. J. Holzmann, "The Model Checker SPIN," *IEEE Trans. Softw. Eng.*, 1997.

[2] "The Network Simulator ns-3," http://www.nsnam.org.

[3] "OMNeT++ Network Simulation Framework," http://www.omnetpp.org.

[4] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[5] J. C. King, "Symbolic Execution and Program Testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[6] V. Ganesh and D. L. Dill, "A Decision Procedure for Bit-Vectors and Arrays," in *CAV*, 2007, pp. 519–531.

[7] R. Sasnauskas, O. Soria Dustmann, B. L. Kaminski, K. Wehrle, C. Weise, and S. Kowalewski, "Scalable Symbolic Execution of Distributed Systems," in *ICDCS*, 2011, pp. 333–342.

[8] O. Soria Dustmann, "Scalable Symbolic Execution of Distributed Systems," 2010, Bachelor's Thesis.

[9] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors," in *LCN*, 2004.

[10] "KleeNet Overview and Contributors at ComSys," http://www.comsys.rwth-aachen.de/research/projects/kleenet.

[11] "Public KleeNet Project Page and SCM," https://code.comsys.rwth-aachen.de/redmine/projects/kleenet-public.

[12] R. Sasnauskas, O. Landsiedel, H. Alizai, C. Weise, S. Kowalewski, and K. Wehrle, "KleeNet: Discovering Insidious Interaction Bugs in Wireless Sensor Networks Before Deployment," in *International Conference on Information Processing in Sensor Networks (ACM IPSN/SPOTS)*. New York, NY, USA: ACM, 2010, pp. 186–196.