Gottfried Wilhelm
Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für Praktische Informatik
Fachgebiet Software Engineering

Verbesserung der Controller Synthese für szenariobasierte Spezifikationen

Masterarbeit

im Studiengang Informatik

von

Pascal Elster

Prüfer: Prof. Dr. Joel Greenyer
Zweitprüfer: Prof. Dr. Kurt Schneider
Betreuer: Dipl.-Inform. Daniel Gritzner

Hannover, 04.11.2016

Erklärung der Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbständig und ohne fremde Hilfe verfasst und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, den 04.11.2016

_____

Pascal Elster

Zusammenfassung

Mit ScenarioTools ist der formale Entwurf szenariobasierter Spezifikationen auf vergleichsweise intuitive Art möglich. Jedoch sind bestehende Synthese-Algorithmen eingeschränkt und können nicht jede Spezifikation korrekt verifizieren. Insbesondere der bisher verwendete On-the-Fly-Algorithmus, der unter Umständen den Zustandsraum während der Suche nur teilweise erzeugt, kann nur besonders simple Spezifikationen verarbeiten. In dieser Arbeit werden deshalb Verfahren zur Verbesserung der szenariobasierten Synthese betrachtet. Dazu wurde ein neuer On-the-Fly-Algorithmus für gedächtnislose Generalized Büchi-Spiele entwickelt, der eine weitaus größere Anzahl Spezifikationen korrekt verifizieren kann. Darüber hinaus hat der Algorithmus eine eingeschränkte Unterstützung für GR(1)-Spiele. Weiterhin wurde ein bestehender vollständiger GR(1)-Algorithmus, welcher nicht on-the-fly arbeitet, in ScenarioTools umgesetzt und um Fairness erweitert. Hiermit kann ein abgeschwächtes Umgebungsverhalten umgesetzt werden, falls eine strikte Auslegung der GR(1)-Semantik sinnwidrig ist. Die Evaluation hat gezeigt, dass die Laufzeit der Verfahren trotz größerer unterstützter Ausdrucksstärke vergleichbar mit den bestehenden Verfahren ist. Der neu entwickelte On-the-Fly-Algorithmus weist, trotz erhöhter Komplexität gegenüber dem bestehenden OTF-Algorithmus, bei den betrachteten Testfällen eine niedrigere Laufzeit auf, als Verfahren welche den gesamten Zustandsraum erstellen müssen.

Abstract

ScenarioTools allows the formal design of scenario-based specifications in a comparatively intuitive manner. However, its existing synthesis algorithms are limited and cannot verify every specification correctly. In particular, the currently used on-the-fly algorithm can only process very simple specifications, which negates the benefit of it not necessarily needing to construct the entire state space. In this work, methods for improving the scenario-based synthesis are considered. To that end, a new on-the-fly algorithm for memoryless Generalized Büchi games was developed which can correctly verify significantly more specifications than the existing algorithm. This new algorithm also has partial support for GR(1) games. Additionally, a complete algorithm for GR(1) games, which does not work on-the-fly, was adapted for ScenarioTools. A fairness extension was developed for it which can be used to specify a weaker environment behavior for cases where the strict enforcement of GR(1) semantics leads to nonsensical behavior. Despite having a higher complexity than existing on-the-fly algorithms, the new on-the-fly algorithm has a lower runtime for the evaluated test cases than algorithms that construct the entire state space.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Moderne Systeme bestehen aus einer Vielzahl kommunizierender Komponenten, aus deren Zusammenspiel sich komplexe implizite Abhängigkeiten ergeben, die nicht immer direkt ersichtlich sind. Um die Korrektheit solcher Systeme zu garantieren eignet sich der formale Entwurf. Jedoch ist er aufwändig und bedarf generell Expertenwissen, etwa im Umgang mit temporaler Logik. Dadurch gestaltet er sich kostspieliger und zeitaufwändiger. Dem möchte ScenarioTools[1] mittels szenariobasierter Spezifikationen entgegenwirken. Dort werden Verhaltensabläufe und insbesondere Kommunikationen einzelner Komponenten durch vergleichsweise intuitive Szenarien beschrieben, die auch ohne tiefergehendes Fachwissen verständlich sind. Das Zusammenspiel der Komponenten kann daraufhin auf Widerspruchsfreiheit überprüft und mittels Controller-Synthese eine Strategie abgeleitet werden, welche eine konsistente Ausführung des Gesamtsystems garantiert.

Für diesen Zweck werden die Szenarien auf einen Spielgraphen abgebildet, in dem versucht wird, eine Strategie zu finden, die eine solche konsistente Ausführung garantiert. Eine Hürde dieses Vorgehens ist, dass die Größe des Zustandsraums dieses Spielgraphen mit steigender Anzahl Szenarien exponentiell wächst, wodurch es zu einer Zustandsraumexplosion kommt. Dies schränkt derzeit die Anwendbarkeit szenariobasierter Spezifikationen für große Systeme ein. Ein Ansatz, um die Auswirkungen der Zustandsraumexplosion zumindest in Teilen zu vermeiden, ist die Nutzung von Algorithmen, die *on-the-fly* arbeiten und somit nicht zwangsläufig den kompletten Zustandsraum konstruieren müssen. Jedoch sind die bestehenden On-the-Fly-Verfahren in der unterstützten Ausdrucksstärke eingeschränkt und dadurch nicht uneingeschränkt für den szenariobasierten Entwurf nutzbar. Auch abseits der Einschränkungen dieser Verfahren kann ScenarioTools noch nicht für jede gültige Spezifikation einen Controller ableiten, da keines der bestehenden Verfahren vollständig für die benötigte Ausdrucksstärke ist.

[1] http://scenariotools.org/

1.2 Ziele der Arbeit

Ziel dieser Arbeit ist es, die Controller-Synthese szenariobasierter Spezifikationen um neue Verfahren zu erweitern. Dazu soll zunächst ein On-the-Fly-Algorithmus konzipiert werden, mit dem eine bedeutend größere Anzahl Spezifikationen verifiziert werden kann. Weiterhin soll ScenarioTools um einen Synthese-Algorithmus erweitert werden, der vollständig für die benötigte Ausdrucksstärke der Spezifikationen ist. Für diesen sollen zusätzlich Erweiterungen wie Partial-Order-Reduction und Fairness umgesetzt werden. Anschließend sollen die neu implementierten Verfahren in einer Evaluation mit den bestehenden Verfahren verglichen und auf Anwendbarkeit untersucht werden.

1.3 Gliederung

Die Arbeit gliedert sich in 7 Kapitel. Kapitel 1 beschreibt die Ziele und die Motivation der Arbeit. In Kapitel 2 werden zum Verständnis der Arbeit notwendige Grundlagen über szenariobasierte Spezifikationen, ScenarioTools, Büchi-Spiele und Controller-Synthese erklärt. Kapitel 3 analysiert die aktuellen Fähigkeiten von ScenarioTools und begründet die daraus entstehende Zielsetzung. Kapitel 4 stellt den Hauptteil der Arbeit dar, hier wird der neu entwickelte On-the-Fly-Algorithmus vorgestellt und erläutert. In Kapitel 5 wird ein weiterer umgesetzter Algorithmus vorgestellt, der darüber hinaus um die Funktion der Fairness erweitert wurde. Die Algorithmen werden in Kapitel 6 auf Funktionalität und Performance evaluiert. In Kapitel 7 folgt schließlich das Fazit und ein Ausblick auf mögliche Weiterentwicklungen.

Kapitel 2

Grundlagen

In diesem Kapitel werden für das Verständnis der Arbeit nötige Grundlagen erläutert. Zunächst wird in Abschnitt 2.1 auf den szenariobasierten Entwurf mit ScenarioTools eingegangen. Anschließend werden in Abschnitt 2.2 die von der Synthese genutzten Büchi-Spiele erläutert. In Abschnitt 2.3 wird gezeigt, wie eine ScenarioTools-Spezifikation auf diese Spiele abgebildet werden kann. Abschnitt 2.4 gibt zuletzt einen kurzen Überblick über die eigentliche Synthese.

2.1 Szenariobasierte Spezifikationen

Das Konzept des *szenariobasierten Entwurfs* wurde von Harel et al. [11] beschrieben. Die Grundidee dabei ist, das Verhalten eines *Systems* mit Hilfe von Szenarien zu beschreiben, die die Abläufe verschiedener Komponenten beschreiben. Dabei befindet sich das System in einer *Umgebung*, die mit diesem interagiert. Harel et al. verwendeten dazu eine grafische Notation, sog. *Live Sequence Charts*.

ScenarioTools setzt dieses Konzept um, verwendet aber anders als Harel et al. eine textuelle Notation, um die Szenarien zu spezifizieren, die *Scenario Modeling Language* (SML). Diese erlaubt es, formale Software-Spezifikationen in einem vergleichsweise intuitiven Format zu spezifizieren. Im Vergleich zu anderen Methoden für formale Spezifikationen, wie beispielsweise die Modellierung des Systemverhaltens durch LTL (*Linear Temporal Logic*, entwickelt und zur Programmverifikation genutzt von Pnueli [18]), ist SML für ungeübte Nutzer deutlich aussagekräftiger und einfacher zu benutzen, weshalb sie eine niedrigere Einstiegshürde für den formalen Entwurf darstellt.

Eine SML-Spezifikation besteht aus einer Menge von Szenarien und Objekten. Listing 1 zeigt beispielhaft einen Auszug einer solchen Spezifikation. Jedes Objekt ist entweder *kontrollierbar* oder *unkontrollierbar*. Kontrollierbare Objekte können vom modellierten System direkt kontrolliert

```
1  specification scenario UserBuysSomething {
2    message user -> machine.productSelected()
3    message strict requested machine -> machine.showPrice()
4    message user -> machine.moneyInserted()
5    message strict requested machine -> machine.deliverGood()
6    message strict requested machine -> machine.outputChange()
7  }
8
9  specification scenario UserPaidForGood {
10   message user -> machine.moneyInserted()
11   message strict requested machine -> machine.calculateChange()
12   message strict requested machine -> machine.outputChange()
13 }
```

Listing 1: Ausschnitt einer SML-Spezifikation

werden, während unkontrollierbare Objekte außerhalb der Kontrolle liegen. Ersteres könnte beispielsweise eine Systemkomponente sein, letzteres ein Benutzer, der mit dem System interagiert. Szenarien beschreiben jeweils eine Interaktion, an der beliebige Objekte beteiligt sein können. Diese beteiligten Objekte werden an *Rollen* zugewiesen, welche entweder *statisch* oder *dynamisch* festgelegt sind. Statischen Rollen werden feste Objektbindungen zugewiesen, sodass immer die gleichen Objekte für das Szenario genutzt werden. Dynamische Rollen beziehen sich hingegen nicht auf ein festes Objekt, sondern auf eine bestimmte Klasse von Objekten. Hier kann es mehrere mögliche Objektbindungen geben, falls im modellierten System mehrere Objekte der gleichen Klasse vorkommen. Im System können beliebig viele Szenarien parallel aktiv sein, wodurch sie sich auch gegenseitig beeinflussen können.

Jedes Szenario ist eine Folge von Nachrichten, ggf. mit Kontrollflussdirektiven wie Schleifen oder Bedingungen. Eine Nachricht hat die Form `message sender -> receiver.func()`. Dabei sind `sender` und `receiver` die Absender- bzw. Empfänger-Objekte, und `func` eine Funktion des Empfängers. Nachrichten werden eindeutig über die Kombination von Absender, Empfänger und aufgerufener Funktion identifiziert. Ein Szenario wird *aktiviert*, wenn die erste Nachricht des Szenarios im System auftritt. In diesem Fall wird eine *aktive Kopie* bzw. *Instanz* erstellt, welche die aktuelle Position innerhalb des Szenarios und etwaige dynamische Objektbindungen enkodiert. Nachrichten, die von der aktuellen Position aus als nächstes auftreten können, werden als *enabled* bezeichnet.

Darüber hinaus kann ein konkretes Auftreten von Nachrichten optional als *strict* und *requested* deklariert werden. Sollte eine Nachricht enabled sein, die als *strict* deklariert wurde, dann darf keine der anderen von diesem Szenario verwendeten Nachrichten im System auftreten. Alle Nachrichten, die von diesem Szenario nicht genutzt werden, sind davon jedoch nicht betroffen. Falls die Nachricht nicht als *strict* deklariert ist, so dürfen auch

andere Nachrichten des selben Szenarios auftreten. In diesem Fall wird die Instanz terminiert. Falls dabei die erste Nachricht des Szenarios auftritt, wird dementsprechend eine neue Instanz erzeugt. Ist eine Nachricht als *requested* deklariert, so darf sie nicht unendlich lange enabled sein, sonst ist die Spezifikation verletzt. Um dies zu gewährleisten schreibt die SML-Semantik vor, dass als requested deklarierte Nachrichten kontrollierbarer Objekte zum frühestmöglichen Zeitpunkt gesendet werden. Nachrichten, die nicht als *requested* deklariert sind können jedoch auch unendlich lange enabled sein. Eine Nachricht, die sowohl als *strict* als auch *requested* deklariert ist, muss dementsprechend zwingend ausgeführt werden, bevor andere Nachrichten des selben Szenarios auftreten dürfen. Jede auftretende Nachricht kann potentiell neue Szenarien aktivieren, wenn sie der ersten Nachricht anderer Szenarien entspricht.

Es gibt drei Arten von Szenarien: Specification-Szenarien, Requirement-Szenarien und Assumption-Szenarien. Specification-Szenarien beschreiben das Verhalten einzelner Aspekte des Systems. In Listing 1 sind zwei solcher Szenarien enthalten, die den Bestellprozess eines Automaten modellieren. Das Szenario `UserBuysSomething` wird aktiviert, wenn die Umgebung (in diesem Fall der Benutzer) die Nachricht `productSelected` an das Objekt `machine` schickt, also wenn der Benutzer eine Ware ausgewählt hat, die er bestellen möchte. Als Reaktion darauf muss `machine` nun zwingend den Preis der gewählten Ware anzeigen. Anschließend wird auf die Nachricht `moneyInserted` gewartet. Da diese weder *strict* noch *requested* ist, muss sie nicht zwangsläufig kommen. Der Benutzer könnte beispielsweise auch ein anderes Produkt auswählen, also nochmals die Nachricht `productSelected` senden. In diesem Fall würde das Szenario abgebrochen und neu gestartet werden. Die Nachricht `moneyInserted` aktiviert das zweite Szenario, `UserPaidForGood`. Dieses soll den Bezahlprozess modellieren. Nun sind zwei Nachrichten im System enabled: `deliverGood` und `calculateChange`. Im konkreten Beispiel ist die Reihenfolge, in der die beiden gesendet werden, nicht relevant. Nach dem Senden einer der beiden Nachrichten ist die Nachricht `outputChange` enabled. Diese wird allerdings vom anderen Szenario blockiert, da die verbleibende der beiden Nachrichten als `strict` deklariert ist. Dies erzwingt, dass die Nachricht `outputChange` als letztes gesendet wird, und damit beide Szenarien gleichzeitig abschließt.

Requirement-Szenarien sind sekundäre Bedingungen. Hiermit können Bedingungen ausgedrückt werden, die zwar irgendwann erfüllt sein müssen, jedoch nicht mit Priorität verfolgt werden. Der requested-Modifier hat hier eine andere Bedeutung: Er drückt nicht mehr aus, dass die Nachricht bei der nächstmöglichen Gelegenheit auftreten muss, sondern, dass sie irgendwann in der Zukunft auftreten muss, die Nachricht also nur endlich lange enabled sein darf. Beispielsweise könnte gelten, dass eine Sicherheitstür nach dem Öffnen wieder geschlossen werden muss. Dies muss allerdings nicht bei der erstmöglichen Gelegenheit passieren, da sonst niemand die Tür passieren

könnte. Falls es möglich ist, dass die Tür nie wieder geschlossen wird, so wäre das Requirement-Szenario verletzt und das System verliert.

Assumption-Szenarien können genutzt werden, um das Verhalten der Umgebung zu modellieren. Dies könnte etwa genutzt werden, um die Handlungsmöglichkeiten in bestimmten Situationen einzuschränken, wo manche Optionen logisch ausgeschlossen sind. Auf diese Optionen muss das System dann nicht reagieren können. Das Gesamt-Systemverhalten ergibt sich aus der Synthese aller Szenarien.

ScenarioTools implementiert den Play-Out-Algorithmus von Harel et. al. [11]. Hierbei wartet das System zunächst auf Nachrichten von der (unkontrollierbaren) Umgebung. Anschließend versucht das System, einen Zustand zu erreichen, in dem wieder auf Umgebungsnachrichten gewartet wird und keine Nachrichten enabled sind, die als requested deklariert wurden. Wie in Listing 1 kann es hierbei mehrere mögliche Reihenfolgen geben, in der die Nachrichten ausgeführt werden. Falls alle möglichen Nachrichten durch andere Szenarien blockiert werden liegt ein Deadlock bzw. eine *Safety Violation* vor. In diesem Fall ist das System handlungsunfähig, weshalb er unter keinen Umständen eintreten darf. Je nach gewählter Ausführungsreihenfolge von Nachrichten kann eine Safety-Violation auftreten oder vermieden werden, weshalb es notwendig ist, eine *Strategie* für das System zu berechnen. In ScenarioTools werden dazu die Spezifikation auf *Büchi-Spiele* abgebildet, mit denen schließlich versucht wird, eine Strategie zu berechnen, um unabhängig von den Handlungen der unkontrollierbaren Objekte die Spezifikation zu erfüllen. Dieses Vorgehen wird als *Controller-Synthese* bezeichnet. Falls eine solche Strategie existiert, ist formal bewiesen, dass die Spezifikation ohne Violations implementiert werden kann.

2.2 Büchi-Spiele

Ein Büchi-Spiel ist ein *unendliches* Spiel zwischen zwei Spielern. Dies bedeutet, dass es unendlich viele Züge beider Spieler gibt. Eine solche unendliche Folge von Zügen wird als *Play* bezeichnet. Büchi-Spiele werden durch spezielle Graphen repräsentiert.

Ein gerichteter Graph $G = (V, E)$ besteht aus einer Menge V von Zuständen und einer Menge E von Transitionen, wobei $E \subseteq V \times V$. Es seien $n = |V|$ und $m = |E|$. Für eine Transition $t = (u, v) \in E$ sei $source(t) = u$ der Startzustand und $target(t) = v$ der Zielzustand. Ein *Pfad* ist eine Folge $v_1, ..., v_k$ von Zuständen mit $(v_i, v_{i+1}) \in E$ für $1 \leq i < k$. Für einen Pfad p sei $start(p) = v_1$ und $end(p) = v_k$. Ein *einfacher Pfad* ist ein Pfad, auf dem kein Zustand doppelt vorkommt. Ein *Zyklus* ist ein Pfad mit $v_1 = v_k$. Ein *einfacher Zyklus* ist ein Zyklus, bei dem $v_1, ..., v_{k-1}$ ein einfacher Pfad ist.

Büchi-Spiele finden auf einem Spielgraph $G = (V, E, (V_1, V_2))$ statt. Für den ersten Spieler ist eine Siegbedingung gegeben, die durch eine Menge

akzeptierender Zielzustände $U \subseteq V$ ausgedrückt wird. Spieler 1 wird im Folgenden als *System* bezeichnet und Spieler 2 als *Umgebung*. Ein Spielgraph ist ein gerichteter Graph, in dem die Zustandsmenge V in zwei Mengen V_1 und V_2 partitioniert ist, die jeweils von einem Spieler kontrolliert werden. Die genommene ausgehende Transition eines Zustands wird jeweils von dem kontrollierenden Spieler gewählt. Soweit nicht anders angegeben seien *kontrollierbare* Zustände bzw. deren ausgehende Transitionen dem System zugeordnet und *unkontrollierbare* der Umgebung. Das System gewinnt, wenn es unabhängig von den Entscheidungen der Umgebung unendlich oft einen akzeptierenden Zustand aus U erreichen kann. Dies wird als *Büchi-Bedingung* bezeichnet [2] und kann alternativ dadurch ausgedrückt werden, dass in jedem möglichen Play auf dem Spielgraph mindestens ein Zielzustand unendlich oft vorkommt. Die komplementäre *co-Büchi-Bedingung* bildet die Siegbedingung der Umgebung. Diese gewinnt, falls nur Zustände aus $V \setminus U$ unendlich oft besucht werden.

Mit Büchi-Spielen können Spezifikationen *offener* Systeme ausgedrückt werden [1]. Dies sind Systeme, deren Verhalten nicht allein von ihrem Zustand abhängt, sondern auch durch Außeneinwirkung von der Umgebung beeinflusst werden kann. Letzteres ist bei einem *geschlossenen* System nicht der Fall. Geschlossene Systeme können durch Büchi-*Automaten* ausgedrückt werden. Im Gegensatz zu Büchi-Spielen gibt es hier keinen Gegenspieler, und somit sind alle Zustände kontrollierbar. Um die Büchi-Bedingung zu erfüllen reicht es deshalb, einen Zyklus zu finden, der mindestens einen Zielzustand enthält.

Ein *verallgemeinertes Büchi-Spiel* (*Generalized Büchi Game*, im folgenden *GBG*) der Form $\bigwedge_{i=1}^{k} g_i$ hat statt einer einzelnen Büchi-Bedingung eine beliebige Anzahl von Büchi-Bedingungen g_i, deren Konjunktion erfüllt sein muss. Es muss also mindestens ein Zustand aus jeder der Büchi-Bedingungen unendlich oft besucht werden. Eine Umwandlung in ein normales Büchi-Spiel ist möglich, jedoch steigt dabei die Anzahl der Zustände um den Faktor k an [5]. Dazu werden k Kopien G_i des Graphen G erstellt. Alle ausgehenden Transitionen der Zielzustände von g_i in G_i werden auf den jeweiligen Folgezustand in G_{i+1} für $i < k$ und G_1 für $i = k$ umgeleitet. Zielzustände dieses Spiels sind die Zustände von *einem* g_i in G_i.

Ein *Assumption-Guarantee-Spiel* ist ein erweitertes GBG der Form $\bigwedge_{i=1}^{k_1} a_i \rightarrow \bigwedge_{i=1}^{k_2} g_i$. Die Menge a_i von Siegbedingungen in der Antezedens impliziert die Menge g_i von Siebedingungen in der Konsequenz. Solange die Antezedens gilt, muss auch die Konsequenz gelten. Gilt die Antezedens jedoch nicht, dann muss auch die Konsequenz nicht gelten. Diese Spiele werden auch als GR(1)-Spiele bezeichnet, *Generalized Reactivity(1)*. Sie sind beliebt, weil sie trotz ihrer Ausdrucksstärke relativ effizient implementierbar sind [17].

In Abbildungen werden im Folgenden kontrollierbare Transitionen als durchgezogene Linien dargestellt und unkontrollierbare Transitionen als

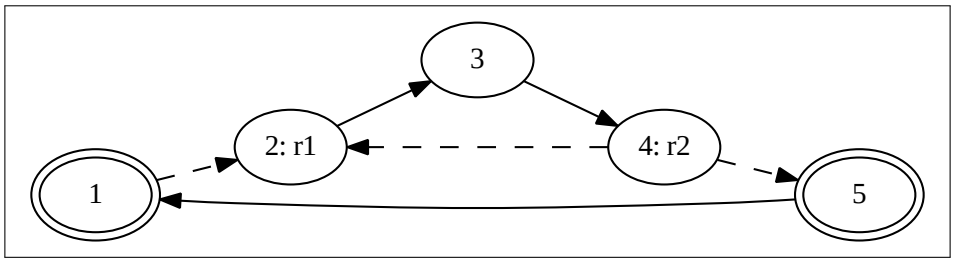

Abbildung 2.1: Beispiel für die genutzte Notation in Graphen

gestrichelte Linien. Zielzustände der ersten Büchi-Bedingung werden doppelt umrandet. Zusätzliche Büchi-Bedingungen sind benannt, und werden im Zustandsnamen nach einem Doppelpunkt aufgeführt. Der Zustand mit der Bezeichnung 1 sei der Startzustand. Dies ist beispielhaft in Abbildung 2.1 zu sehen. Die Zustände 1 und 5 sind Zielzustände der ersten Büchi-Bendingung. Darüber hinaus gibt es zwei weitere Büchi-Bedingungen: $r1$ und $r2$. Sie sind jeweils in Zustand 2 bzw. 4 erfüllt. Die Zustände 1 und 4 sind unkontrollierbar, da die ausgehenden Transitionen gestrichelt dargestellt sind. Alle anderen Zustände sind kontrollierbar. In diesem Beispiel verliert darüber hinaus das System, da die Umgebung in Zustand 4 den Zyklus $(4 \rightarrow 2 \rightarrow 3 \rightarrow 4)$ erzwingen kann, auf dem kein Zielzustand der ersten Büchi-Bedingung vorkommt.

2.3 Abbildung von SML-Spezifikationen auf Spiele

SML-Spezifikationen können wie erwähnt auf Büchi-Spiele abgebildet werden. Dabei enkodiert jeder Zustand die aktiven Szenarien sowie Positionen innerhalb der Szenarien. Die aktuelle Position innerhalb der Szenarien bestimmt die möglichen Nachrichten, die von dieser Konstellation aus auftreten können. Jede dieser Nachrichten wird als eine von dem Zustand ausgehende Transition repräsentiert. Dazu wird zunächst die Nachricht auf die Szenarien-Menge des Zustands ausgeführt, was eine modifizierte Menge mit eventuell neuen oder jetzt abgeschlossenen Szenarien ergibt. Diese entspricht wieder einem Zustand, der bei Bedarf erstellt wird. Anschließend werden die beiden Zustände verbunden. Handelt es sich um eine Nachricht, die vom einem kontrollierbaren Objekt ausging, so gehört die Transition dem System. Ansonsten wird sie der Umgebung zugewiesen.

Eine Spezifikation, in der nur Specification-Szenarien vorkommen, kann durch ein Büchi-Spiel dargestellt werden. Die Menge der Zielzustände sind all jene Zustände, in denen keine Nachrichten mit Requested-Modifier enabled sind und auf Events der Umgebung gewartet wird (diese Zustände entsprechen dem Ziel des Play-Out-Algorithmus). Eine Spezifikation, die zusätzlich Requirement-Szenarien enthält, kann durch ein GBG gelöst werden.

Dazu wird jede Instanz eines Requirement-Szenarios in eine Siegbedingung überführt. Die generalisierte Siegbedingung des GBGs entspricht dann der Konjunktion dieser Siegbedingungen und der Specification-Szenarien. Enthält eine Spezifikation Assumption-Szenarien, dann lässt sich die Spezifikation nur durch ein GR(1)-Spiel lösen. Die Instanzen der Assumption-Szenarien werden hier auf Siegbedingungen der Antezedens überführt, während die Konsequenz durch die Specification- bzw. etwaige Requirement-Szenarien bestimmt wird. Der Controller ergibt sich aus der Strategie des Systems, um das jeweilige Spiel zu gewinnen.

Die Zielzustände der Requirement- und Assumption-Szenarien sind all jene Zustände, in denen in der jeweiligen Instanz keine Nachrichten mit requested-Modifier enabled sind. Dies schließt auch Zustände ein, in denen die Instanz gar nicht vorkommt. Die Menge der Zielzustände entspricht also dem Komplement der Zustände, in denen Nachrichten mit requested-Modifier enabled sind. Diese Menge ist im Allgemeinen bedeutend größer, da insbesondere Requirement-Szenarien oft nur in Teilen des Zustandsraums vorkommen. Es kann daher in der Implementierung vorteilhaft sein, nur das Komplement der Zielzustände zu betrachten, d.h. die verlierenden Zustände. Dies bietet sich insbesondere während On-The-Fly-Algorithmen an, da hier unter Umständen nur ein Teil des Zustandsraums erstellt wird.

Anmerkung. Die Instanzen eines Szenarios werden an dieser Stelle nur über die Objektbindungen identifiziert, und eine einzelne Instanz kann somit auch mehrmals an verschiedenen Stellen des Spielgraphen erstellt und abgeschlossen werden.

2.4 Controller-Synthese

Controller-Synthese ist der Prozess, aus einer formalen Spezifikation einen Controller zu generieren, der die Einhaltung der Spezifikation garantieren kann. Falls die Spezifikation widersprüchlich oder nicht implementierbar ist, so ist es Aufgabe der Synthese, ein Gegenbeispiel anzugeben, mit der die Spezifikation verletzt wird. Dazu wird die Spezifikation generell in einem Spielgraphen bzw. Automaten übersetzt, welcher anschließend auf Erfüllbarkeit untersucht wird. Falls die Spezifikation umsetzbar ist, existiert eine *Strategie* für das System, um die sich aus der Spezifikation ergebenden Siegbedingungen zu erfüllen.

Eine Strategie für einen Spieler s ist eine Funktion $V \mapsto 2^V \setminus \emptyset$, mit der allen von s kontrollierten Zuständen Nachfolgerzustände zugewiesen werden. Die Strategie ist gewinnend, falls mit ihr die Büchi- bzw. co-Büchi-Bedingung erfüllt wird. Von einem festen Startzustand aus existiert nur für genau einen Spieler eine gewinnende Strategie [15].

Eine Strategie, die nur von dem aktuellen Zustand abhängt, heißt *gedächtnislos*. Strategien *mit* Gedächtnis können bei der Entscheidung eines

Folgezustands auch zuvor besuchte Zustände einbeziehen. In Büchi-Spielen existiert immer eine gedächtnislose Strategie, jedoch gibt es GBGs, in denen es für das System nur eine Strategie mit Gedächtnis gibt [24]. Da ein GBG der Form $\bigwedge_{i=1}^{k} g_i$ durch das GR(1)-Spiel $true \rightarrow \bigwedge_{i=1}^{k} g_i$ gelöst werden kann, folgt daraus direkt, dass auch Strategien für GR(1)-Spiele unter Umständen ein Gedächtnis benötigen.

Es gibt eine Vielzahl von Algorithmen, die zur Berechnung einer Strategie auf einen Spielgraphen genutzt werden können. Sie lassen sich in zwei Gruppen einteilen: Algorithmen, die *on the fly* arbeiten, und solche, die es nicht tun. On-the-fly-Algorithmen (OTF-Algorithmen) konstruieren den Zustandsraum während der Suche, und müssen so unter Umständen nicht den gesamten Zustandsraum konstruieren. Da die Übersetzung einer Spezifikation generell zu einem exponentiellem Zuwachs führt [19], kann auf diese Art viel Zeit eingespart werden.

Kapitel 3

ScenarioTools-Analyse

In diesem Kapitel werden die Fähigkeiten der ScenarioTools-Synthese analysiert. In Abschnitt 3.1 werden zunächst die bestehenden von ScenarioTools implementierten Verfahren betrachtet und ihre Stärken und Schwächen aufgelistet. Anschließend wird in Abschnitt 3.2 basierend auf der Auswertung dieser Algorithmen die Zielsetzung der Arbeit erläutert. In Abschnitt 3.3 wird auf verwandte Arbeiten eingegangen.

3.1 Bestehende Synthese-Algorithmen

Momentan implementiert ScenarioTools zwei verschiedene Synthese-Algorithmen: Ein On-the-Fly-Algorithmus angelehnt an das Verfahren von Tripakis und Altisen [23] und ein von Greenyer entwickelter Algorithmus, `DFSBased`. Beide haben unterschiedliche Stärken und Schwächen, jedoch ist keiner ausreichend, um alle Spezifikationen vollends zu realisieren.

On the fly Algorithmus

Der OTF-Algorithmus implementiert normale Büchi-Spiele. Er geht dabei wie folgt vor: Zunächst wird vom Startzustand aus eine Reachability-Suche ausgeführt. Dabei werden mögliche Pfade in einer Tiefensuche erkundet. Falls auf diese Art ein Zielzustand gefunden wird, so wird versucht, diesen

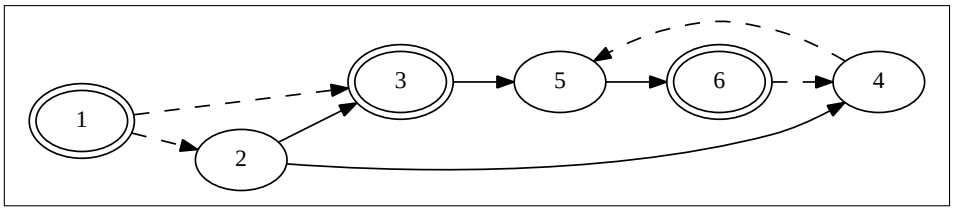

Abbildung 3.1: Beispiel für einen Spielgraphen, bei dem der OTF-Algorithmus erfolgreich ist

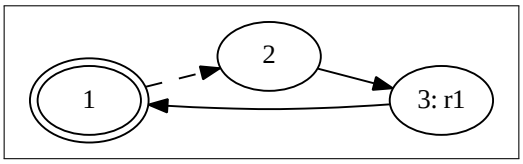

Abbildung 3.2: Beispiel für einen Spielgraphen, bei dem der OTF-Algorithmus scheitert

über alle unkontrollierbaren Transitionen zurück zum Startzustand der Reachability-Suche zu propagieren. Bei einem Zustand, der mehrere ausgehende unkontrollierbare Transitionen hat, müssen so über alle Transitionen hinweg Zielzustände gefunden werden. Gelingt es, diese Propagierung bis zum Startzustand fortzusetzen so ist dieser tendenziell gewinnend. Die Reachability-Suche wird für alle auf diese Art erreichten Zielzustände wiederholt. Falls sie für einen Zielzustand fehlschlägt, so wird dieser als verlierend markiert und von ihm abhängende Zustände neu evaluiert. Das System gewinnt, falls eine Menge *goal* von Zielzuständen gefunden werden kann, die ausgehend vom Startzustand immer das Erreichen eines Zustands aus *goal* garantieren kann.

Mit diesem Vorgehen kann der Algorithmus etwa für Abbildung 3.1 eine Siegstrategie für das System erzeugen. Zustand 1 kann über die Pfade $(1 \rightarrow 3)$ und $(1 \rightarrow 2 \rightarrow 3)$ das Erreichen von Zielzuständen garantieren. Zustand 3 wiederum erreicht über den Pfad $(3 \rightarrow 5 \rightarrow 6)$ einen Zielzustand. Für diesen ist der einzig mögliche Pfad $(6 \rightarrow 4 \rightarrow 5 \rightarrow 6)$, wodurch jeder Zielzustand wieder einen Zielzustand erreichen kann.

Da Büchi-Spiele nur eine Siegbedingung unterstützen, kombiniert er die Specification- und Requirement-Szenarien. Die akzeptierten Zielzustände entsprechen der Schnittmenge aller Zielzustandsmengen. Da diese Mengen jedoch durchaus disjunkt sein können, kann der Algorithmus in vielen Fällen keine gültige Strategie erzeugen, wenn Requirement-Szenarien verwendet werden. Assumption-Szenarien werden nur sehr begrenzt unterstützt. Sie werden zur Erzeugung des Zustandsraumes genutzt und Violations werden erkannt, jedoch wird in keiner Weise auf nicht akzeptierende Assumption-Zyklen geprüft. Abbildung 3.2 zeigt ein simples Beispiel, bei dem der OTF-Algorithmus versagt. Zustand 1 ist Zielzustand für die Specification-Szenarien, Zustand 3 für das Requirement-Szenario $r1$. Der Zyklus $(1 \rightarrow 2 \rightarrow 3 \rightarrow 1)$ wäre somit eine gewinnende Strategie für das System. Jedoch gibt es keinen Zustand, in dem sowohl die Specification-Szenarien als auch $r1$ erfüllt sind, weshalb die Suche nach einem Zielzustand erfolglos bleibt und, im Kontext von ScenarioTools, ein False-Negative zurückgeliefert wird.

Die Vorteile des OTF-Algorithmus liegen darin, dass er nicht zwangsläufig den kompletten Zustandsraum generieren muss. Falls es einen Teilgraphen gibt, für den eine gültige Strategie generiert werden kann, dann muss der Rest

des Zustandsraum unter Umständen gar nicht betrachtet werden. Dadurch kann besonders bei großen Zustandsräumen sehr viel Zeit eingespart werden, weshalb der Algorithmus trotz der Einschränkungen weiterhin für passende Anwendungszwecke relevant bleibt. Falls es keine Lösung gibt muss jedoch in den meisten Fällen der komplette Zustandsraum erkundet werden, etwa bei Fällen wie in Abbildung 3.2, wo der Algorithmus überhaupt keinen Zustand als gewinnend ansieht.

DFSBased-Algorithmus

Der DFSBased-Algorithmus wurde konzipiert, um die beschriebenen Schwächen des OTF-Algorithmus zu vermeiden. Er arbeitet ebenfalls On-The-Fly, muss jedoch den gesamten Zustandsraum erkunden, um eine positive Antwort zu geben. Es kann jedoch frühzeitig erkannt werden, dass keine Strategie existieren kann. In solchen Fällen muss nur ein Teil des Zustandsraums aufgebaut werden. Der Algorithmus implementiert GR(1)-Spiele.

Die Grundidee besteht darin, die Zyklen im Spielgraphen auf Akzeptanz zu prüfen und diesen Zustand zu propagieren. Der Algorithmus führt zunächst vom Startzustand eine Tiefensuche aus. Von jedem Zustand, der erstmals besucht wird, wird ebenfalls eine Tiefensuche initiiert. Der Algorithmus merkt sich dabei den Pfad, der vom initiierenden Zustand aus aktuell genommen wurde. Wird einer der Zustände auf diesem Pfad wieder gefunden, so existiert ein Zyklus. Dieser Zyklus wird analysiert, es wird geprüft ob er die Büchi-Bedingungen erfüllt oder die Assumptions verletzt. Ist der Zyklus verlierend, so wird ausgehend vom letzten nichtwiederholtem Zustand aus dieser Status an die Vorgänger propagiert. Eine Strategie existiert, falls nach Abschluss der Suche der Startzustand nicht als verlierend markiert wurde. Die Strategie ergibt sich, indem jeweils den nicht als verlierend gekennzeichneten Zuständen gefolgt wird.

Auf diese Weise kann der DFSBased-Algorithmus für das Beispiel in Abbildung 3.2 eine gültige Strategie für das System finden. Die Tiefensuche wird vom Startzustand 1 aus initiiert, wo nach einigen Schritten der einzige Zyklus im Graphen gefunden wird. Dieser Zyklus enthält für jede Siegbedingung einen Zielzustand, weshalb kein Zustand als verlierend markiert wird und das System gewinnt.

Der Algorithmus ist jedoch nicht vollständig. Abb. 3.3 zeigt einen Spielgraphen, bei dem eine existierende Stragie nicht gefunden werden kann. Hier gibt es die Specification-Zielzustände 1 und 4, während der einzige Zielzustand für das Requirement-Szenario $r1$ der Zustand 5 ist. In diesem Graph gibt es zwei mögliche Zyklen, abhängig von den Entscheidungen des Systems in Zustand 2:

1. Einen gewinnenden Zyklus, $(1 \to 2 \to 4 \to 5 \to 3 \to 1)$
2. Einen verlierenden Zyklus, $(1 \to 2 \to 3 \to 1)$

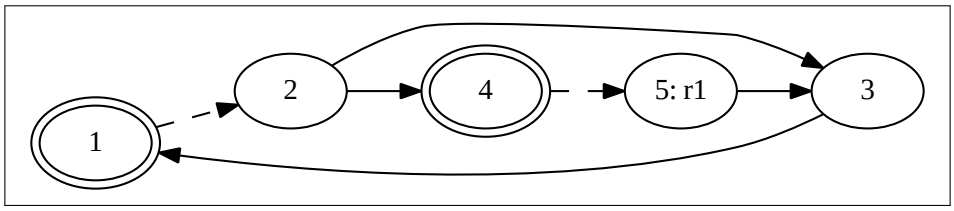

Abbildung 3.3: Beispiel für einen Spielgraphen, bei dem der DFSBased-Algorithmus scheitert

Das Problem ist hier, dass die Zyklen das selbe Suffix ($3 \rightarrow 1$) haben. Wenn der verlierende Zyklus betrachtet wird, wird ausgehend von Zustand 3 der Zyklus als verlierend markiert. Jeder Vorgänger, der keine alternativen vom System kontrollierbaren Nachfolger hat, wird ebenfalls als verlierend markiert. Dies schließt also ebenfalls Zustand 5 und dessen Vorgänger mit ein. So wird insgesamt der komplette Graph als verlierend betrachtet, obwohl es eine gewinnende Strategie gibt.

Darüber hinaus wurde für den DFSBased-Algorithmus *Partial Order Reduction* implementiert, was eine Taktik ist, um die Größe des Zustandsraums zu reduzieren. Nach Clarke et al. [6] wird dabei die Unabhängigkeit von parallel auftretenden Nachrichten ausgenutzt, die in jeder möglichen Ausführungsreihenfolge zum selben Systemzustand führen. Dies bedeutet insbesondere, dass keine Ausführungsreihenfolge zu einem Deadlock führt. In Abbildung 3.4 ist ein Beispiel abgebildet, bei dem Partial Order Reduction anwendbar ist. Hier gibt es $n = 3$ mögliche Nachrichten, a, b und c, die jeweils an den Transitionen notiert sind. Die Zustandsbezeichner geben an, welche der Nachrichten bisher ausgeführt wurden. Es gibt in solch einer Situation 2^n Zustände und $n!$ mögliche Ausführungsreihenfolgen. Im konkreten Beispiel sind dies also 8 Zustände und 6 Ausführungsreihenfolgen. Ohne Partial-Order-Reduction müssten auch potentiell alle Reihenfolge betrachtet werden, was bei einer Vielzahl unabhängiger Nachrichten sehr aufwändig sein kann. Da sie jedoch alle zum selben Systemzustand führen, wäre es ausreichend, nur *eine* repräsentative Reihenfolge mit $n + 1$ Zuständen zu betrachten. Dies ist das Ziel der Partial-Order-Reduction. Jedoch ist es schwierig, die Unabhängigkeit von Nachrichten automatisiert zu erkennen, sodass unter Umständen für ein korrektes Ergebnis Expertenwissen notwendig ist. Dies ist etwa dadurch möglich, dass ein Experte einzelne Nachrichten als voneinander unabhängig kennzeichnet. Da dies jedoch ein potentiell fehleranfälliger Prozess ist, werden in ScenarioTools notwendige Eigenschaften der Nachrichten weiterhin geprüft.

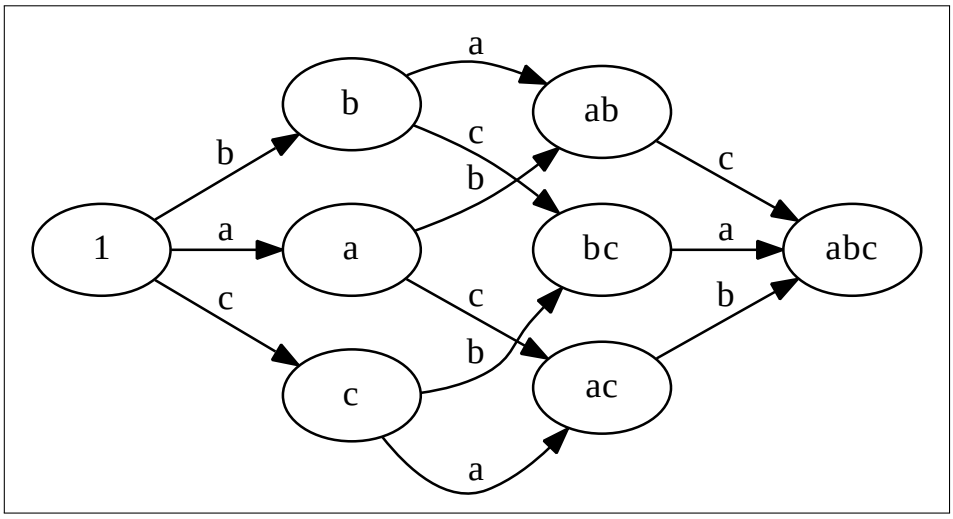

Abbildung 3.4: Beispiel für parallele unabhängige Nachrichten, angelehnt an Beispiel von Clarke et al. [6]

3.2 Zielsetzung

Keiner der beiden Algorithmen kann alle Spezifikationen korrekt bearbeiten. Der DFSBased-Algorithmus ist jedoch der bei weiterem mächtigere, er kann fast alle Fälle behandeln, die der OTF-Algorithmus akzeptiert, und darüber hinaus noch die meisten Spezifikationen, die ein GBG- oder GR(1)-Spiel benötigen. Die Vorzüge des OTF-Algorithmus sind einige wenige Fälle, in denen der DFSBased-Algorithmus versagt und es eine ausreichende Überlappung der Specification- und Requirement-Zielzustände gibt. Jedoch hat der DFSBased-Algorithmus für erfüllbare Spezifikationen oft eine teils deutlich höhere Laufzeit, da der komplette Zustandsraum erstellt werden muss. Es wäre deshalb wünschenswert, einen Algorithmus zu haben, der mehr Spezifikationen verarbeiten kann als der bestehende OTF-Algorithmus, ohne jedoch wie im DFSBased-Algorithmus notwendigerweise den kompletten Zustandsraum zu erstellen. Ein GBG- oder GR(1)-Algorithmus, der on-the-fly arbeitet, könnte bei vielen Spezifikationen eine hohe Zeitersparnis bewirken. Wenn dieser zusätzlich noch vollständig ist, könnten darüber hinaus bisher nicht implementierbare Spezifikationen verarbeitet werden.

Die bestehenden Algorithmen erzeugen beide Strategien, die ohne Gedächtnis auskommen. Bei einer gedächtnislosen Strategie können in der Vergangenheit besuchte Zustände keine Auswirkung auf den gewählten Nachfolgerzustand haben, wodurch jeder Zustand nur maximal einen, festgelegten kontrollierbaren Nachfolger hat. Abbildung 3.5 zeigt einen Graphen, bei dem keine gedächtnislose Strategie existiert. Dort müsste zwischen beiden Nachfolgern von Zustand 2 alterniert werden, um die Requirements $r1$ bzw.

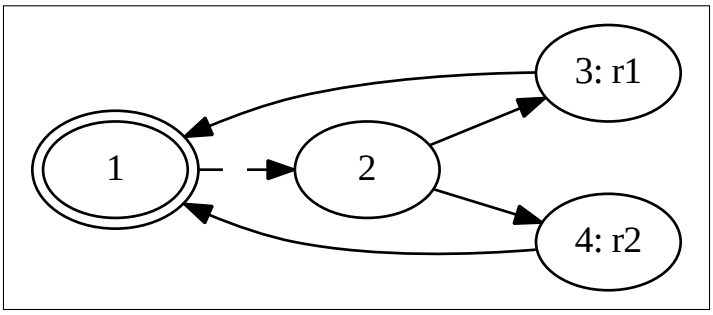

Abbildung 3.5: Beispiel für eine Spezifikation, die ein Gedächtnis benötigt

$r2$ unendlich oft zu erfüllen.

Das Ziel ist deshalb, zunächst einen mächtigeren OTF-Algorithmus zu entwickeln, der mindestens Spezifikationen umsetzen kann, die GBGs wie in Beispiel 3.2 ergeben. Weiterhin soll als Alternative und Referenz ein Algorithmus umgesetzt werden, der vollständig für GR(1)-Spiele ist, also unter anderem auch Strategien mit Gedächtnis erzeugen kann, wie sie für Beispiel 3.5 nötig sind. Hier gibt es bestehende Algorithmen, die für ScenarioTools angepasst werden können. Sie arbeiten zwar nicht on-the-fly, jedoch stellt jede Form von vollständigen Algorithmen eine lohnenswerte Erweiterung der Fähigkeiten von ScenarioTools dar.

3.3 Verwandte Arbeiten

Da die Synthese szenariobasierter Spezifikationen durch Abbildung auf Büchi bzw. GR(1)-Spiele gelöst wird, sind prinzipiell alle Forschungsfortschritte dieser Gebiete relevant. Neben den vorgestellten Spieltypen gibt es noch weitere, mächtigere unendliche Spiele, mit denen komplexere Siegbedingungen ausgedrückt werden können. Über diese soll hier zunächst ein kurzer Überblick gegeben werden. Thomas et al. [22] fassen die gebräuchlichsten Spieltypen zusammen. Eine *Rabin-Bedingung* wird etwa durch eine Menge von Paaren $P = \{(e_1, f_1), ..., (e_k, f_k)\}$ ausgedrückt, wobei jeweils $e_i \subseteq V$ und $f_i \subseteq V$. Sie ist erfüllt, wenn ein i mit $1 \leq i \leq k$ existiert, für das alle Zustände in e_i nur endlich oft besucht werden, während mindestens ein Zustand aus f_i unendlich oft besucht wird [4]. *Streett-Bedingungen* sind komplementär zu Rabin-Bedingungen. Sie sind erfüllt falls für jedes $1 \leq i \leq k$ gilt, dass Zustände aus f_i nur dann unendlich besucht werden, wenn auch mindestens ein Zustand aus e_i unendlich oft besucht wird. *Parity-Bedingungen* sind Bedingungen, für die $e_1 \subset f_1 \subset e_2 \subset ... \subset e_k \subset f_k$ gilt. Sie sind sowohl Rabin- als auch Streett-Bedingungen. Rabin-Bedingungen finden unter anderem Anwendung bei der Determinisierung von nicht-deterministischen Büchi-Automaten [20]. Die Algorithmen zur Lösung dieser Spiele weisen eine höhere Laufzeitkomplexität als Algorithmen zur Lösung von GR(1)-Spielen auf. Für

SML-Spezifikation ist die erhöhte Ausdrucksstärke jedoch momentan nicht erforderlich.

Ehlers [8] beschreibt ein Verfahren, mit dem die Synthese *robuster* Systeme effizient umgesetzt werden kann. Die Robustheit eines Systems drückt sich durch sein Verhalten bei Verletzung der Assumptions aus. Die dort vorgestellte *Generalized Rabin(1)*-Synthese ist eine Erweiterung der GR(1)-Synthese, mit der auch Rabin-Bedingungen mit einem Paar spezifiziert werden können. Mit diesen können auch co-Büchi-Assumption-Guarantee-Bedingungen ausgedrückt werden. Auf diese Art kann Verhalten spezifiziert werden, das auch bei Verletzung der Assumptions noch gelten muss. Dies ist ein typisches Verhalten manuell konstruierter Systeme, bei denen generell versucht wird, zumindest einen Teil der Garantien in jedem Fall zu erfüllen. Für Systeme, die durch GR(1)-Synthese entstanden sind, trifft dies im Allgemeinen nicht zu. Die Synthese wird auf ein Parity-Spiel mit fester Anzahl $k = 5$ von Paaren abgebildet. Weiterhin wird gezeigt, dass für keine weitere Erweiterung der Ausdrucksstärke (die eine *Generalized Streett(1)*-Synthese benötigen würde) ein effizienter Algorithmus existiert falls $\mathrm{P} \neq \mathrm{NP}$.

Ein weiterer interessanter Ansatz ist die *kompositionelle* Synthese von Systemen. Hier wird die Synthese nicht für das ganze System auf einmal durchgeführt, sondern sie wird ggf. für Teile des Systems separat oder im Voraus durchgeführt. De Giacomo et al. [7] nutzen so einen Ansatz, um ein System aus schon bestehenden Komponenten zusammenzusetzen, und einen Controller für das Gesamtsystem zu erstellen. Kugler et al. [14] stellen ein Verfahren vor, bei dem zunächst eine Teilmenge einer szenariobasierten Spezifikation synthetisiert und als Eingabe für eine Obermenge genutzt werden kann. Dies gelingt nicht immer, führte jedoch in einigen Fällen zu einer effizienteren Synthese. Greenyer und Kindler [10] beschreiben ein Verfahren, mit dem bestimmte szenariobasierte Spezifikationen in zwei Teilprobleme aufgeteilt und so effizienter gelöst werden können.

Neben den in dieser Arbeit beschriebenen Ansätzen ist ebenfalls die Synthese mit *symbolischer* Berechnung beliebt. Auf diese Art konnte etwa die benötigte Rechenzeit für viele LTL-Spezifikationen deutlich reduziert werden. Piterman et al. [16] beschreiben ein Verfahren zur symbolischen GR(1)-Synthese. Wie die Evaluation in Kapitel 6 jedoch zeigt, würde ScenarioTools nur bedingt dadurch profitieren, da die überwiegende Mehrheit der Rechenzeit für dem Aufbau des Zustandsraums aufgebracht wird.

Die Forschung von OTF-Methoden beschränkte sich bisher größtenteils auf geschlossene Systeme, und somit auf verschiedene Ausprägungen von Büchi-Automaten. Tauriainen [21] beschreibt etwa ein Verfahren, mit dem für Generalized-Büchi-Automaten eine Strategie on-the-fly berechnet werden kann. Neben dem Algorithmus von Tripakis et al. [23] für Büchi-Spiele konnten keine OTF-Algorithmen für komplexere Spiele gefunden werden.

Kapitel 4

Generalized On-the-Fly-Algorithmus

Je nach Anwendung kann das Erstellen des Zustandsraums sehr aufwändig sein. In ScenarioTools ist das Erstellen eines neuen Zustands im Büchi-Spiel beispielsweise die mit Abstand teuerste Operation, da hierfür aufwändige Simulationen auf Basis des Vorgängerzustands ausgeführt werden müssen. Ein Algorithmus, der nicht den kompletten Zustandsraum erzeugen muss, also on-the-fly arbeitet, kann daher einen großen Geschwindigkeitszuwachs bedeuten, selbst wenn die algorithmische Komplexität im Worst-Case bedeutend höher liegt, als bei Verfahren die den gesamten Zustandsraum betrachten.

In diesem Kapitel wird ein solcher OTF-Algorithmus für GBGs vorgestellt. Die grundlegende Idee ist, die Siegbedingungen in eine primäre und eine beliebige Anzahl sekundäre Siegbedingungen aufzuteilen. Zunächst wird eine Lösung für die primäre Siegbedingung gesucht, und anschließend verifiziert, ob die sekundären Siegbedingungen erfüllt sind. Sind sie es nicht, werden Anpassungen vorgenommen und der Prozess wiederholt. Er war zunächst als reiner GBG-Algorithmus konzipiert, wurde jedoch später um Aspekte von GR(1)-Spielen erweitert. Diese werden jedoch nicht vollständig unterstützt, da die primäre Siegbedingung in jedem Fall die Büchi-Bedingung erfüllen muss. Für sekundäre Siegbedingungen ist auch ein Sieg im Sinne eines Assumption-Guarantee-Spiels möglich. Weiterhin ist der Algorithmus für Strategien *ohne* Gedächtnis konzipiert. Diese sind zwar für GBGs weniger mächtig, als Strategien mit Gedächtnis, jedoch ist auf diese Weise die Extraktion der Strategie deutlich simpler.

Das Kapitel ist wie folgt aufgebaut: Zunächst wird in Abschnitt 4.1 anhand eines Beispiels erklärt, warum die On-the-Fly-Suche bei GBGs deutlich schwieriger ist, als bei normalen Büchi-Spielen. Anschließend wird in Abschnitt 4.2 die allgemeine Idee für den Algorithmus beschrieben. Die genaue Vorgehensweise wird in Abschnitt 4.3 in mehreren Unterabschnitten

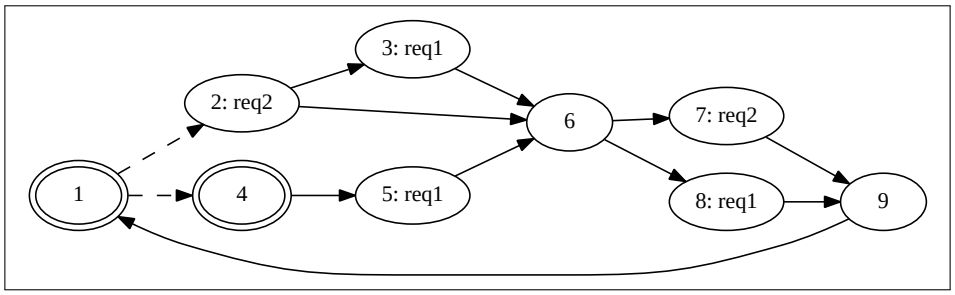

Abbildung 4.1: GBG-Spielgraph

erklärt. Aufgrund des Umfangs des Algorithmus werden hier nacheinander einzelne Teilfunktionen behandelt. Das Vorgehen wird anschließend in Abschnitt 4.4 an einem umfangreicheren Beispiel demonstriert, das viele der vorher beschriebenen Spezialfälle behandelt. Abschnitt 4.5 diskutiert die Komplexität des Verfahrens. In Abschnitt 4.6 wird die Korrektheit des Algorithmus begründet, und in Abschnitt 4.7 werden abschließend einige Details über die prototypische Umsetzung in ScenarioTools besprochen.

4.1 Schwierigkeiten

Die OTF-Suche in GBGs gestaltet sich deutlich schwieriger, als in normalen Büchi-Spielen. Der einfache OTF-Algorithmus mit einer einzelnen Siegbedingung ist relativ simpel: Vom Startzustand aus wird, unter Beachtung von unkontrollierbaren Transitionen, versucht eine Menge von Zielzuständen zu finden, deren Erreichen in jedem Fall garantiert werden kann. Dieses Verfahren wird mit allen so entstehenden Zielzuständen wiederholt. Gelingt die Suche nicht, werden Zustände als verlierend gekennzeichnet und ggf. von ihnen abhängende Zielzustände neu evaluiert. Jeder Zielzustand merkt sich also nur, welche anderen von ihm abhängen. Darüber hinaus gibt es keine weitere Interaktion zwischen den Zuständen. Die konkreten Pfade zwischen Zielzuständen sind ebenfalls vernachlässigt. Wichtig ist nur, dass es sie gibt.

Für gedächtnislose GBGs ist dieses Vorgehen nicht ausreichend, wie anhand des Beispiels in Abb. 4.1 demonstriert werden soll. In diesem gibt es drei Siegbedingungen: Die primäre Siegbedingung (durch doppelt umrandete Zustände hervorgehoben), sowie die sekundären Siegbedingungen

Siegbedingung	**Zielzustände**
Primär	$\{1, 4\}$
req1	$\{3, 5, 8\}$
req2	$\{2, 7\}$

Tabelle 4.1: Siegbedingungen von Beispiel aus Abbildung 4.1

$req1$ und $req2$. Diese sind in den Namen ihrer Zielzustände notiert. Die Siegbedingungen und deren Zielzustände sind in Tabelle 4.1 aufgelistet. Zustand 1 sei wie vorher beschrieben der Startzustand des Graphen.

Zunächst zum Vergleich das Vorgehen im einfachen OTF-Algorithmus. Hier würde nur die primäre Siegbedingung berücksichtigt werden. Vom Startzustand aus ist aufgrund der unkontrollierbaren Transitionen die Menge der erreichbaren Folgezielzustände $\{1, 4\}$. Zustand 4 wiederum hat als einzigen Folgezielzustand den Startzustand, 1. Damit kann jeder benötigte Zielzustand das Erreichen weiterer Zielzustände garantieren, weshalb der klassische OTF-Algorithmus an dieser Stelle fertig wäre.

Wenn man jedoch die sekundären Siegbedingungen hinzuzieht wird es schwieriger. Für eine gültige Lösung muss es möglich sein, unabhängig von den Entscheidungen der Umgebung unendlich oft mindestens einen Zustand aus jeder Siegbedingung zu erreichen. Dies kann alternativ dadurch ausgedrückt werden, dass auf jedem einfachem Zyklus im Graphen mindestens ein Zustand jeder Siegbedingung vorkommen muss.

Im einfachen OTF-Algorithmus war es nicht relevant, welcher Pfad zwischen zwei Zielzuständen gewählt wurde. Relevant war nur, dass es unter Berücksichtigung der unkontrollierbaren Transitionen Nachfolger gibt, deren Erreichen garantiert werden kann. Falls es zu einem Nachfolger mehrere mögliche Pfade gibt, war nicht relevant, welcher Pfad konkret gewählt wurde. Durch die zusätzlichen sekundären Siegbedingungen werden die gewählten Pfade jedoch entscheidend, da in vielen Fällen ein ganz bestimmter Pfad gewählt werden muss. Zustand 2 hat beispielsweise die Nachfolger 3 und 6. Zustand 3 ist ein Zielzustand für $req1$ und führt seinerseits zu Zustand 6. Falls nun Zustand 3 nicht besucht wurde, bleibt als einziger garantiert erreichbarer Zielzustand für $req1$ Zustand 8 übrig. Wurde jedoch Zustand 3 besucht, so ist es für diesen konkreten Pfad bei der Verzweigung in Zustand 6 egal, welcher Nachfolger besucht wird.

Es kann jedoch auch Abhängigkeiten zwischen den Pfaden verschiedener Zielzustände geben. Tabelle 4.2 zeigt die Pfade zwischen primären Zielzuständen und ihren Folgezielzuständen. Der Zielzustand 4 hat als einzigen Nachfolger den Zustand 1, welcher über die Pfade $p6$ oder $p7$ erreicht werden kann. In $p6$ sind sowohl $req1$ als auch $req2$ erfüllt, in $p7$ jedoch nur $req1$. Da auf dem Pfad $p5$ von 1 zu 4 keinerlei sekundäre Siegbedingungen erfüllt werden, ist $p7$ somit verlierend, weshalb nur $p6$ Teil einer gültigen Lösung sein kann.

Neben Zustand 4 hat 1 auch noch sich selbst als Nachfolger, welcher über die Pfade $p1$-$p4$ erreichbar ist. Bis auf $p3$ bilden sie alle einen gewinnenden Zyklus, da beide sekundären Siegbedingungen erfüllt sind. Jedoch stehen die Pfade $p2$ und $p4$ mit dem notwendigen Pfad $p6$ von Zustand 4 im Konflikt: Das Suffix $(6 \to 8 \to 9)$ kann nicht gewählt werden, da Zustand 4 zwingend das Vorkommen von Zustand 7 erfordert, welcher ein alternativer Nachfolger von Zustand 6 ist. Der für eine globale Lösung einzig gewinnende Pfad ist

Zustand	**Ziel**	**Pfade**	**Erf. Bed.**
1	1	$p1 : (1 \to 2 \to 3 \to 6 \to 7 \to 9 \to 1)$	{req1,req2}
		$p2 : (1 \to 2 \to 3 \to 6 \to 8 \to 9 \to 1)$	{req1,req2}
		$p3 : (1 \to 2 \to 6 \to 7 \to 9 \to 1)$	{req1}
		$p4 : (1 \to 2 \to 6 \to 8 \to 9 \to 1)$	{req1,req2}
1	4	$p5 : (1 \to 4)$	$\emptyset$
4	1	$p6 : (4 \to 5 \to 6 \to 7 \to 9 \to 1)$	{req1,req2}
		$p7 : (4 \to 5 \to 6 \to 8 \to 9 \to 1)$	{req1}

Tabelle 4.2: Pfade zwischen Zielzuständen im Beispiel aus Abbildung 4.1

also $p1$.

Eine gedächtnislose OTF-Suche auf GBGs muss deshalb die konkret gewählten Pfade beachten, und ggf. Pfade, die den notwendigen Pfaden anderer Zielzustände widersprechen als verlierend kennzeichnen. Dadurch entstehen komplexe Abhängigkeiten zwischen verschiedenen Zielzuständen, die weit über die des einfachen Algorithmus hinausgehen.

Zusammengefasst ergibt das folgende Anforderungen an den OTF-Algorithmus für GBGs:

- Es soll eine gedächtnislose Strategie gefunden werden, die garantiert, dass alle Siegbedingungen unendlich oft erfüllt werden können.
- Da jeder kontrollierbare Zustand aufgrund der Gedächtnislosigkeit nur einen Nachfolger haben kann, und die gewählten Pfade relevant sind, müssen die gewählten Nachfolgerzustand schon während der Suche verwaltet und beachtet werden.
- Dies kann dazu führen, dass andere Zielzustände eventuell zur Erfüllung von sekundären Siegbedingungen benötigte Zustände nicht erreichen können. In solchen Fällen müssen die widersprüchlichen Pfade, aufgrund derer diese Zustände nicht erreicht werden können, als verlierend betrachtet werden, und davon abhängende Zielzustände neu evaluiert werden.

4.2 Idee

Der Algorithmus wurde als Erweiterung bestehender On-The-Fly-Algorithmen konzipiert, und nutzt daher eine grundlegend ähnliche Vorgehensweise. Er arbeitet in drei Phasen, welche iterativ wiederholt werden. Zunächst wird eine modifizierte Version des einfachen OTF-Algorithmus durchgeführt. Die Lösung dieses Schrittes ist ein Kandidat für die GBG-Lösung. Anschließend werden alle Zyklen innerhalb des entstehenden Teilgraphen analysiert. Für diesen Zweck wurden im ersten Schritt zusätzliche Daten gesammelt. Jeder Zyklus wird darauf analysiert, ob

Algorithmus 1 : Ablauf des GBG-Algorithmus

Input : Spielgraph G, Siegbedingungen $\bigwedge_{i=1}^{k} g_i$
Result : *true* gdw. in G eine Strategie für das System gefunden wurde, die $\bigwedge_{i=1}^{k} g_i$ erfüllt

Erstelle Menge der Konfiguration *remaining* mit leerer Startkonfiguration.
while *remaining ist nicht leer* **do**
  Wähle eine Konfiguration *next* aus *remaining* und entferne sie.
  Führe modifizierte OTF-Suche mit verlierenden Pfaden *next* aus.
  if *Suche war erfolglos* **then**
    continue
  else if *Kandidat hat keine verlierenden Zyklen* **then**
    return *true*
  else
    Erstelle auf Basis von *next* neue Konfigurationen, die mögliche und garantierte verlierende Pfade aus den verlierenden Zyklen beinhalten, und füge sie zu *remaining* hinzu.
  end
end
return $false$

er die sekundären Siegbedingungen erfüllt, also für jede dieser Siegbedingung mindestens einen Zielzustand enthält. Ist dies nicht der Fall, so wird der Zyklus als verlierend vermerkt. Nach der Analyse aller Zyklen werden in der dritten Phase mögliche verlierende Teilpfade innerhalb der verlierenden Zyklen identifiziert. Dies sind Pfade zwischen zwei Zielzuständen der primären Siegbedingung, auf denen es Zustände gibt, die noch alternative kontrollierbare Nachfolger haben. Anschließend werden verschiedene Kombinationen verlierender Pfade als *Konfiguration* vermerkt, und der erste Schritt mit diesen wiederholt. Hier werden nun all diese verlierenden Pfade ignoriert, sodass alternative Pfade zu den Folgezielzuständen gefunden werden müssen. Gelingt dies nicht, so wird die Konfiguration verworfen. Ansonsten wird wieder auf verlierende Zyklen analysiert. Der Algorithmus terminiert wenn ein Kandidat ohne verlierenden Zyklus gefunden wurde, oder keine mögliche Konfiguration mehr übrig ist.

Die mögliche Anzahl Konfigurationen hängt von der Menge verlierender Pfade und deren Beschaffenheit ab, wodurch potentiell eine hohe Anzahl Konfigurationen entstehen kann. Pfade, die nur aus einer Transition bestehen oder gänzlich von der Umgebung kontrolliert werden führen jedoch zu keiner Vergrößerung der Konfigurationsmengen. Das Analysieren einer Konfiguration ist vergleichsweise günstig, solange keine gänzlich neuen Zustände erstellt werden müssen.

4.3 Vorgehen

Das Vorgehen gliedert sich in drei Teile: Die Erstellung eines Lösungskandidaten (Suche), die Verifikation des Kandidaten, und ggf. Anpassungen für die nächste Iteration. Während der Suche wird eine Lösung für das Büchi-Spiel der primären Siegbedingung gesucht. In der Verifikation wird das Ergebnis dieser Suche auf verlierende Zyklen untersucht. Anschließend werden, soweit nötig, neue Konfigurationen erzeugt oder das Ergebnis der Suche als Lösung ausgegeben. Diese drei Schritte werden solange wie nötig iterativ wiederholt. Der Ablauf der Phasen ist in Algorithmus 1 dargestellt.

4.3.1 Suche

Zunächst die modifizierte OTF-Suche. Sie funktioniert von der Grundidee ähnlich wie im einfachen Fall, jedoch wurde sie um Fähigkeiten erweitert, um verlierende Pfade zu ignorieren und Abhängigkeiten zwischen Zielzuständen zu verwalten. Von Zielzuständen aus wird zunächst in einer Vorwärtsanalyse versucht, weitere Zielzustände zu erreichen. Anschließend wird versucht, diese in einer Rückwärtsanalyse zurück zum initialen Zustand zu propagieren.

Während der Suche werden einige Daten verwaltet, die im Pseudo-Code nicht explizit als Ein- oder Ausgabe gekennzeichnet sind:

- *stack* verwaltet zur Analyse ausstehende Transitionen. Mit den Transitionen ist jeweils vermerkt, ob sie Vorwärts- oder Rückwärts analysiert werden müssen. Eine zur Vorwärtsanalyse ausstehende Transition ist zusätzlich mit einem unvollständigem Pfad hinterlegt, während eine zur Rückwärtsanalyse ausstehende Transition keine weiteren Daten enthält.
- $next(v)$ für $v \in V$ bildet einen Zustand auf eine Menge von Zuständen ab. Dies sind die direkten Nachfolgerzustände von v. Die Menge ist einelementig falls v vom System kontrolliert ist.
- $dep(v)$ bildet einen Zustand v auf die Menge der Zielzustände ab, die von ihm aus erreicht werden können. Dies sind also die Abhängigkeiten des Zustands.
- $paths(v)$ speichert, welche (vorläufig) gewinnenden Pfade durch den Zustand v verlaufen.
- *losesAt* speichert Paare (p, v) von Pfaden und Zuständen. Der Pfad p ist verlierend, und ab dem Zustand v wurde der *next*-Funktion gefolgt. Dies wird genutzt, um Pfade zu finden, deren Suffixe verlierende Zyklen erzwingen.

Durch die Nutzung von *stack* wird der Graph über eine Tiefensuche erforscht. Zunächst werden für den Startzustand der Suche alle ausgehenden

Transitionen auf dem Stack platziert. In jeder Iteration wird das oberste Stack-Element entfernt und ggf. durch die ausgehenden Transitionen ersetzt. Daraus folgt, dass während der Suche der Sub-Graph, zu dem eine ausgehende Transition eines Zustands führt, soweit nötig komplett erforscht wird, bevor weitere ausgehende Transitionen des Zustands betrachtet werden.

Die Funktion *next* wird stückweise während der Suche erstellt. Findet ein Zielzustand einen gültigen Pfad zu einem Folgezielzustand, so werden die gewählten kontrollierbaren Entscheidungen entlang des Pfades der Funktion hinzugefügt. Bei unkontrollierbaren Zuständen bildet sie auf alle Nachfolgerzustände ab, da jeder dieser Zustände erreicht werden könnte. Darüber hinaus werden auf die selbe Art die Funktionen *dep* und *paths* aufgebaut. Wird ein Pfad als verlierend erkannt, so wird er aus den *paths*-Mengen entfernt, und falls $paths(v)$ für einen Zustand v nach solch einer Operation nun leer ist, wird v ebenfalls aus der Funktion *next* entfernt. Wird während der Suche ein Zustand gefunden, für den $next(v)$ definiert ist, so wird der Funktion bis zu einem oder mehreren Endzuständen gefolgt. Dabei sind alle unkontrollierbaren Transitionen zu nehmen, wodurch mehrere Pfade entstehen können.

Weiterhin sind während der Vorwärtsanalyse die Vorgänger von Zuständen zu verwalten. Während der Rückwärtsanalyse werden sie genutzt, um die möglichen Vorgänger zu aktualisieren. Dies ist im Pseudo-Code nicht explizit erwähnt.

Die Suche besteht aus folgenden Teilfunktionen:

- Die Funktion `onTheFlySearch` initiiert die Suche und evaluiert, ausgehend vom Startzustand, alle nötigen Zielzustände.
- Die Funktion `reach` nimmt als Parameter einen Zielzustand und versucht, unter Berücksichtigung aller unkontrollierbaren Transitionen eine passende Menge von erreichbaren, nicht verlierenden Folgezielzuständen zu finden.
- Die Funktion `forwardExploreTransition` wird genutzt, um eine Transition innerhalb von `reach` vorwärts zu erkunden.
- Die Funktion `completePath` wird während der Ausführung von `forwardExploreTransition` aufgerufen, falls ein durch *next* bereits festgelegter Pfad vervollständigt werden muss. Dies könnte etwa eintreten, wenn der Zustand schon in einem früheren Aufruf von `reach` betrachtet wurde.
- Die Funktion `backwardsExploreTransition` wird zur Rückwärtsanalyse einer Transition innerhalb von `reach` genutzt.

Hilfsfunktionen

Zunächst müssen einige Funktionen beschrieben werden, die zwar nicht direkt die Suche implementieren, jedoch wichtig für die Verwaltung des

Algorithmus 2 : `cleanLosingPath`

Input : Pfad *path*

foreach *Zustand* v *in path* **do**
  Entferne *path* aus $paths(v)$.
  Sei *count* die Anzahl der Pfade p in $paths(v)$ mit $end(p) \neq v$ oder $start(p) = v$.
  if $count = 0$ **then**
    Setze $next(v)$ auf undefiniert.
  end
end

Setze $dep(start(path))$ auf undefiniert.
Vermerke *path* als verlierend.

Zustands sind. Dies sind folgende Funktionen:

- Die Funktion `cleanLosingPath` wird immer dann aufgerufen, wenn ein *Pfad* als verlierend erkannt wird.
- Die Funktion `setLosing` wird immer dann aufgerufen, wenn ein *Zustand* als verlierend erkannt wird.

`cleanLosingPath` ist in Algorithmus 2 dargestellt. Wenn ein Pfad p verliert, wird er aus allen *paths*-Mengen entfernt. Dazu kann einfach über die Zustände des Pfades iteriert werden. Nach dem Entfernen aus $paths(v)$ muss unter Umständen die *next*-Funktion für den Zustand v zurückgesetzt werden. Dies ist der Fall, falls nun keine Pfade mehr hinterlegt sind, die entweder nicht in v enden oder die in v starten. Dann verläuft kein Pfad mehr über die in *next* definierten ausgehenden Transitionen, weshalb die Strategie für v zurückgesetzt werden kann um ggf. neue Pfade zu finden. Zuletzt wird $dep(start(p))$ zurückgesetzt. Da nur gültige Pfade in *paths* hinterlegt sind, hängt der Startzustand des Pfades von diesem ab, um einen Zielzustand zu erreichen. Der Pfad ist nun verlierend, weshalb der Startzustand nicht mehr garantieren kann, dass dies eintritt. Er muss deshalb neu evaluiert werden, was dadurch bewirkt wird, dass die *dep*-Funktion für diesen Zustand zurückgesetzt wird.

`setLosing` wird aufgerufen, wenn ein Zustand v als verlierend erkannt wurde. Das Vorgehen ist relativ simpel, weshalb an dieser Stelle auf Pseudo-Code verzichtet wird. Wenn ein Zustand verliert kann er nicht mehr Teil einer Siegstrategie für das System sein. Alle Pfade, die durch den Zustand verlaufen, können deshalb ebenfalls nicht enthalten sein. Es wird deshalb für alle $p \in paths(v)$ die Funktion `cleanLosingPath` aufgerufen, wodurch die Pfade entfernt werden und die dazugehörigen Startzustände neu evaluiert werden müssen. Darüber hinaus merkt sich der Algorithmus, dass v verliert.

Algorithmus 3 : `onTheFlySearch`

Input : Graph $G = (V, E)$
Output : *true* gdw. primäre Büchi-Bedingung erfüllbar mit aktueller Konfiguration erfüllbar

if *Startzustand start wurde noch nicht untersucht* **then**
  if $reach(start) = false$ **then return** $false$;
end
while ∃ *Zielzustand* v *mit* $dep(v)$ *ist undefiniert* **do**
  if $reach(v) = false$ **then**
    $setLosing(v)$
    if $v = start$ **then return** $false$;
  end
end
return *true*

On-The-Fly-Suche

Dieser Schritt funktioniert analog zum einfachen OTF-Algorithmus. Das Vorgehen ist in Algorithmus 3 dargestellt. Zunächst wird versucht, vom Startzustand *start* aus Zielzustände zu erreichen. Dazu wird die Funktion `reach` aufgerufen. Gelingt dies nicht, so kann es keine Lösung geben und der Algorithmus gibt eine negative Antwort zurück. Die so gefundenen Zielzustände müssen nun ihrerseits wieder Zielzustände erreichen können und werden nacheinander analysiert. Beim erfolgreichen Aufruf von $reach(v)$ wird der Wert von $dep(v)$ gesetzt, welcher die von v gefundenen Zielzustände angibt. Für so erstmals gefundene Zielzustände ist *dep* noch undefiniert. Falls `reach` jedoch für einen dieser Zustände eine negative Antwort zurückgibt, so heißt das nicht zwangsläufig, dass es keine Lösung gibt. Der betrachtete Zustand wird als verlierend gekennzeichnet, wodurch auch seine Abhängigkeiten neu evaluiert werden müssen, da die *dep*-Funktion für sie zurückgesetzt wird. Dort können sie potentiell noch alternative Zielzustände finden. Dies wird solange fortgesetzt, bis jeder Zielzustand das Erreichen weiterer Zielzustände garantieren kann oder der Startzustand verliert.

Vorwärtsanalyse von Transitionen

Eingabe dieses Schrittes ist ein Pfad *path* und eine Transition t, wobei *path* beim Zustand $source(t)$ endet. Das Vorgehen ist in Algorithmus 4 dargestellt. Zunächst wird *path* um die Transition t erweitert, sodass er nun auf $target(t)$ endet. Der erweiterte Pfad sei im Folgenden $newPath$. Dabei ist zu beachten, dass ein neues Objekt erzeugt werden muss, da *path* durchaus Eingabe für mehrere Transitionen sein kann. Anschließend können folgende Fälle eintreten, sortiert nach der Priorität:

Algorithmus 4 : `forwardExploreTransition`

Input : Unvollständiger Pfad *path*, Transition t

Sei $dest = target(t)$.
if *dest ist ein Deadlock-Zustand* **then**
  setLosing(*dest*)
end

Sei *newPath* der Pfad *path* erweitert um Transition t.
if *dest ist verlierend* **then**
  Füge t zur Rückwärtsanalyse zu *stack* hinzu.
else if *dest ist ein Zielzustand* **then**
  if *newPath ist nicht verlierend* **then**
    Füge *newPath* zu $paths(target(t))$ hinzu.
    Füge t zur Rückwärtsanalyse zu *stack* hinzu.
  end
else if $next(dest)$ *ist definiert* **then**
  $completePath(path, t)$
else if $target(t) \notin path$ **then**
  foreach *ausgehende Transition succ von dest* **do**
    Füge $(newPath, succ)$ zur Vorwärtsanalyse zu *stack* hinzu.
  end
end

1. Der Zustand $target(t)$ ist ein Deadlock-Zustand (hat also keine ausgehenden Transitionen) oder verliert. Deadlock-Zustände verlieren per Definition, da durch sie die Büchi-Bedingung unerfüllbar ist. In diesem Fall wird t zur Rückwärtsanalyse vermerkt, um den Status dort soweit wie nötig zu propagieren.

2. Der Zustand $target(t)$ ist ein Zielzustand. Hier muss geprüft werden, ob *newPath* ein verlierender Pfad ist. In diesem Fall wird keine weitere Aktion ausgeführt, der Pfad endet somit an dieser Stelle. Ist er jedoch nicht verlierend, so wird *newPath* zur *path*-Funktion des Zielzustands hinzugefügt und t zur Rückwärtsanalyse vermerkt. Der Pfad wird dort, soweit möglich, in Richtung des Startzustands propagiert.

3. Für den Zustand $target(t)$ ist die *next*-Funktion definiert. In diesem Fall wurde $target(t)$ schon von einer Rückwärtsanalyse betrachtet und kann deshalb das Erreichen von Zielzuständen garantieren. Jedoch kann an dieser Stelle noch nicht die Rückwärtsanalyse angestoßen werden, da noch geprüft werden muss, ob durch die *next*-Funktion verlierende Pfade entstehen. Das weitere Vorgehen wird in der Funktion `completePath` beschrieben.

4. Der Zustand $target(t)$ ist bereits in *path* enthalten. In diesem Fall

wurde ein Zyklus gefunden, in dem kein Zielzustand der primären Siegbedingung enthalten ist. Es findet deshalb keine weitere Aktion statt, die Suche für diese Transition endet hier.

5. Tritt keiner dieser Fälle ein, so muss die Vorwärtsanalyse fortgesetzt werden. Für alle Nachfolger *succ* von $target(v)$ wird $(newPath, succ)$ zur Vorwärtsanalyse vermerkt.

Das Vorgehen entspricht damit weitestgehend dem einfachen OTF-Algorithmus. Der wichtigste Unterschied ist hier, dass bei einem Zustand v, der schon durch die Rückwärtsanalyse erreicht wurde (für den also $next(v)$ definiert ist), nicht direkt eine weitere Rückwärtsanalyse der Transition angestoßen wird, sondern die Pfade entlang der *next*-Funktion bis zu den jeweiligen Zielzuständen vervollständigt werden. Dies ist wichtig, um verlierende Pfade zu erkennen.

Vervollständigung von Pfaden

Diese Funktion wird aufgerufen, falls während der Vorwärtsanalyse ein Zustand gefunden wird, für den in der *next*-Funktion schon einer oder mehrere Nachfolger festgelegt wurden. Dieser Zustand wurde also entweder schon vorher von diesem Startzustand von `reach` aus betrachtet und bereits rückwärts analysiert, oder er ist ebenfalls Teil eines Pfades, der von einem weiteren Zielzustand ausging. Die *next*-Funktion führt zwangsläufig zu einem oder mehreren Zielzuständen, da sie nur in der Rückwärtsanalyse festgelegt wird. Dadurch kann in jeden Fall garantiert werden, dass von der initiierenden Transition *start* aus unabhängig von der Umgebung ein Zielzustand erreicht werden kann. Jedoch ist es möglich, dass dabei ein Pfad entsteht, der laut der aktuellen Konfiguration verliert. Dies kann erst ab der zweiten Iteration des Algorithmus eintreten, nachdem von der Verifikation verlierende Zyklen gefunden wurden, deren Pfade ein Suffix mit anderen Pfaden teilen. In diesem Fall kann von dieser Transition aus kein Erreichen eines Zielzustandes garantiert werden.

Das Vorgehen ist in Algorithmus 5 dargestellt. Zunächst wird der Pfad mithilfe der *next*-Funktion vervollständigt. Da ein Zustand mehrere Nachfolger haben kann, falls er von der Umgebung kontrolliert wird, können hier eine Vielzahl von Pfaden entstehen. Dazu wird ein Stack oder eine vergleichbare Datenstruktur verwendet, in der jedes Element ein Paar bestehend aus einem unvollständigem Pfad *path* und der nächsten hinzuzufügenden Transition t ist. Für jedes Element wird zunächst ein neuer Pfad *newPath* erzeugt, der *path* um die Transition t erweitert. Falls der Pfad nun auf einem Zielzustand endet, dann ist der Pfad komplettiert und muss nicht weiter erweitert werden. Allerdings muss geprüft werden, ob der Pfad verliert. Diese Information ist in der aktuellen Konfiguration hinterlegt. Falls der Pfad verliert, wird zusammen mit ihm in der Menge *losesAt* vermerkt,

Algorithmus 5 : `completePath`

Input : Unvollständiger Pfad *input*
Input : Transition *start*, deren Zielzustand der erste Zustand mit definierter *next*-Funktion ist

Erzeuge Stack *pathStack* und füge $(start, input)$ hinzu.
while *pathStack ist nicht leer* **do**
  Entferne oberstes Element $(t, path)$ von *pathStack*.
  Sei *newPath* der Pfad *path* erweitert um Transition t.
  if $target(t)$ *ist Zielzustand* **then**
    if *newPath ist verlierend* **then**
      Füge $(newPath, target(start))$ zu *losesAt* hinzu.
    end
  else
    foreach $transition \in next(target(t))$ **do**
      Füge $(transition, newPath)$ zu *pathStack* hinzu.
    end
  end
end
if *kein Pfad war verlierend* **then**
  Hinterlege alle vervollständigten Pfade an Zuständen ab $target(start)$.
  Füge *start* zur Rückwärtsanalyse zu *stack* hinzu.
end

ab welchem Zustand der *next*-Funktion gefolgt wurde. Dies ist der Zustand $target(start)$. Die Menge wird später betrachtet, falls die Suche für den Startzustand erfolglos war. Endet *newPath* nicht auf einem Zielzustand, so muss er weiter erweitert werden. Dazu werden alle Paare $(newPath, succ)$ mit $succ \in next(target(t))$ zum Pfad-Stack hinzugefügt.

Die neu entstandenen Pfade müssen nun noch in die *paths*-Mengen aufgenommen werden. Dies wird jedoch nur gemacht, falls keiner der Pfade verlierend war. Da die Pfade ab dem Zustand $target(start)$ fest vorgeschrieben sind, erzwingt die Inklusion eines Pfades auch die aller anderen. Verliert also einer der so entstandenen Pfade, dann kommen auch alle anderen Pfade, die sich ab dem Zustand $target(start)$ ein Präfix mit ihm teilen, nicht in Betracht. In diesem Fall wird auch keine Rückwärtsanalyse von *start* angestoßen, die gesamte Analyse dieses Teilpfades endet an dieser Stelle. War kein Pfad verlierend, so werden die Pfade in alle *path*-Mengen hinzugefügt, die sie ab dem Zustand $target(start)$ besuchen. Dies entspricht dem Vorgehen einer normalen Rückwärtsanalyse, die an diesen Stellen jedoch unnötig ist, da die Zustände schon evaluiert wurden. Anschließend wird

Algorithmus 6 : `backwardsExploreTransition`

Input : Transition t die zur Rückwärtsanalyse aussteht

Sei src=$source(t)$.
if *src verliert und ist nicht als verlierend bekannt* **then**
  $setLosing(src)$
  Füge alle von src abhängenden Zustände dem Evaluationsstack zur Rückwärtsanalyse hinzu und entferne alle Nachfolger, die noch zur Vorwärtsanalyse ausstehen.
  return
end

Sei $succ = \{target(t)\}$ falls t kontrollierbar, sonst die Menge aller unkontrollierbaren Nachfolger von src.
if *alle Elemente aus succ können Zielzustand erreichen* **then**
  $dep(src) = \bigcup_{s \in succ} dep(s)$
  $paths(src) = \bigcup_{s \in succ} paths(s)$
  $next(src) = succ$
  Füge alle von src abhängenden Zustände dem Evaluationsstack zur Rückwärtsanalyse hinzu und entferne alle Nachfolger, die noch zur Vorwärtsanalyse ausstehen.
end

start zur Rückwärtsanalyse vermerkt, sodass die entstandenen Pfade weiter propagiert werden können.

Rückwärtsanalyse von Transitionen

Die Rückwärtsanalyse wird angestoßen, wenn während der Vorwärtsanalyse ein Zielzustand oder ein verlierender Zustand gefunden wird. Über die Rückwärtsanalyse wird dieser Status nun soweit wie möglich bzw. nötig zurückpropagiert. Hierbei werden auch die gewinnenden Pfade an den Zuständen des Pfades vermerkt. Eingabe ist die Transition, die zurückgegangen werden soll. Der Zielzustand der Transition wurde dabei schon betrachtet, und der Startzustand soll unter Berücksichtigung der Kontrollierbarkeit der Transition erreicht werden. Solange Rückwärtsanalysen ausgeführt werden können haben sie Vorrang vor Vorwärtsanalysen, werden also oben auf dem Stack platziert. Dieser Schritt des Algorithmus funktioniert weitestgehend analog zum einfachen OTF-Algorithmus. Das Vorgehen ist in Algorithmus 6 dargestellt.

Während der Rückwärtsanalyse der Transition t können folgende Fälle auftreten:

1. $source(t)$ ist nicht als verlierend bekannt, jedoch ist ein unkontrollierbarer Nachfolger oder alle kontrollierbaren Nachfogler verlierend.

2. t ist unkontrollierbar, und $source(t)$ hat noch weitere unkontrollierbare Nachfolger, für die das Erreichen von Zielzuständen noch nicht garantiert werden kann.
3. t ist entweder kontrollierbar, oder alle Nachfolger von $source(t)$ können das Erreichen von Zielzuständen garantieren.

Im ersten Fall muss $source(t)$ als verlierend markiert werden, da das System von diesem Zustand aus keine Siegchancen mehr hat. Alle Zustände, die von $source(t)$ abhängen müssen anschließend ebenfalls von der Rückwärtsanalyse evaluiert werden und müssen so auf dem Stack platziert werden. Falls noch weitere ausgehende Transitionen von $source(t)$ zur Vorwärtsanalyse ausstehen, so müssen diese vom Stack entfernt werden.

Im zweiten Fall kann die Rückwärtsanalyse noch nicht fortgesetzt werden, da der Status von $source(t)$ zu diesem Zeitpunkt noch nicht evaluierbar ist. Es findet deshalb keine weitere Aktion statt und die Suche wird normal mit dem nächsten Element des Stacks fortgeführt. Solange der Suchstack aufgrund verlierender Pfade nicht vorzeitig erschöpft wird, wird $source(t)$ früher oder später von einer weiteren seiner ausgehenden Transitionen evaluiert, wo unter Umständen einer der anderen beiden Fälle auftreten kann.

Im dritten Fall muss zwischen kontrollierbaren und unkontrollierbaren Transitionen unterschieden werden. Hier müssen unter Umständen mehrere Nachfolger betrachtet werden. Falls t kontrollierbar ist, so können die Abhängigkeiten von $target(t)$ direkt an $source(t)$ propagiert werden, und alle Pfade von $target(t)$, die die Transition t enthalten, werden zu $paths(source(t))$ hinzugefügt. Falls t unkontrollierbar ist, so müssen alle unkontrollierbaren Nachfolger betrachtet werden. Die Abhängigkeiten ergeben sich hier aus der Vereinigung aller Abhängigkeiten der Nachfolger, und es müssen alle Pfade der Nachfolger hinzugefügt werden, die über $source(t)$ verlaufen. Hier würden also auch die Pfade von Transitionen hinzugefügt werden, deren Analyse zuvor im zweiten Fall endete. Anschließend muss die Rückwärtsanalyse fortgeführt werden, und alle Transitionen, die einen von $source(t)$ abhängenden Zustand mit diesem verbinden, müssen zur Rückwärtsanalyse auf dem Stack platziert werden. Falls noch weitere ausgehende Transitionen von $source(t)$ zur Vorwärtsanalyse ausstehen, so müssen diese vom Stack entfernt werden. Weiterhin werden in diesem Schritt die Nachfolger $next(source(t))$ festgelegt.

reach

Nun werden die einzelnen Funktionen verbunden, um für einen Zustand die Suche zu implementieren. Das Vorgehen ist in Algorithmus 7 dargestellt.

Zu Beginn der Funktion ist $dep(v)$ undefiniert, und *stack* leer. Zunächst werden alle ausgehenden Transitionen des Startzustands v zur Vorwärtsanalyse dem Stack hinzugefügt. Ist v unkontrollierbar muss nun für jede

Algorithmus 7 : `reach`

Input : Zustand v, von dem aus die Suche ausgeführt wird
Result : *true* gdw. das Erreichen von Zielzuständen garantiert werden kann

Initialisiere *stack* mit ausgehenden Transitionen von v.
$dep(v) \leftarrow undef$
while *stack ist nicht leer und* $dep(v)$ *ist undefiniert* **do**
  Entferne oberstes Element *elem* von *stack*.
  if *elem steht zur Vorwärtsanalyse aus* **then**
    $forwardExploreTransition(elem.path, elem.transition)$
  else
    $backwardsExploreTransition(elem.transition)$
  end
end
Entferne temporäre Pfade, die nicht zu v propagiert wurden.
if $dep(v)$ *ist undefiniert und losesAt ist nicht leer* **then**
  foreach $(path, state) \in losesAt$ **do**
    Rufe *cleanLosingPath* für alle Pfade auf, die ab *state* mit *path* identisch sind.
  end
  return $reach(v)$
end

return *true* gdw. $dep(v)$ nicht undefiniert ist.

dieser Transitionen ein Pfad zu Zielzuständen gefunden werden, ansonsten nur über mindestens eine. In letzterem Fall kann die Reihenfolge, in der die Transitionen betrachtet werden, einen großen Unterschied ausmachen. Dies wird hier jedoch vernachlässigt, die Transitionen werden in beliebiger Reihenfolge auf *stack* gespeichert.

Solange $dep(v)$ undefiniert ist, existiert nicht für alle nötigen Nachfolger von v ein Pfad zu Zielzuständen, da dieser Wert bei erfolgreicher Rückwärtsanalyse gesetzt wird. Solange *stack* nicht leer ist, kann es jedoch noch mögliche Pfade geben, da über bisher unbetrachtete Transitionen neue Pfade entstehen können. Trifft eine dieser beiden Bedingungen nicht mehr zu, so war die Suche entweder erfolgreich ($dep(v)$ ist definiert) oder erfolglos (*stack* ist leer). Entsprechend werden von *stack* referenzierte Transitionen solange in einer Schleife analysiert, bis einer dieser beiden Fälle eintritt. Dazu werden, abhängig von der Art der Analyse, die eine Transition erfordert, die vorher beschriebenen Funktionen `forwardExploreTransition` und `backwardsExploreTransition` aufgerufen.

Nach Abschluss der Schleife müssen zunächst alle während der Suche erstellten temporären Pfade betrachtet werden. Alle Pfade, die durch die

Rückwärtsevaluation nicht bis zu v propagiert werden konnten, müssen aus der *paths*-Funktion entfernt werden. Dies sind „gescheitertete“ Pfade, die je nach Nutzung der *paths*-Funktion in der Implementierung zu Problemen führen könnten (etwa, indem Zyklen anhand von in Zuständen hinterlegten Pfaden berechnet werden. Jeder dieser Pfade ist zumindest in einem Zielzustand hinterlegt). Dazu können die Pfade schon während der Suche gelistet werden. Anschließend muss nur noch rückwärts durch sie iteriert werden, solange sie in den jeweiligen Zuständen hinterlegt sind. Nun kann einer von drei Fällen eintreten:

1. $dep(v)$ ist definiert. In diesem Fall kann v garantieren, dass Zielzustände über Pfade erreicht werden, die nicht als verlierend vermerkt wurden. Das Ergebnis ist in diesem Fall *true*.
2. $dep(v)$ ist nicht definiert, jedoch ist $losesAt$ nicht leer. In diesem Fall *könnte* v potentiell das Erreichen von Zielzuständen garantieren, jedoch wird es durch die *next*-Funktion in verlierende Pfade gezwungen. Alle anderen Möglichkeiten, einen Zielzustand zu erreichen, wurden erschöpft (da *stack* leer ist), sodass ein durch die *next*-Funktion definiertes Suffix in diesem Fall inkompatibel mit einer Lösung für den Zustand v ist. Alle anderen Pfade, die ebenfalls das selbe Suffix verwenden, können deshalb auch nicht Teil einer Lösung sein, in der die auslösenden Pfade verlieren. All diese werden also ebenfalls als verlierend markiert. Dadurch wird die Definition der *next*-Funktion an mindestens einem Zustand des Suffixes aufgehoben, sodass hier nun potentiell weitere Pfade gefunden werden können. Die Suche wird also durch erneuten Aufruf von `reach` wiederholt.
3. $dep(v)$ ist nicht definiert und $losesAt$ ist leer. In diesem Fall kann kein Erreichen von Zielzuständen garantiert werden, unabhängig von verlierenden Pfaden und der *next*-Funktion. Das Ergebnis ist $false$, und v muss deshalb als verlierender Zustand angesehen werden. Zustände, die von v abhängen, müssen erneut analysiert werden.

Das Ergebnis wird anschließend von der oben beschriebenen Funktion `onTheFlySearch` ausgewertet. Unter Umständen müssen die von v gefunden Zielzustände nun ebenfalls analysiert werden. $dep(v)$ gibt nun alle Zustände an, von denen v abhängt.

4.3.2 Verifikation

Das Ergebnis der vorherigen Suche ist ein Subgraph, in dem garantiert werden kann, dass von jedem Zustand aus immer ein Zielzustand der primären Siegbedingung erreichbar ist. Weiterhin werden Pfade ignoriert, die als verlierend gekennzeichnet wurden und für jeden vom System kontrollierbaren Zustand im Subgraph wurde ein Nachfolger gewählt. Dieser Graph wird als

Algorithmus 8 : `checkCycles`

Input : Lösungs-Kandidat aus der Suche
Result : Menge der verlierenden Zyklen

foreach *gültigen Zyklus c im Kandidaten* **do**
  Sei *grnt* die Menge der sekundären Siegbedingungen, für die in c kein Zielzustand vorkommt.
  Sei *asm* die Menge der Assumptions, für die in c kein Zielzustand vorkommt.
  if $grnt \neq \emptyset$ *und* $asm = \emptyset$ **then**
    Markiere c als verlierend.
  end
end

return Alle als verlierend gekennzeichneten Zyklen.

Kandidat bezeichnet. Er ist eine mögliche Lösung, jedoch müssen noch die sekundären Siegbedingungen verifiziert werden.

Auch für diese gilt, dass sie von jedem Zustand aus immer erreichbar sein müssen. Dies kann alternativ so aufgefasst werden, dass sie in jedem einfachem Zyklus innerhalb des Kandidaten erfüllt sind. Gibt es einen Zyklus, in dem sie nicht erfüllt sind, so könnte die Umgebung das System in diesen Zyklus zwingen und so die Erfüllung verhindern. Algorithmus 8 zeigt das Vorgehen zur Verifikation der Zyklen. Ein Zyklus gewinnt, wenn für jede sekundäre Siegbedingung ein Zielzustand auf dem Zyklus vorkommt. Die einzige Ausnahme sind Zyklen, in denen es Assumptions gibt, die in keinem Zustand erfüllt sind. Hier werden Annahmen über das Verhalten der Umgebung verletzt, und das System ist nicht verpflichtet, seine eigenen Verpflichtungen einzuhalten. Die Menge der verlierenden Zyklen wird für den nächsten Schritt vermerkt.

Zum Finden der einfachen Zyklen kann Johnsons Algorithmus [12] genutzt werden. Er arbeitet in $O(n^2 \cdot c)$, wobei n die Anzahl Zustände und c die Anzahl eindeutiger einfacher Zyklen im Graph ist. In einem gerichteten Graphen kann es exponentiell viele eindeutige einfache Zyklen geben, weshalb er eine exponentielle Laufzeit hat. Dies wirkt bei der Größe der Zustandsräume von Szenario-Graphen zunächst sehr ungünstig. Es gibt jedoch einige Eigenheiten, die diese Laufzeit für typische Spezifikationen weitaus niedriger halten, als im Worst-Case:

- Die Worst-Case-Laufzeit nimmt einen vollständigen Graphen an. Speziell in ScenarioTools sind Szenario-Graphen im Durchschnitt sehr dünn besetzt, die Menge der ausgehenden Transitionen eines Zustands ist beschränkt durch die Anzahl der möglichen Umgebungs-Events und die Menge der möglichen Objekt-Bindings. Zustände vom Grad

> 5 sind dadurch ungewöhnlich, was die Menge möglicher Zyklen beträchtlich reduziert.

- Die meisten Zustände müssen gar nicht betrachtet werden. Jeder gültige Zyklus enthält mindestens einen Zustand, der Zielzustand der primären Siegbedingung ist. Auf den Pfaden zwischen solchen Zielzuständen gibt es keine Zyklen, da diese von vornherein als verlierend erkannt und verworfen werden.
- Die möglichen Pfade, die von einem Zielzustand aus zu weiteren Zielzuständen genommen werden können, sind schon aus der Suche bekannt.

Es ist daher ausreichend, die Zyklen auf einem vereinfachten Graph zu suchen, in dem nur die Zielzustände der primären Siegbedingung enthalten sind und jeder Zustand mit den Zuständen verbunden ist, von denen er abhängt. Für jeden Zyklus innerhalb dieses vereinfachten Graphen müssen nun nur noch die möglichen Kombinationen von Pfaden zwischen aufeinanderfolgenden Zuständen innerhalb des Zykluses getestet werden. Dies verbessert zwar die Worst-Case-Laufzeit nicht, jedoch wird es für einen typischen Szenario-Graphen durch diese Eigenschaften effizient berechenbar. Die Zyklen können auch schon während der Suche erkannt werden, vgl. Abschnitt 4.7.

4.3.3 Auswertung verlierender Zyklen

Aus den verlierenden Zyklen müssen nun Anpassungen für die folgenden Iterationen des Algorithmus berechnet werden. Die Zyklen als Ganzes sind zwar verlierend, jedoch heißt das nicht, dass auch jedes Teilstück verliert. Die Ursache kann schon in einem einzigen Pfad zwischen zwei Zielzuständen liegen. In diesem Schritt werden mögliche Ursachen identifiziert und aus ihnen neue Konfigurationen gebildet, die geprüft werden müssen. Eine Konfiguration ist dabei eine Menge verlierender Pfade.

Dazu werden für jeden Zyklus enthaltene Pfade gesucht, auf denen noch weitere vom System kontrollierbare Transitionen gewählt werden könnten. Hierdurch könnte das System einen weiteren Zyklus erreichen, der eventuell nicht verliert. Falls es keine Teilpfade gibt, auf denen das System eine andere Entscheidung treffen könnte, dann verliert jeder im Zyklus enthaltene Pfad. Es ist dem System nicht möglich, das Verlieren dieses Zykluses zu verhindern. Auch wenn Teile möglicherweise noch zu anderen Zyklen gehören, die Umgebung kann von jedem Teilstück des Zykluses ein Verlieren des Systems erzwingen.

Für jeden Zyklus wurde nun eine Menge von Pfaden gefunden, die verlierend sein könnten. Enthält ein Zyklus keine Alternativen, so werden einfach alle Pfade als verlierend in jeder neuen Konfiguration übernommen. Alle weiteren Zyklen werden zu neuen Konfigurationen kombiniert, die

Algorithmus 9 : `createConfigurations`

Input : Verlierende Zyklen
Result : Menge der neuen Konfigurationen

Sei *current* die aktuelle Konfiguration.
Sei $new = \{current\}$ eine Menge von Konfigurationen.

foreach *verlierenden Zyklus p* **do**
  Sei *controllable* die Menge der Teilpfade von p, in denen das System für mindestens einen Zustand alternative Nachfolger hat.
  if *controllable ist nicht leer* **then**
    $new = new \times controllable$
  else
    $new = new \oplus p$
  end
end

return *new*

jeweils einen Pfad aus jedem verlierenden Zyklus enthalten. Auf diese Art wird jede mögliche Kombination erzeugt. Das Vorgehen ist in Algorithmus 9 dargestellt. $new \oplus p$ stehe dabei für das Hinzufügen von p zu jedem Element von *new*, der Operator $\times$ für das kartesische Produkt zwischen den Konfigurationen und den Elementen der kontrollierbaren Pfade.

Die neuen verlierenden Pfade müssen aus der *paths*-Funktion entfernt werden. Dabei muss ggf. auch die *next*-Funktion zurückgesetzt werden, falls nun keine Pfade mehr durch einen Zustand verlaufen. Alle Zielzustände der primären Siegbedingung, von denen ein Pfad als verlierend verzeichnet wurde, müssen in der neuen Konfiguration erneut analysiert werden.

4.4 Beispiel

Nun soll der Ablauf des Algorithmus an einem umfangreichen Beispiel demonstriert werden, welches in Abbildung 4.2 dargestellt ist. Dort gibt es wieder drei Siegbedingungen: die primäre Siegbedingung mit den Zielzuständen $\{1, 7\}$, sowie die sekundären Siegbedingungen $req1$ mit den Zielzuständen $\{3, 6, 9\}$ und $req2$ mit dem Zielzustand $\{10\}$. Der Startzustand 1 ist unkontrollierbar und hat zwei ausgehende Transitionen, die zu den Zuständen 2 und 3 führen. An dieser Stelle müssen also parallel verlaufende Pfade beachtet werden. Eine gültige Lösung muss auf jeden Fall die Zustände 6 und 10 enthalten. Zustand 10 ist der einzige Zielzustand für $req2$. Zustand 6 ist einer von zwei Zielzuständen für $req1$, die von Zustand 2 aus erreicht werden können. Jedoch schließt sich der andere Zustand, 9, gegenseitig mit dem $req2$ erfüllenden Zustand 10 aus. Die im Graph enthaltenen Pfade sind in Tabelle 4.3 aufgelistet. Hier gibt es drei Gruppen von Pfaden, jeweils eine

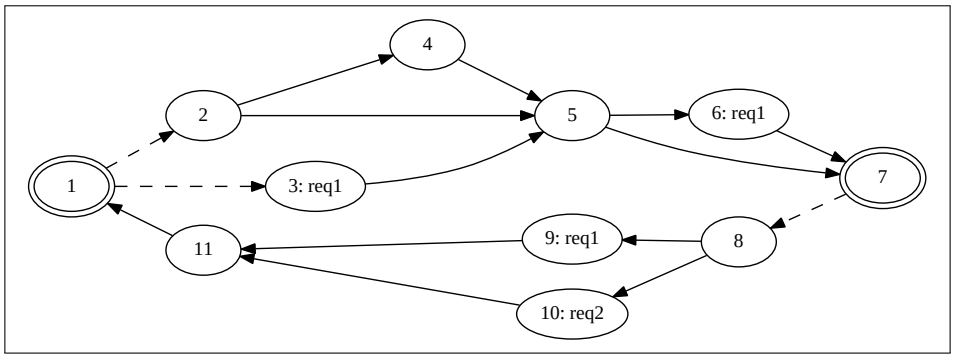

Abbildung 4.2: Spielgraph für Beispieldurchführung

Zustand	**Ziel**	**Pfade**	**Erf. Bed.**
1	7	$p1 : (1 \to 3 \to 5 \to 7)$	$\{req1\}$
		$p2 : (1 \to 3 \to 5 \to 6 \to 7)$	$\{req1\}$
		$q1 : (1 \to 2 \to 5 \to 7)$	$\emptyset$
		$q2 : (1 \to 2 \to 5 \to 6 \to 7)$	$\{req1\}$
		$q3 : (1 \to 2 \to 4 \to 5 \to 7)$	$\emptyset$
		$q4 : (1 \to 2 \to 4 \to 5 \to 6 \to 7)$	$\{req1\}$
7	1	$z1 : (7 \to 8 \to 9 \to 11 \to 1)$	$\{req1\}$
		$z2 : (7 \to 8 \to 10 \to 11 \to 1)$	$\{req2\}$

Tabelle 4.3: Pfade des Graphen aus Abbildung 4.2

pro parallel verlaufenden Pfad ausgehend von Zustand 1 ($p1$-$p2$ und $q1$-$q4$) sowie Pfade, die von Zustand 7 ausgehen ($z1$-$z2$).

Erste Iteration

In dieser Iteration gibt es noch keine verlierenden Pfade, sodass zunächst einfach versucht wird, die Büchi-Bedingung der primären Siegbedingung zu erfüllen. Da die Suche blind ist, also keinerlei Heuristiken zur Wahl des zuerst betrachteten Nachfolgezustands nutzt, ist nicht fest definiert, welche der möglichen Pfade dabei entstehen. Auch die Reihenfolge, in der die Nachfolger unkontrollierbarer Zuständen betrachtet werden, ist nicht festgelegt. Zu Demonstrationszwecken wird in diesem Beispiel deshalb zum Teil von unvorteilhaften Entscheidungen ausgegangen.

Vom Startzustand 1 aus wird zunächst die Funktion `reach` ausgeführt. Dabei muss für beide Nachfolger 2 und 3 ein Pfad zu einem Zielzustand gefunden werden, wodurch der initiale Suchstack in Abbildung 4.3 entsteht. $\uparrow$ drückt dort eine Vorwärtsanalyse aus, während $\downarrow$ für eine Rückwärtsanalyse stehen würde. An dieser Stelle wird, wie im Suchstack dargestellt, zunächst die Transition $(1, 3)$ betrachtet. Von dieser Transition aus wird die Vorwärtsanalyse fortgeführt, bis schließlich der Zielzustand 7 über den

↑ (1,3)
↑ (1,2)

Abbildung 4.3: Suchstack zu Beginn von `reach(1)`

↓ (5,7)
↑ (5,6)
↑ (1,2)

Abbildung 4.4: Suchstack nach Finden von Pfad $p1$

Pfad $p1$ gefunden wurde. Zustand 6 wurde dabei also nicht betrachtet, die Transition $(5, 6)$ befindet sich jedoch noch auf dem Suchstack. Anschließend wird die Transition $(5, 7)$ zur Rückwärtsanalyse vermerkt, wodurch der Suchstack nun Abbildung 4.4 entspricht. Dort ist zu erkennen, dass neben der Transition $(5, 7)$ noch die Transitionen $(5, 6)$ und $(1, 2)$ ausstehen. Erstere wird jedoch im ersten Schritt der Rückwärtsanalyse vom Suchstack entfernt, da von Zustand 5 aus nun schon ein fest definierter Pfad zu einem Zielzustand existiert. Über sie können bestenfalls nur alternative Wege zu Zielzuständen entstehen, die an dieser Stelle nicht relevant sind. Hier wird auch die *next*-Funktion für Zustand 5 definiert. Die Rückwärtsanalyse wird mit der Transition $(3, 5)$ fortgesetzt. Die Transition $(1, 3)$ kann jedoch noch nicht erfolgreich rückwärts analysiert werden, da über die unkontrollierbare Transition $(1, 2)$ noch kein Zielzustand erreicht wurde.

Diese Transition wird als nächstes betrachtet, da sie die einzig verbleibende auf dem Suchstack ist. Zunächst verläuft die Vorwärtsanalyse wie gehabt. An dieser Stelle sei die Transition $(2, 5)$ gewählt, während $(2, 4)$ weiterhin auf dem Suchstack verbleibt, falls die Suche für den anderen Pfad ergebnislos endet. An Zustand 5 ist die *next*-Funktion schon definiert, sodass hier die Funktion `completePath` aufgerufen wird, um der *next*-Funktion zu dem schon gefundenen Zielzustand zu folgen. Dabei entsteht der Pfad $q1$. Da kein verlierender Pfad entstand, wird die Rückwärtsanalyse der Transition $(2, 5)$ initiiert. Während dieser wird die Transition $(2, 4)$ vom Stack entfernt. Die Rückwärtsanalyse der Transition $(1, 2)$ ist erfolgreich, da über alle ausgehenden Transitionen ein Zielzustand gefunden wurde. Somit ist die Suche für Zustand 1 erfolgreich abgeschlossen.

Anschließend muss `reach` auch für den Zustand 7 ausgeführt werden. Dies verläuft weitestgehend ähnlich, jedoch ohne parallele Pfade durch unkontrollierbar Transitionen. An dieser Stelle sei an Zustand 8 die Transition $(8, 10)$ gewählt, sodass der Pfad $z2$ entsteht. Da nun jeder Zielzustand das Erreichen eines Zielzustands garantieren kann, ist die OTF-Suche zunächst erfolgreich und der erste Lösungskandidat gefunden, welcher in Abbildung 4.5 dargestellt ist. Transitionen des Kandidaten sind farblich hervorgehoben. Graue Zustände wurden bisher noch nicht erstellt.

In diesem Kandidaten gibt es zwei Zyklen. Der Zyklus $c1$ entsteht durch Konkatenation der Pfade $p1$ und $z2$, der Zyklus $c2$ durch Konkatenation der

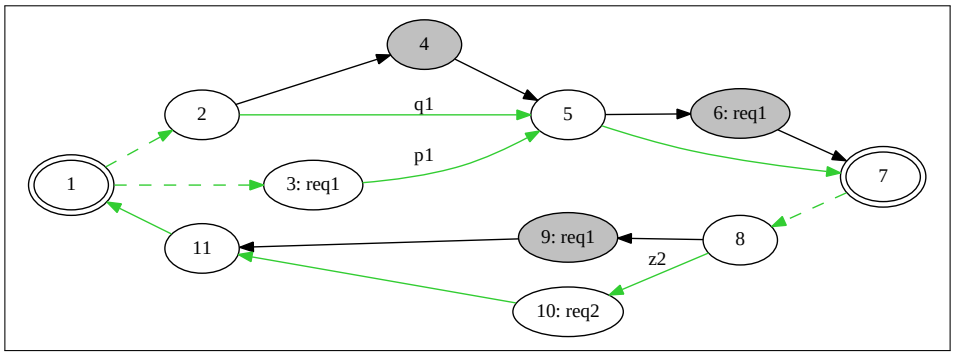

Abbildung 4.5: Erster Lösungskandidat des Beispiels aus Abbildung 4.2

Pfade $q1$ und $z2$. Von diesen Zyklen enthält jedoch nur $c1$ einen Zielzustand der Siegbedingung $req1$, weshalb $c2$ ein verlierender Zyklus ist. In beiden Pfaden von $c2$ gibt es noch alternative kontrollierbare Nachfolger. Dies wäre etwa Zustand 9 in $z2$ oder Zustand 4 in $q1$. Es werden deshalb zwei neue Konfigurationen erzeugt: Eine, in der $q1$ als verlierend angesehen wird und eine in der $z2$ verliert. Eine weitere mögliche Konfiguration wäre die, in der sowohl $q1$ als auch $z2$ als verlierend angesehen werden. Diese wird jedoch, falls nötig, später aus den anderen beiden Konfigurationen abgeleitet.

In jeder verlierenden Konfiguration werden die jeweiligen Pfade aus der *paths*-Funktion entfernt. Dabei wird ggf. auch die *next*-Funktion für den jeweiligen Zustand zurückgesetzt, falls dort keine weiteren Pfade hinterlegt sind. In der Konfiguration, wo $z2$ verliert, geschieht dies bei jedem Zustand des Pfades. Für $q1$ geschieht dies nur bei den Zustand 2, da die restlichen Zustände mit dem Pfad $p1$ geteilt werden. Welche Konfiguration zuerst betrachtet wird ist wieder nicht festgelegt, weshalb zunächst mit der ungünstigeren Konfiguration fortgefahren wird, in der $z2$ verliert.

z2 verliert

Hier muss `reach` für den Zustand 7 wiederholt werden. Falls bei der Vorwärtsanalyse wieder der Pfad $z2$ gefunden wird, so wird dort keine Rückwärtsanalyse angestoßen. Die Suche wäre erfolglos, falls der Suchstack jetzt leer wäre. Es gibt jedoch noch die in der ersten Iteration nicht betrachtete Transition $(8, 9)$. Über sie kann der Pfad $z1$ komplettiert werden. Damit können trotz des verlierenden Pfades $z2$ wieder alle Zielzustände das Erreichen eines Zielzustands garantieren, weshalb dies ein weiterer Lösungskandidat ist.

In diesem gibt es die Zyklen $c3$ und $c4$, die durch Konkatenation der Pfade $q1$ und $z1$ bzw. $p1$ und $z1$ entstehen. Jedoch enthält keiner dieser Zyklen einen Zielzustand von $req2$. Alle enthaltenen Pfade haben alternative kontrollierbare Nachfolger, wodurch an dieser Stelle viele neue

Konfigurationen entstehen können. Da jedoch der einzige Zielzustand von $req2$ auf einem verlierenden Pfad liegt führen sie alle zum selben Ergebnis. Repräsentativ für diese Fälle wird deshalb nur die Konfiguration betrachtet, in der $z1$ und $z2$ verlierend sind.

z1 und z2 verlieren

Wieder muss `reach` für den Zustand 7 ausgeführt werden. Unabhängig davon, welcher Pfad zuerst genommen wird, wird dieser beim Zielzustand als verlierend erkannt. Die verbleibende Transition von Zustand 8 führt zum selben Ergebnis. Damit ist der Suchstack erschöpft, jedoch noch kein Zielzustannd gefunden. Zustand 7 verliert deshalb. Alle Pfade, die zu Zustand 7 führen sind deshalb ebenfalls verlierend. Dies führt dazu, dass auch Zustand 1 neu evaluiert werden muss. Da Zustand 7 der einzig erreichbare Zielzustand war, wird allerdings auch dort kein Nachfolger gefunden, und somit verliert Zustand 1. Da nun der Startzustand verliert sind alle Möglichkeiten erschöpft, und es werden keine neuen Konfigurationen aus dieser Konfiguration abgeleitet.

q1 verliert

In dieser Konfiguration ist der Pfad $p1$ durch die *next*-Funktion weiter vorgegeben. Der Pfad $q1$ wird jedoch verworfen, da er als verlierend bekannt ist. An dieser Stelle wird mit der vorher nicht betrachteten Transition $(2, 4)$ fortgefahren, wodurch mittels der *next*-Funktion der Pfad $q3$ entsteht. Da dieser nicht verliert ist somit ein weiterer Kandidat gefunden, jedoch gibt es auch hier wieder einen verlierenden Zyklus $c5$, der durch die Konkatenation von $q3$ und $z2$ entsteht. In $c5$ existieren wieder in beiden Pfaden alternative Entscheidungen, sodass zwei neue Konfigurationen abgeleitet werden können. Die Konfiguration, in der $z2$ verliert entspricht jedoch weitestgehend einer vorher beschriebenen Konfiguration, sodass an dieser Stelle nur die Konfiguration betrachtet wird, in der $q3$ verliert.

q1 und q3 verlieren

Hier wird zunächst wieder der Pfad $p1$ gefunden, der weiterhin durch die *next*-Funktion vorgegeben ist. Die Pfade $q1$ und $q3$ sind jedoch beide verlierend, sodass für diesen Zweig der Suchstack erschöpft ist, bevor eine Rückwärtsanalyse zum Startzustand möglich ist. Nun tritt der Fall ein, in dem die Menge *losesAt* betrachtet wird. Alle unter Berücksichtigung der *next*-Funktion möglichen Pfade, die über die unkontrollierbare Transition $(1, 2)$ führen, sind verlieren. Daraus folgt, dass das über *next* definierte Suffix sich gegenseitig mit einer gültigen Lösung ausschließt - in diesem Fall ausgehend von Zustand 5. Alle Pfade, die das selbe Suffix teilen, werden

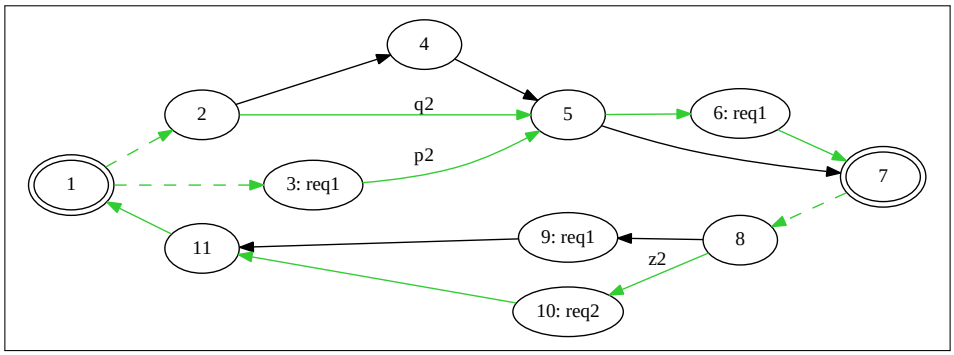

Abbildung 4.6: Lösung des Beispiels aus Abbildung 4.2

deshalb ebenfalls als verlierend angesehen. Deshalb wird nun $p1$ als verlierend vermerkt und `reach` für den Zustand 1 nochmals aufgerufen.

Wird nun der Pfad $p1$ gefunden, so wird er verworfen, da er verliert. Die nächste Alternative auf dem Suchstack ist die Transition $(5, 6)$, wodurch der Pfad $p2$ entsteht. Für den anderen Zweig ergibt sich deshalb entweder der Pfad $q2$ oder der Pfad $q4$. Für dieses Beispiel sei es der Pfad $q2$. Dies kann zum Startzustand zurückpropagiert werden, sodass ein neuer Kandidat gefunden ist. In diesem gibt es die Zyklen $c6$ und $c7$, die jeweils eine Konkatenation von $p2$ und $z2$ bzw. $q2$ und $z2$ sind. Auf beiden Zyklen sind Zielzustände aller Siegbedingungen vorhanden, sodass dies schlussendlich ein gewinnender Kandidat ist.

Ergebnis

In Abbildung 4.6 ist der gewinnende Kandidat abgebildet. Die gewählten Transitionen sind wieder grün hervorgehoben. Bei der Beispielrechnung wurde der komplette Zustandsraum erzeugt, jedoch wäre dies durch bessere Entscheidungen bei der Suche vermeidbar gewesen. Ebenso sind manche Konfigurationen vielversprechender als andere. In diesem Beispiel haben etwa alle Konfigurationen, die $z2$ als verlierenden Pfad enthalten, keine gültige Strategie. An dieser Stelle existiert mit geeigneten Heuristiken also noch Verbesserungspotential.

4.5 Komplexität

Eine triviale Abschätzung der Laufzeitkomplexität gestaltet sich schwierig, da sie in hohem Maße von der Struktur des Graphen und der Siegbedingungen abhängt. Begrenzend für die Laufzeit ist die Anzahl möglicher Konfigurationen, da die drei Phasen des Algorithmus schlimmstenfalls für jede erzeugte Konfiguration wiederholt werden müssen.

Die maximale Anzahl erzeugter Konfigurationen einer Iteration ergibt sich aus der Anzahl und dem Aufbau der verlierenden Zyklen. Ein einfacher Zyklus kann maximal $n + 1$ Zustände enthalten, bei doppelter Zählung des ersten bzw. letzten Zustands. In diesem Fall würde er über den kompletten Graph gehen. Die mögliche Anzahl verlierender Pfade ergibt sich aus der Anzahl Zielzustände der primären Siegbedingung auf dem Zyklus, da ein Pfad jeweils zwei Zielzustände verbindet. Es kann daher maximal n Pfade auf einem Zyklus geben, die als verlierend in Betracht kommen. In diesem Fall wäre die primäre Siegbedingung jedoch trivial erfüllbar: Jeder Zustand des Graphen erfüllt sie, wodurch sie redundant ist. Eine der sekundären Siegbedingungen könnte ihren Platz einnehmen. Eine realistische Annahme ist deshalb, dass ein Zyklus maximal $k < n$ Zielzustände enthält, und somit maximal k mögliche verlierende Pfade. Da es exponentiell viele Zyklen geben kann, und jeder Zyklus bis zu $k = O(n)$ mögliche verlierende Pfade enthält, können somit auch exponentiell viele Konfigurationen entstehen.

Bei diesen Abschätzungen wird jedoch vom Worst-Case ausgegangen, einem vollständigen Graphen. Aus szenariobasierten Spezifikationen erstellte Spielgraphen sind, wie bereits in Abschnitt 4.3.2 erwähnt, generell sehr dünn besetzt, wodurch die mögliche Anzahl Zyklen beträchtlich reduziert wird. Darüber hinaus ist jeder Zielzustand der primären Siegbedingung (d.h. der Specification-Szenarien in ScenarioTools) unkontrollierbar, da explizit gefordert ist, dass das System auf Umgebungsnachrichten wartet. Wenn jeder Zustand unkontrollierbar ist, sind an keinem Zustand alternative Folgezustände für das Systems möglich. In diesem Fall verliert der Zyklus als Ganzes, ohne weitere Konfigurationen zu erzeugen. Die maximale Anzahl in Betracht kommender Pfade auf einem Zyklus ist $\frac{n}{2}$, falls jeder zweite Zustand des Zyklus ein Zielzustand ist. Dies ist jedoch aufgrund der durchschnittlichen Knotengrade ein theoretisches Maximum. In einer Evaluation, die in Kapitel 6 dieser Arbeit diskutiert wird, hat sich gezeigt, dass die Laufzeit des Algorithmus für einen realistischen Anwendungsfall niedriger ist, als bei Verfahren, die den kompletten Zustandsraum erstellen.

In der beschriebenen Form des Algorithmus kann es passieren, dass ein Pfad wiederholt in verschiedenen Konfigurationen als verlierend erkannt werden muss. Eine mögliche Alternative wäre, nicht alle möglichen Kombinationen von verlierenden Teilpfaden aller verlierenden Zyklen als neue Konfigurationen aufzunehmen, sondern sich stattdessen auf einen Zyklus auf einmal zu beschränken. So könnten potentiell viele Konfigurationen auf einmal verworfen werden, wenn ein verlierendes Teilstück zu einer verlierenden Gesamtlösung führt.

4.6 Korrektheit

Nun soll begründet werden, dass der Algorithmus korrekt arbeitet. Dazu zunächst eine vorbereitende Feststellung.

Lemma 4.1. *Es ist ausreichend, die Zyklen im Graphen zu prüfen.*

Beweis. Falls ein Zustand nicht Teil eines Zykluses ist, so kann nur eine der folgenden Möglichkeiten gelten:

1. er ist Vorgänger eines Zustands, der Teil eines Zyklus ist
2. er ist kein Vorgänger eines Zustands, der Teil eines Zyklus ist

Im zweiten Fall ist die Büchi-Bedingung verletzt, da mit endlichem Zustandsraum ohne Zyklen nur endlich lange Pfade existieren. Damit wäre der Ausgangszustand verlierend. Während der Suche könnte vom Ausgangszustand aus mindestens ein Zielzustand auf den Weg keinen Nachfolger finden, wodurch er als verlierend eingestuft wird und dieser Status an die Vorgänger propagiert wird. Somit wird der Ausgangszustand korrekt als verlierend erkannt.

Im ersten Fall sind mögliche oder fehlende Zielzustände der sekundären Siegbedingungen vor den Zyklen vernachlässigbar. Die Büchi-Bedingung besagt, dass jede Siegbedingung unendlich oft erfüllt werden muss. Somit müssen sie zwingend auch auf den jeweiligen Zyklen erfüllt werden, weshalb alle Zustände außerhalb von Zyklen während der Verifikation vernachlässigt werden können. □

Mit dieser Feststellung kann nun die Korrektheit gezeigt werden.

Lemma 4.2. *Wenn der Algorithmus eine positive Antwort liefert, dann ist sie gewinnend für das System.*

Beweis. Die primäre Siegbedingung ist erfüllt, wie sich analog zum einfachen OTF-Algorithmus zeigen lässt. Verbleibend sind die sekundären Siegbedingungen. Bevor ein Lösungs-Kandidat akzeptiert wird, werden alle Zyklen im Kandidaten geprüft. Jeder Zyklus enthält mindestens einen Zielzustand der primären Siegbedingung. Ein Zyklus wird akzeptiert, falls auf ihm von jeder weiteren Siegbedingung mindestens ein Zielzustand vorkommt (Büchi-Bedingungen sind erfüllt) oder falls für mindestens eine Assumption-Bedingung kein Zielzustand vorkommt (Assumption ist verletzt, wodurch das System nicht zu seinen Siegbedingungen verpflichtet ist). Gilt dies für jeden Zyklus des Kandidaten, so kann das System nach Lemma 4.1 von jedem Zustand aus einen gewinnenden Zyklus erreichen, unabhängig von den Entscheidungen der Umgebung. Somit ist das Ergebnis gewinnend für das System. □

Lemma 4.3. *Wenn der Algorithmus eine negative Antwort liefert, dann existiert keine Siegstrategie für das System, in der für die primäre Siegbedingung die Büchi-Bedingung erfüllt ist und für die sekundären Siegbedingungen die GR(1)-Bedingungen erfüllt sind.*

Beweis. Es gibt zwei Arten, auf die eine Negativantwort auftreten kann:

1. In der ersten Iteration kann für mindestens einen Zustand keine Menge an Folgezielzuständen gefunden werden, für die das Erfüllen der primären Siegbedingung garantiert werden kann.

2. Das Erfüllen der primären Siegbedingung kann garantiert werden, jedoch konnten die Sekundären Siegbedingungen nicht erfüllt werden.

Für den ersten Fall ist es simpel. Die Büchi-Bedingung der primären Siegbedingung ist verletzt. Dieser Fall ist analog zum einfachen OTF-Algorithmus.

Im zweiten Fall existiert im ersten Kandidaten mindestens ein Zyklus, auf dem mindestens eine sekundäre Siegbedingung nicht erfüllt ist. Von diesem Zyklus kann mindestens ein Teilpfad nicht zur Siegstrategie des Systems gehören. Welcher Teilpfad es ist, ist an dieser Stelle jedoch unbekannt, weshalb alle Möglichkeiten betrachtet werden. Der Zyklus wird analysiert, und es werden alle Pfade zwischen zwei primären Zielzuständen gesucht, in denen sich das System noch anders entscheiden könnte. Für jeden so gefundenen Pfad wird eine neue Konfiguration erstellt, in der dieser Pfad als verlierend betrachtet wird. Es werden also alle kontrollierbaren Möglichkeiten durchprobiert, wegen denen dieser Zyklus verlieren könnte. Es kann passieren, dass ein Zustand nun keine passenden Nachfolger mehr findet. Dann verliert er, und seine Vorgänger werden erneut analysiert. Falls auf dem Zyklus kein Pfad existiert, auf dem das System eine andere Entscheidung treffen kann, dann kann die Umgebung von jedem Zustand des Pfades aus gewinnen, wodurch sie alle korrekt als verlierend betrachtet werden. Alle Zielzustände, die Vorgänger eines Zustands dieses Zyklus sind, werden in diesem Fall neu analysiert. Falls mehrere verlierende Zyklen existieren, werden alle möglichen Kombinationen von verlierenden Pfaden betrachtet.

Insgesamt werden alle möglichen Kombinationen von verlierenden Pfaden durchgespielt, und aus diesen verlierenden Pfaden können sich pro Konfiguration noch weitere ergeben, die von einem gemeinsamen Suffix abhängen. Die Umgebung gewinnt nur, falls für keine mögliche Kombination eine Menge gewinnender Zyklen für das System existiert. □

Daraus folgt, dass der Algorithmus korrekt ist und bis auf die besprochenen Einschränkungen auch vollständig ist.

Anmerkung. Der Algorithmus ist wie besprochen nicht vollständig für GR(1)-Spiele. Für die primäre Siegbedingung wird während der OTF-Suche nicht auf nicht-akzeptierende Assumption-Zyklen geprüft, weshalb für diese die Büchi-Bedingung erfüllt sein muss. Es könnte sein, dass das System die Ausführung zwischen zwei Zielzuständen in einen nicht-akzeptierenden Assumption-Zyklus zwingen kann, der keinen Zielzustand enthält. Hier werden jedoch solche Zyklen von vornherein verworfen.

4.7 Implementierung

Bei der Implementierung wurde versucht, schon während der Suche die Zyklenerkennung durchzuführen. Dies hat sich jedoch als komplex und fehleranfällig herausgestellt. Jeder Zielzustand verwaltet eine Liste von Vorgänger-Zielzuständen. Hier werden auch transitive Vorgänger aufgenommen. Dazu registriert sich ein Zustand bei allen Nachfolgern, von denen er abhängt. Anschließend werden ausgehend von diesen direkten Vorgängern transitiv die Vorgänger der Vorgänger berechnet. Falls während der Suche ein Zielzustand gefunden wird, der bereits Vorgänger des suchenden Zustands ist, dann wird dieses zyklenschließende Paar gesondert vermerkt. Der suchende Zustand trägt sich hier nicht als Vorgänger ein.

Bei der Verifikation wird jedes zyklenschließende Paar (*closing*, *starting*) betrachtet. Der Zustand *closing* ist der letzte Zustand im Zyklus. Er hat den Zustand *starting* als notwendigen Nachfolger, jedoch ist *starting* auch ein transitiver Vorgänger von *closing*. Ausgehend von allen Pfaden, die *closing* mit *starting* verbinden, werden rückwärts die Vorgänger verfolgt, solange diese ebenfalls *starting* als Vorgänger haben. Jeder so entstehende Zyklus, der *starting* erreicht, wird anschließend wie beschrieben auf Akzeptanz überprüft. Die während der Suche erstellten Pfade merken sich direkt, welche Bedingungen auf ihnen erfüllt sind, sodass in der Verifikation nicht nochmals über die Pfade iteriert werden muss.

Die Verwaltung von Konfigurationen wurde simpel gehalten. Jede Konfiguration enthält den kompletten Zustand des Algorithmus, mit Ausnahme des Zustandsgraphen. Dazu zählen gefundene Zielzustände, verlierende Zustände, die *dep*-, *next*- und *paths*-Funktionen sowie verlierende Pfade und Daten zur Verwaltung der Zyklen. Eine Konfiguration belegt deshalb relativ viel Speicher, was insbesondere bei einer größeren Anzahl Konfigurationen problematisch sein könnte. Eine Verbesserung der Speichernutzung wäre möglich, indem eine Konfiguration etwa nur Unterschiede zur ihr übergeordneten Konfiguration speichert. Dies würde sich jedoch zu Lasten der Laufzeit auswirken. Hier ist ein angemessener Tradeoff nötig.

Bei diesem Ansatz gibt es einen hohen Verwaltungsaufwand. Das Neuberechnen der Vorgänger kann eine teure Operation sein. Wenn einzelne Pfade als verlierend markiert werden, wird ein Zyklus aufgebrochen, sodass *starting* nicht mehr zwangsläufig Vorgänger von *closing* ist. In diesen Fällen muss *closing* nachträglich als Vorgänger von *starting* eingetragen werden, sodass bei der nächsten Komplettierung des Zyklus ein neues zyklenschließendes Paar entsteht.

Kapitel 5

Attractor-Algorithmus

Die bisher betrachteten Algorithmen haben alle das Ziel, unter gewissen Umständen nicht den kompletten Zustandsraum zu erstellen. Dies wirkt sich mitunter positiv auf die Laufzeit aus, jedoch hat es auch zur Folge, dass ihnen eine ganzheitliche Sicht auf den Zustandsraum verwehrt bleibt. Darüber hinaus führten sie zu teils sehr komplexem Vorgehen, was fehleranfällig ist und die Wartbarkeit erschwert. Mit dem Wissen, welche Zustände es im gesamten Zustandsraum gibt und für welche Siegbedingungen sie gewinnend sind, lassen sich deutlich wartbarere Algorithmen formulieren, die darüber hinaus die gesamtheitliche Sicht nutzen können, um Strategien zu berechnen, bei denen die komplexeren OTF-Algorithmen versagen.

In diesem Kapitel wird ein solcher Algorithmus von Chatterjee et al. [3] für GBG und GR(1)-Spiele betrachtet. Er besitzt Eigenschaften, die ihn zu einer lohnenswerten Erweiterung für ScenarioTools machen: Er ist vollständig für GR(1)-Spiele, woraus direkt folgt, dass er Strategien mit Gedächtnis berechnen kann. Darüber hinaus hat er eine bewiesen effiziente Laufzeitschranke, die unter bestimmten Annahmen sogar optimal ist. Er eignet sich deshalb primär als Alternative für Spezifikationen, bei denen die bisherigen OTF-Algorithmen nicht genutzt werden können. Weiterhin wird eine neue Erweiterung des Algorithmus vorgestellt, mit der eine angenommene *Fairness* der Umgebung umgesetzt werden kann. Durch diese Erweiterung kann realitätsfernem Verhalten entgegengewirkt werden.

Das Kapitel ist wie folgt aufgebaut: Zunächst werden in Abschnitt 5.1 notwendige Definitionen und die Funktionsweise der implementierten Algorithmen erläutert. Anschließend wird in Abschnitt 5.2 ein Verfahren beschriebenen, mit dem aus einem Spielgraphen und einer Siegstrategie mit Gedächtnis ein neuer Graph erzeugt werden kann, in dem die einzig existierende Siegstrategie gedächtnislos ist. In Abschnitt 5.3 wird die Fairness-Erweiterung beschrieben. Abschließend werden in Abschnitt 5.4 einige Details zur Implementierung in ScenarioTools besprochen.

5.1 Funktionsweise

Die Herangehensweise unterscheidet sich grundsätzlich von den vorgestellten OTF-Algorithmen, die versuchen, von einem Zustand aus *vorwärts* das Erreichen weiterer Zielzustände zu garantieren. Hier wird ausgehend von allen Zielzuständen *rückwärts* berechnet, für welche weiteren Zustände der Spieler diese unabhängig vom Gegenspieler erreichen kann. Dies sind die *Attractor* der Zielzustände.

Formal sind die Attractor eines Spielers von einer Zustandsmenge U alle Zustände, von denen der Spieler unabhängig von den Entscheidungen des Gegenspielers das Erreichen eines Zustands aus U garantieren kann [24]. Sie können durch eine Rückwärtssuche ausgehend von den Zuständen in U berechnet werden. Zunächst wird die Menge der Attractor auf die Menge U gesetzt, da jeder dieser Zustände trivialerweise in null Schritten erreichbar ist. Anschließen werden iterativ alle Zustände hinzugefügt, die entweder eine vom Spieler kontrollierbare Transition zu einem Attractor haben, oder von denen aus alle vom Gegenspieler kontrollierten Transitionen zu einem Attractor führen. Die Menge der Attractor kann in $O(m)$ für die Anzahl m der Transitionen berechnet werden. Eingabe für die Attractor-Berechnung sind meistens Zielzustände, mit dem Ziel, alle Zustände zu finden, von denen der Spieler eine Strategie zum Erreichen eines Zielzustands besitzt. Da für die korrekte Berechnung alle Zielzustände bekannt sein müssen, kann diese Art von Algorithmen nicht On-The-Fly ausgeführt werden.

Eine *geschlossene Menge* für einen Spieler p ist eine Menge von Zuständen U, für die gilt, dass für alle von p kontrollierten Transitionen t, die von Zuständen in U ausgehen, gilt, dass $target(t) \in U$. Dem Spieler p ist es also nicht möglich, allein durch eigene Entscheidungen die Menge U zu verlassen. Für die Attractor von p gilt, dass deren Komplement eine geschlossene Menge von p ist [24]. Ein *Dominion* eines Spielers p ist eine Menge von Zuständen U, für die eine gewinnende Strategie für p existiert, bei der nur Zustände aus U besucht werden [13]. Ein Dominion von Spieler p ist eine geschlossene Menge des Gegenspielers. Die Attractor eines Dominions von Spieler p sind ebenfalls wieder ein Spieler p Dominion.

Von Chatterjee et al. [3] werden zwei GBG-Algorithmen vorgestellt: Ein einfacher Algorithmus `GenBuchiGameBasic` mit Worst-Case-Laufzeit $O(k \cdot n^3)$, wobei k die Anzahl Büchi-Bedingungen und n die Anzahl Zustände ist, und eine optimierte Version `GenBuchiGame` mit Laufzeit $O(k \cdot n^2)$. Beide Algorithmen basieren auf der Idee, Dominions der Umgebung zu berechnen und diese aus dem Spiel zu entfernen. In `GenBuchiGame` wird eine *Hierarchische Graph-Dekomposition* genutzt, welche für $1 \leq i \leq \lceil \log_2 n \rceil$ die Graphen $G_i = (V, E_i)$ definiert. E_i enthält die Transitionen aller Zustände mit ausgehendem Grad $\leq 2^i$ und die ersten 2^i eingehenden Transitionen aller Zustände. Ziel ist es, idealerweise schon auf kleineren Graphen G_i Umgebungs-Dominions zu finden und somit bestenfalls nicht für jeden Zu-

Algorithmus 10 : `GenBuchiGameBasic` von Chatterjee et al. [3]

Input : Spielgraph $G = (V, E)$ und Büchi-Bedingungen $\bigwedge_{l=1}^{k} T_l$
Output : Gewinnende Zustandsmenge V für das System

$G^1 \leftarrow G$
$\{T^1\} \leftarrow \{T_l\}$
$j \leftarrow 0$
repeat
  $j \leftarrow j + 1$
  for $1 \leq l \leq k$ **do**
    $Y_l^j \leftarrow Attr_S(G^j, T_l^j)$
    $S^j \leftarrow V^j \setminus Y_l^j$
    if $S^j \neq \emptyset$ **then break**;
  end
  $D^j \leftarrow Attr_E(G^j, S^j)$
  $G^{j+1} \leftarrow G^j \setminus D^j$
  $\{T_l^{j+1}\} \leftarrow \{T_l^j \setminus D^j\}$
until $D^j = \emptyset$;
return V^j

stand den gesamten Graphen betrachten zu müssen. Es wurde gezeigt, dass dieser Algorithmus optimal ist für vollständige Graphen unter der Annahme, dass kein effizienterer Algorithmus für die kombinatorische Matrixmultiplikation als bisher bekannt existiert. Es wurde auch gezeigt, dass die einfache Variante des Algorithmus unter vergleichbaren Annahmen optimal für dünne Graphen ist. Szenario-Graphen sind generell sehr dünn, der maximale Zustandsgrad wäre schon für sehr kleine i erreicht. Aus diesem Grund ist anzunehmen, dass aus der iterativen Prüfung auf zunehmend größeren Transitionsmengen kein nennenswerter Geschwindigkeitsgewinn entsteht, weshalb in dieser Arbeit nur der einfache Algorithmus implementiert wurde.

Weiterhin werden von Chatterjee et al. zwei GR(1)-Algorithmen vorgestellt: ein einfacher Algorithmus `GR(1)GameBasic`, der in $O(k_1 \cdot k_2 \cdot n^3)$ arbeitet, wobei k_1 und k_2 jeweils die Anzahl Siegbedingungen der Antezedens bzw. Konsequenz sind, und ein erweiterter Algorithmus `GR(1)Game` der in $O(k_1 \cdot k_2 \cdot n^{2.5})$ arbeitet. Hier wurde aus dem selben Grund die einfache Variante gewählt, da der erweiterte Algorithmus nur für die Anzahl Transitionen $m > n^{1.5}$ eine Verbesserung darstellt.

Der GBG-Algorithmus ist in Algorithmus 10 dargestellt und funktioniert wie folgt: Für jede Siegbedingung des Systems werden wie oben beschrieben die Attractor berechnet. Ist in einer dieser Mengen nicht der komplette Graph enthalten, so ist das Komplement der Attractor (S_j in Iteration j) ein Dominion für die Umgebung. Für diese Dominion werden nun aus Sicht der Umgebung die Attractor berechnet, was alle Zustände ergibt, von denen aus

die Umgebung das Verlieren des Systems garantieren kann. Diese Zustände werden anschließend aus dem Graphen und den Siegbedingungen entfernt. Das ganze wird wiederholt bis entweder alle Attractor dem kompletten verbleibenden Graphen entsprechen, oder der Graph leer ist. Optional kann die Suche auch abgebrochen werden, wenn ein ausgezeichneter Startzustand verliert. Assumptions können in diesem Spiel nicht betrachtet werden.

Im GR(1)-Algorithmus können auch Assumptions bedacht werden. Der Algorithmus verwendet den GBG-Algorithmus als Unterprogramm. Er funktioniert ähnlich wie der GBG-Algorithmus, der einzige Unterschied folgt nach der Attractor-Berechnung. Ist eine Attractor-Menge unvollständig, so wird ein neuer Graph mit nur den Zuständen aus dem Komplement dieser Menge erstellt. Auf diesem spielt nun die *Umgebung* gegen das System - sie versucht innerhalb der Assumptions zu bleiben, welche ihre Siegbedingungen für dieses Unter-Spiel sind. Gewinnt die Umgebung dieses Spiel, d.h. es gibt eine nicht-leere Menge von Zuständen auf denen sie garantieren kann, dass die Assumptions eingehalten werden, so sind diese Zustände das Dominion der Umgebung. Ansonsten wird die Attractor-Menge als vollständig angesehen, da die Assumptions verletzt sind und somit die GR(1)-Bedingung trivial erfüllt ist.

5.2 Strategie-Erstellung

Auf dem verbleibenden Graph kann das System nun von jedem Zustand aus das Erreichen aller Siegbedingungen garantieren. Dazu zunächst eine nötige Definition. Im Folgenden sei $l \oplus_k 1$ für $1 \leq l \leq k$ wie folgt definiert ($l \oplus_k j$ sei entsprechend die j-fache Hintereinanderausführung):

$$l \oplus_k 1 = \begin{cases} l + 1, \text{ falls } l < k \\ 1, \text{ falls } l = k \end{cases}$$

Ist der Graph leer oder der Startzustand nicht enthalten, so verliert das System. Andernfalls kann das System mittels folgender Strategie einen Sieg garantieren: Beginnend mit $l = 1$, folge der *Attractor-Strategie* zu einem Zielzustand der Siegbedingung g_l. Da jeder verbleibende Zustand Teil der Attractor ist, ist dies von jedem Zustand aus möglich. Bei Erreichen eines Zielzustands, setze $l \leftarrow l \oplus_k 1$ und wiederhole den Schritt.

Die Attractor-Strategie kann berechnet werden, indem ausgehend von den Zielzuständen einer Siegbedingung jeweils rückwärts der Nachfolgezustand festgelegt wird. Für alle direkten kontrollierbaren Vorgänger u eines Zielzustands v wird die ausgehende Transition (u, v) in die Strategie übernommen. Dies wird mit allen so neu-erreichten Zuständen wiederholt. Ist für einen Zustand schon ein Nachfolger festgelegt, so wird dieser nicht überschrieben. Bei kontrollierbaren Zuständen wird so je ein Nachfolger festgelegt, der zielgerichtet zu einem Zielzustand führt. Bei unkontrollierbaren

Zuständen müssen wie üblich alle Folgezustände als Nachfolger angenommen werden. In GR(1)-Spielen muss zusätzlich noch die Strategie bei verletzten Assumptions berechnet werden. Hierzu wird die selbe Berechnung mit der Menge Zuständen ausgeführt, von denen aus die Umgebung das Einhalten der Assumptions nicht garantieren kann. Dies sei im Folgenden als co-Büchi-Strategie bezeichnet. Für ein Gegenbeispiel wiederum wird das selbe Vorgehen auf den Zuständen aller Dominions ausgeführt. Hier ist zu beachten, dass vorher unkontrollierbare Transitionen nun vom Spieler der Strategie (der Umgebung) kontrolliert werden und umgekehrt.

Für diese Strategie ist, anders als bei den bisherigen Algorithmen, jedoch ein Gedächtnis nötig. Das System muss sich merken, welche Attractor-Strategie momentan verwendet wird. Aus Vergleichbarkeitsgründen wird der Spielgraph deshalb in einen neuen Graphen überführt, in dem vom Startzustand eine gedächtnislose Siegstrategie für das System existiert. In diesem Graph hat jeder kontrollierbare Zustand nur eine ausgehende Transition, weshalb nur eine Strategie existiert die sich direkt aus den kontrollierbaren Transitionen ergibt. Das Vorgehen wird für ein GR(1)-Spiel der Form $\bigwedge_{i=1}^{k_1} a_i \rightarrow \bigwedge_{i=1}^{k_2} g_i$ erklärt, ist jedoch ohne große Änderungen auch für ein GBG nutzbar.

Dazu wird der Index der aktuellen Attractor-Strategie k in jeden Zustand des Graphen $G' = (V', E')$ enkodiert, sodass die entstehenden Zustände V' eine Teilmenge von $V \times \{1, 2, ..., k_2\}$ sind. Ein Zustand (v, k) bedeute, dass sich G' momentan im Zustand $v \in V$ befindet, und der Attractor-Strategie für g_k folgt. Die Konstruktion des Graphen ist in Algorithmus 11 beschrieben und funktioniert wie folgt: Erstelle den Startzustand (v_1, i) und speichere ihn auf einem Stack. Die Variable i entspricht dem Index der ersten unerfüllten Siegbedingung des Startzustands. Ist der Startzustand für alle Siegbedingungen erfüllend, so kann ein beliebiger Index gewählt werden, etwa 1. Anschließend wird über den Stackinhalt iteriert.

Für jedes Element (v_i, k) des Stacks werden mittels der Attractor-Strategie von g_k die Nachfolger u_x bestimmt. Falls v_i ein Zustand ist, der in g_k aufgrund verletzter Assumptions gewinnt, so muss hier die entsprechende co-Büchi-Strategie genutzt werden. Für jeden der Nachfolger wird geprüft, ob er die Siegbedingung g_k und ggf. g_{k+1} etc. erfüllt. Sei j die erste k folgende Siegbedingung, die u_x nicht erfüllt. Erfüllt u_x jede Siegbedingung, so kann eine beliebige ausgewählt werden. Dafür würde sich eine Siegbedingung anbieten, für die schon ein Zustand erstellt wurde. Im Algorithmus wird jedoch die bisherige Siegbedingung weiterverwendet (nach $k_2 + 1$-maligem Durchlauf der entsprechenden Schleife). Existiert der Zustand (u_x, j), so wird eine Transition von (v_i, k) zu (u_x, j) erzeugt und mit dem nächsten Nachfolger von (v_i, k) fortgefahren. Andernfalls wird der Zustand zunächst erzeugt und auf dem Stack abgelegt. Das Vorgehen wird wiederholt bis keine Elemente auf dem Stack verbleiben.

Der so entstehende Graph hat schlimmstenfalls $|V| * k_2$ Zustände und

Algorithmus 11 : Erstellen einer gedächtnislosen Strategie für den Attractor-Algorithmus

Input : Ergebnis-Graph $G = (V, E)$ für GR(1)-Spiel $\bigwedge_{i=1}^{k_1} a_i \rightarrow \bigwedge_{i=1}^{k_2} g_i$
Result : Gedächtnisloser Automat $G' = (V', E')$

Erstelle Attractor-Strategien für alle s_i.
Erstelle Graph G' und Startzustand (v_1, i), wobei i der Index der ersten unerfüllten Siegbedingung von v_1 und sonst 1 ist.

Sei $stack = \{(v_1, i)\}$
while *stack ist nicht leer* **do**
  Entferne oberstes Element (v_i, k) von *stack*.
  Sei *strategy* die Strategie von Siegbedingung g_k.
  foreach *Nachfolger u von* v_i *in strategy* **do**
    for $0 \leq i \leq k_2$ **do**
      $j \leftarrow k \oplus_{k_2} i$
      if *u ist nicht gewinnend für* g_j*)* **then**
        break
      end
    end
    if (u, j) *existiert nicht in* G' **then**
      Erstelle (u, j) in G' und füge es zu *stack* hinzu.
    end
    Erstelle Transition in G' von (v_i, k) zu (u, j).
  end
end

return G'

arbeitet deterministisch und ohne Gedächtnis, da jeder kontrollierbare Zustand nur eine ausgehende Transition hat. Für praktische Anwendungen von Szenarien sind es durchschnittlich jedoch deutlich weniger, da die meisten Requirement-Szenarien nur wenige und meistens lokal beschränkte verlierende Zustände haben, sodass ein einzelner Zustand oft für mehrere Requirements gewinnend ist und daher nicht dupliziert werden muss.

In Abbildung 5.2 ist ein so entstehender Controller für das Beispiel in Abbildung 5.1 dargestellt, welches eine Strategie mit Gedächtnis benötigte. Dies wurde durch Duplizierung der Zustände 1 und 2 gelöst, wodurch nun nur noch ein einfacher Zyklus übrig ist, auf dem alle Siegbedingungen erfüllt sind. Die Zustände, durch die eine Siegbedingung erfüllt wird, sind wie bisher hervorgehoben. Dieser Graph hat nur 6 Zustände - deutlich weniger als der Worst-Case von 12 Zuständen. In vielen Fällen ist es sogar möglich, dass der entstehende Controller weniger Zustände als der verbleibende Graph aus dem

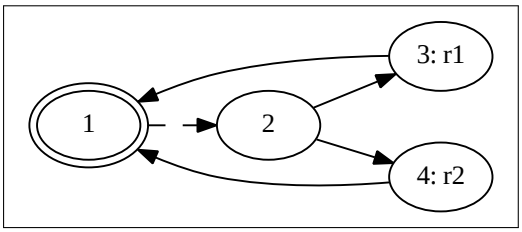

Abbildung 5.1: Beispiel für eine Spezifikation, die ein Gedächtnis benötigt (entspricht Abbildung 3.5)

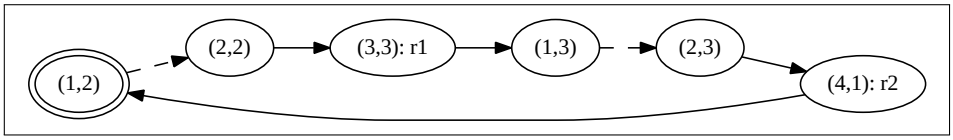

Abbildung 5.2: Entstehender Controller für das Beispiel in Abbildung 5.1

GR(1)-Algorithmus hat. Dieser Fall tritt ein, wenn mehrere Siegstrategien für das System existieren. Da in G' nur eine eine einzige Strategie existiert, fallen unter Umständen Zustände weg, die nicht Teil dieser Strategie sind. Es wird jedoch nicht zwangsläufig die optimale bzw. kleinste Strategie ausgewählt.

5.3 Fairness

In einem gewöhnlichen GBG nutzt die Umgebung jede Möglichkeit, einen Sieg des Systems zu verhindern. Dies kann je nach Anwendungszweck günstig oder ungünstig sein. Da GBGs unendliche Spiele sind, ist es möglich, dass die Umgebung bei wiederholtem Besuchen von Zuständen immer die selbe Entscheidung trifft, um den Gegenspieler so in einer Endlosschleife von verlierenden Zuständen festzuhalten. Je nachdem was die Umgebung modelliert ist es jedoch eine unwahrscheinliche Annahme, dass in unendlich vielen Zyklen immer die selbe Entscheidung getroffen wird. Man könnte hier annehmen, dass die Umgebung, soweit ein Zustand unendlich oft besucht wird, auch jede ihrer möglichen Entscheidungen unendlich oft wählt. Dieses Verhalten wird als *Fairness* bezeichnet werden, und wurde in ScenarioTools für den DFSBased-Algorithmus implementiert.

Abb. 5.3 zeigt ein Beispiel, in dem Fairness Sinn ergibt. Es ist angelehnt an das car-to-x-Beispiel aus ScenarioTools. Modelliert ist eine vereinfachte Situation, in der ein Auto auf einer zweispurigen Straße fährt. Es kann jederzeit die Spur wechseln, jedoch ist eine der Spuren durch ein Hindernis blockiert, wodurch es auf dieser Spur nichts weiter machen kann, als wieder auf die andere Spur zu wechseln. Die Aktionen des Autos sind Umgebungs-Events. Das System könnte beispielsweise eine auf der Straße installierte Kontrollstation sein. Die Umgebung könnte nun in Zustand 2 eine

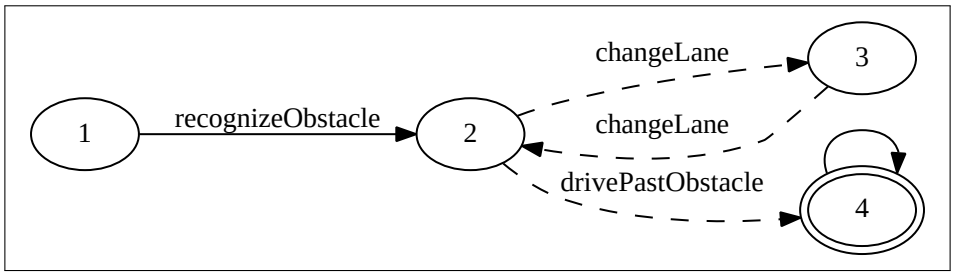

Abbildung 5.3: Beispiel für einen Spielgraphen, wo Fairness Sinn ergibt

Endlosschleife erzwingen, indem sie kontinuierlich zwischen beiden Spuren wechselt. Somit würde nie der Zielzustand erreicht werden, in dem das Auto das Hindernis passiert hat. Das System verliert aus diesem Grund. Dies ergibt in der modellierten Realität jedoch keinen Sinn. Es ist anzunehmen, dass der Fahrer des Autos früher oder später auf der richtigen Spur das Hindernis passiert. Geht man davon aus, dass sich die Umgebung fair verhält, so wird sie irgendwann die Endlosschleife zwischen den Zuständen 2 und 3 verlassen und in den gewinnende Zustand 4 übergehen.

Nun soll der Attractor-Algorithmus um Fairness erweitert werden. Betrachtet man Abbildung 5.3, so ist leicht zu erkennen an welcher Stelle das Beispiel mit dem bisherigen Algorithmus verliert: Zustand 2 kann nicht in die Attractor-Menge übernommen werden, da nur einer seiner beiden Nachfolger Attractor ist. Somit verlieren bis auf Zustand 4 alle Zustände, und da dort der Startzustand enthalten ist, verliert das System. Um den Aspekt der Fairness zu berücksichtigen muss die Berechnung der Attractor angepasst werden. Geht man davon aus, dass die Umgebung alle Möglichkeiten ausschöpft, so wäre es ausreichend, wenn *ein* Nachfolger von Zustand 2 die Siegbedingung erfüllt, was in diesem Fall erfüllt ist. Falls der Zustand wiederholt besucht wird, wird aufgrund dieser Annahme irgendwann der Nachfolger gewählt, in dem die Siegbedingung erfüllt ist, und die Büchi-Bedingung ist somit erfüllt. Die Attractor-Berechnung des Systems wird also auf die Berechnung der *Vorgänger* der Zielzustände abgeschwächt.

Falls nur ein Nachfolger eine Siegbedingung erfüllt, und der Zustand nur ein einziges Mal besucht wird, wird auf diese Art die Siegbedingung jedoch nicht erfüllt. In diesem Fall wären die anderen Zweige nicht Teil der Attractor, und somit Dominion für die Umgebung. Der Zustand würde deshalb als verlierend erkannt werden, da er selbst Attractor des Dominions ist. Fairness führt also nur zum Sieg, falls von allen Nachfolgern des Zustands ein Pfad zu einem Zielzustand existiert.

Im GR(1)-Algorithmus wird zudem ein GBG-Spiel aus Sicht der Umgebung geführt, in dem die Umgebung versucht innerhalb der Assumptions zu bleiben. Für diesen Fall muss die Attractor-Berechnung der Umgebung angepasst werden. Zustände sind für die Umgebung nur noch Attractor, wenn

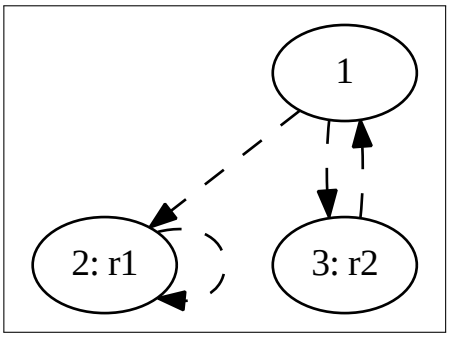

Abbildung 5.4: Die Zustände 1 und 3 werden endlich oft besucht

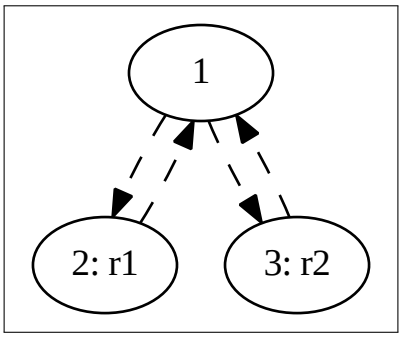

Abbildung 5.5: Alle Zustände werden unendlich oft besucht

alle kontrollierten sowie unkontrollierten Nachfolger bereits als Attractor bekannt sind. Bei der normalen Berechnungen würde ein kontrollierter Nachfolger reichen. Assumptions sind also nur erfüllt, wenn es keine Möglichkeit gibt, einen nicht-erfüllenden Zyklus zu bilden. Da die Umgebung alle Möglichkeiten durchprobiert, würde irgendwann dieser Zyklus besucht werden, was dann zum Sieg des Systems führt.

Zusammengefasst müssen zur Integration von Fairness folgende Änderungen vorgenommen werden:

- Ist der Spieler das System, so entfällt bei der Attractor-Berechnung die Bedingung, dass alle vom Gegenspieler kontrollierten Transitionen zu einem Attractor führen müssen. Ist es nun ausreichend, wenn mindestens eine beliebige Transition zu einem Attractor führt. Die Menge der Attractor entspricht dadurch der Menge der Vorgänger eines Zustands.
- Ist der Spieler die Umgebung, so entfällt die Bedingung, dass eine kontrollierte Transition zu einem Attractor führen muss. Stattdessen müssen alle Transitionen zu einem Attractor führen.

Für die Attractor-Berechnung des Gegenspielers sind keine Anpassungen nötig. Die angepassten Attractor-Berechnungen können sowohl im GBG als auch im GR(1)-Algorithmus genutzt werden.

Nun sollen die Fälle noch einmal detaillierter an Beispielen demonstriert werden. Abbildung 5.4 zeigt einen Graphen mit Zielzuständen von zwei Siegbedingungen $r1$ und $r2$, die jeweils in Zustand 2 bzw. 3 erfüllt sind. Von Zustand 1 aus sind beide Zustände erreichbar, jedoch hat nur Zustand 3 eine Transition, die zurück zu Zustand 1 führt. Von Zustand 2 aus sind also keine anderen Zustände mehr erreichbar. Das Beispiel wäre also verlierend, da nur eine Siegbedingung unendlich oft erreicht werden kann. Berechnet man in diesem Beispiel die Attractor unter Berücksichtigung der Fairness, so ergibt dies:

$Attr_S(r1) = Attr(\{2\}) = \{1, 2, 3\}$
$Attr_S(r2) = Attr(\{3\}) = \{1, 3\}$

Zustand 2 ist vom ganzen Graph aus erreichbar, Zustand 3 kann jedoch aufgrund der fehlenden Rückwärtstransition nicht von Zustand 2 erreicht werden. Da die Attractor von $r2$ nicht dem kompletten Graph entsprechen, werden wiederum die Umgebungs-Attractor des Komplements berechnet:

$Attr_U(\overline{Attr_S(r2)}) = Attr_U(\{2\}) = \{1, 2, 3\}$

Somit verliert das System, da die Umgebung trotz Fairness vom gesamten Graph aus verhindern kann, dass $r2$ erfüllt wird. In Abbildung 5.5 ist nun die Rückwärtstransition von Zustand 2 zu Zustand 1 enthalten, wodurch die Attractor beider Siegbedingungen dem kompletten Graphen entsprechen und somit keine Zustände verlieren. Dies entspricht auch dem gewünschten Ergebnis, denn solange die Umgebung wie angenommen jede ausgehende Transition von Zustand 1 unendlich oft wählt, werden beide Siegbedingungen unendlich oft erfüllt.

Lemma 5.1. *Wenn die Umgebung bei wiederholtem Besuch eines Zustands irgendwann alle Möglichkeiten durchgeht, dann ist von jedem Zustand des Ergebnis-Graphen eines GBGs jede Siegbedingung erfüllbar.*

Beweis. Angenommen es gäbe im Ergebnis einen Zustand u, von dem aus die Umgebung eine Entscheidung treffen kann, die bewirkt, dass eine Bedingung g_j vom nächsten Zustand aus nie wieder erfüllt werden kann. O.B.d.A. sei u unkontrollierbar. u hat daher mindestens einen Nachfolger v, von dem aus g_j nicht mehr erfüllt werden kann. Daraus folgt, dass v kein Vorgänger eines Zielzustands von g_j ist, und somit nicht in der Menge der Attractor ist. Also ist v in der Menge S_i einer Iteration i. Da u ein unkontrollierbarer Vorgänger von v ist, ist u ein Attractor für v. u gewinnt deshalb für die Umgebung, und kann somit nicht Teil des Ergebnis sein. □

5.4 Implementierung

Die Implementierung in ScenarioTools verlief weitestgehend problemlos, sodass hier nur einige interessante Aspekte angesprochen werden sollen. Da ScenarioTools für Algorithmen ausgelegt ist, die on-the-fly arbeiten, muss zunächst der Zustandsraum erzeugt werden. Dabei können auch direkt die Zielzustandsmengen berechnet werden. Vom Startzustand aus wird eine Tiefensuche durch den Szenario-Graphen gestartet. Für jeden erstmals gefundenen Zustand wird geprüft, ob er gewinnend für die Specification-Szenarien ist oder verlierend für Instanzen von Requirement- bzw. Assumption-Szenarien. Anschließend wird die Tiefensuche von diesem Zustand aus wiederholt. Die Menge der Zielzustände von Instanzen von Requirement- bzw. Assumption-Szenarien ergibt sich aus dem Komplement der verlierenden Zustände.

Während der Berechnungen wird eine Menge verlierender Zustände verwaltet. Transitionen von und zu diesen Zuständen werden bei Berechnungen von Attractorn und der Strategie-Erstellung ignoriert. Zustände, die als Dominion der Umgebung erkannt werden, werden zu dieser Menge verlierender Zustände hinzugefügt. Auf diese Art können Berechnungen auf den Graphen G^j durchgeführt werden, ohne diese explizit zu erstellen.

Die Partial-Order-Reduction-Implementierung des DFSBased-Algorithmus konnte ohne weitere Anpassungen übernommen werden. Diese wird direkt bei der Erstellung des Zustandsraum und der Berechnung der Zielzustände ausgeführt. Da dies der mit Abstand zeitaufwändigste Schritt ist, kann die Laufzeit dadurch teils beträchtlich reduziert werden kann.

Kapitel 6

Evaluation

Nun sollen die neu entwickelten bzw. umgesetzten Algorithmen mit den bestehenden Algorithmen in ScenarioTools verglichen werden. Zunächst wird in Abschnitt 6.1 geprüft, welche möglichen Spezialfälle von den jeweiligen Algorithmen behandelt werden. In Abschnitt 6.2 wird anschließend anhand eines umfangreichen Testfalls die Performance und der Speicherverbrauch der Algorithmen verglichen. Anschließend werden in Abschnitt 6.3 Auffälligkeiten der Evaluation diskutiert. Zuletzt werden in Abschnitt 6.4 die Ergebnisse zusammengefasst und Schlussfolgerungen daraus gezogen.

Verglichen werden der alte OTF-Algorithmus für einfache Büchi-Spiele (in den folgenden Abschnitten als *OTF* abgekürzt), der DFSBased-Algorithmus mit und ohne Partial-Order-Reduction (*POR*) und Fairness (*F*), der neu entwickelte OTF-Algorithmus für GBGs aus Kapitel 4 (*GOTF*) sowie der Attractor-Algorithmus für GR(1)-Spiele aus Kapitel 5, mit und ohne Partial-Order-Reduction und Fairness. Alle Algorithmen sind für ScenarioTools implementiert, weshalb sie in einer Eclipse-Umgebung unter Java laufen.

6.1 Test-Suite

Zunächst soll mit mehreren kleinen Problemstellungen geprüft werden, mit welchen Spezialfällen die einzelnen Algorithmen zurechtkommen. Dazu wurde eine Test-Suite geschrieben, die auf der beiliegenden CD enthalten ist.

6.1.1 Aufbau der Test-Suite

Die Testfälle lassen sich in drei Kategorien einteilen: allgemeine und GBG-Tests, Assumption-Guarantee-Tests und Fairness-Tests.

Allgemeine Tests

Mit dieser Kategorie sollen zunächst die grundlegenden Eigenschaften geprüft werden. Folgende Testfälle sind enthalten:

- `requirement` prüft, ob der Algorithmus eine Spezifikation mit Requirement-Szenarien umsetzen kann, deren Zielzustände in den Zielzuständen der Specification-Szenarien enthalten sind.
- `disjoint` prüft, ob der Algorithmus eine Spezifikation umsetzen kann, die Requirement-Szenarien verwendet, deren Zielzustände disjunkt mit denen der Specification-Szenarien sind (vgl. Abb. 3.2).
- `includedPath` prüft, ob der Algorithmus eine Spezifikation umsetzen kann, in dem sich ein gewinnender Pfad sowohl ein Präfix als auch ein Suffix mit einem verlierenden Pfad teilt (vgl. Abb. 3.3).
- `includedPathReq` prüft das gleiche, nur dass zusätzlich noch ein Requirement-Szenario genutzt wird, dessen Zielzustände disjunkt mit den Specification-Szenarien sind.
- `memory` prüft, ob der Algorithmus eine Spezifikation umsetzen kann, die ein simples Gedächtnis erfordert (vgl. Abb. 3.5).

Assumption-Guarantee-Tests

Diese Kategorie prüft Eigenschaften, die zur korrekten Erfüllung von GR(1)-Spielen nötig sind. Folgende Testfälle sind enthalten:

- `reqAssumption` prüft, ob der Algorithmus eine Spezifikation umsetzen kann, in der ein Requirement-Szenario aufgrund eines nicht-akzeptierenden Assumption-Zyklus erfüllt wird.
- `alwaysAssumption` prüft, ob der Algorithmus eine Spezifikation umsetzen kann, in der es keine Zielzustände gibt, jedoch in jedem Zustand Assumption-Szenarien verletzt sind.

Fairness-Tests

In dieser Kategorie sind Testfälle enthalten, mit denen die Implementierung von Fairness evaluiert wird. Folgende Testfälle sind enthalten:

- `fairnessRequirement` enthält eine Spezifikation, bei der nur bei angenommener Fairness das Erfüllen eines Requirement-Szenarios garantiert werden kann.
- `fairnessAssumption` enthält eine Spezifikation, bei der nur bei angenommener Fairness eine globale Assumption-Violation vorliegt.

Testfall	OTF	DFSBased	DFSBased (POR)	DFSBased (F)	DFSBased (F+POR)	GOTF	Attractor	Attractor (POR)	Attractor (F)	Attractor (F+POR)
`requirement`	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
`disjoint`		✓	✓	✓	✓	✓	✓	✓	✓	✓
`includedPath`	✓					✓	✓	✓	✓	✓
`includedPathReq`						✓	✓	✓	✓	✓
`memory`							✓	✓	✓	✓
`reqAssumption`		✓	✓	✓	✓	✓	✓	✓	✓	✓
`alwaysAssumption`		✓	✓	✓	✓		✓	✓	✓	✓
`fairnessRequirement`				✓	✓				✓	✓
`fairnessAssumption`				✓	✓				✓	✓
`fairnessEverywhere`				✓	✓				✓	✓

Tabelle 6.1: Ergebnisse der Test-Suite

- `fairnessEverywhere` enthält eine Spezifikation, bei der die Umgebung von jedem von ihr kontrollierten Zustand aus das System in einen verlierenden Zyklus zwingen kann, jedoch ebenfalls ein Zyklus existiert, der all diese Zustände zu einem verlierenden Assumption-Zyklus verbindet.

6.1.2 Ergebnisse

Alle beschriebenen Algorithmen wurden mit der erläuterten Test-Suite evaluiert. Die Ergebnisse sind in Tabelle 6.1 dargestellt.

Es ist leicht zu erkennen, dass der OTF-Algorithmus sehr stark eingeschränkt ist. Er erfüllt zwar den `requirement`-Testfall, dieser ist jedoch bei weitem nicht repräsentativ für eine typische Nutzung von Requirement-Szenarien. Darüber hinaus erfüllt er den `includedPath`-Test, dieser ist allerdings eher eine Schwäche des DFSBased-Algorithmus als eine Stärke des OTF-Algorithmus. Bei allen weiteren Testfällen scheitert der OTF-Algorithmus.

Der GOTF-Algorithmus kann insbesondere den wichtigen `disjoint`-Testfall lösen. Dadurch stellt er eine große Verbesserung gegenüber dem OTF-Algorithmus dar, da dieser Testfall repräsentativ für eine große Gruppe von GBGs ist. Die `includedPath`-Tests konnten ebenfalls beide gelöst werden, was zeigt, dass er die Schwächen des DFSBased-Algorithmus nicht

teilt. Die Unterstützung für GR(1)-Spiele ist jedoch stark beschränkt. Der grundlegende Testfall `reqAssumption` wird zwar korrekt gelöst, jedoch keiner der verbleibenden Testfälle. Auf diese Art kann er zwar schon eine Vielzahl von Spezifikationen umsetzen, die mit einem reinen GBG nicht kompatibel wären, jedoch muss trotzdem die primäre Büchi-Bedingung immer erfüllt sein. Im Kontext von ScenarioTools bedeutet dies, dass die Specification-Szenarien immer erfüllt sein müssen, die Requirement-Szenarien können jedoch auch durch nicht-akzeptierende Assumption-Zyklen erfüllt werden. Insgesamt stellen die Fähigkeiten des GOTF-Algorithmus jedoch eine nennenswerte Verbesserung gegenüber denen des OTF-Algorithmus dar.

Der DFSBased-Algorithmus unterstützt die meisten Testfälle. Ausnahmen sind die bereits beschriebene Schwäche und Spezifikationen, die ein Gedächtnis benötigen. Damit ist er für sehr viele Spezifikationen einsetzbar, jedoch lässt sich nicht ohne weiteres im Voraus abschätzen, ob eine Spezifikation kompatibel ist. Die `includedPath`-Testfälle, in denen er versagt, sind eine Eigenschaften der Suche, die sich aus der Struktur des Zustandsraums ergibt. Da der Zustandsraum aus einer textuellen Spezifikation erzeugt wird, indem die aktiven Szenarien in einen Zustand enkodiert werden, ist diese Beschaffenheit beim Verfassen der Spezifikation nicht direkt nachvollziehbar. Für den Benutzer ist es dadurch unter Umständen schwer nachzuvollziehen, ob nun seine Spezifikation Widersprüche enthält oder ob sie diese Schwäche des Algorithmus auslöst.

Der Attractor-Algorithmus ist der einzige Algorithmus, der für jeden Testfall korrekt arbeitet. Selbst wenn man den `memory`-Testfall weglässt, der eine Eigenschaft prüft, die bei den anderen Algorithmen nicht vorliegt, erfüllt er trotzdem mehr Testfälle als jeder andere Algorithmus. Er ist damit gut als Referenz für andere Algorithmen geeignet. Jedoch ist nicht direkt ersichtlich, ob eine Spezifikation nur aufgrund der Unterstützung für gedächtnisbehaftete Spiele unterstützt wird, da der Algorithmus inhärent auf die Nutzung eines Gedächtnis ausgelegt ist. Dies wird auch dann genutzt, wenn andere Algorithmen ohne auskommen, wodurch der entstehende Controller unter Umständen suboptimal ist. Dennoch ist er der einzige bisher implementierte Algorithmus in ScenarioTools, der alle Spezifikationen korrekt beantworten kann.

Die optionale Erweiterung des Attractor-Algorithmus um Fairness ist konsistent mit der Implementierung im DFSBased-Algorithmus. Beide Algorithmen können auf diese Art weitere Spezifikationen umsetzen, bei denen eine strenge Auslegung der Büchi-Bedingung logisch sinnwidrig wäre. Partial-Order-Reduction hatte keinen Einfluss auf die Ergebnisse, dies ist jedoch bei so kleinen Zustandsräumen nicht überraschend.

6.2 Performance und Speicherverbrauch

Nun soll die Performance der Algorithmen verglichen werden. Dazu wird das *car-to-x*-Beispiel genutzt, welches von Greenyer et al. [9] entwickelt wurde. Dies stellt ein System dar, welches das Passieren von Hindernissen auf einer Straße reglementiert. Die Anzahl an Autos, die aus jeder Richtung kommen, ist dabei variabel definierbar, woraus sich mehrere Testfälle ergeben. Das System muss sicherstellen, dass jedes Auto irgendwann das Hindernis passieren kann, und dass aus verschiedenen Richtungen kommende Autos nicht kollidieren. Zur Evaluation werden die Fälle 1-0, 1-1, 2-0 und 3-0 betrachtet. Für einen Testfall x-y geben x und y dabei jeweils an, wie viele Autos aus einer bzw. der anderen Richtung auf das Hindernis zufahren. Die verbleibenden Testfälle waren aufgrund ihres Zeit- bzw. Speicherverbrauchs auf dem Testsystem nicht sinnvoll durchführbar.

6.2.1 Erfasste Werte

Das wichtigste Maß ist dabei zunächst die Laufzeit, die in Sekunden zwischen Beginn und Ende der Ausführung gemessen wird. Insbesondere bei den On-the-Fly-Algorithmen könnte es hier zu sehr unterschiedlichen Ergebnissen kommen, da potentiell in verschiedenen Ausführungen jeweils eine sehr kurze und eine sehr umfangreiche Strategie gefunden werden könnte. Aus diesem Grund wurde für diese Verfahren zusätzlich zu den erfassten Werten jeder Testfall fünfmal wiederholt, wobei allerdings keine nennenswerten Ausreißer auftraten. Weiterhin wird betrachtet, wie viele Zustände während der Ausführung erstellt wurden. On-the-Fly-Algorithmen erstellen bestenfalls deutlich weniger Zustände. Da insbesondere der GOTF-Algorithmus sehr viele Daten verwaltet (die erstellten Pfade), wird zusätzlich der Speicherverbrauch der Algorithmen evaluiert.

Der von einem Programmabschnitt genutzte Speicher lässt sich unter Java jedoch nicht exakt bestimmen, da der Garbage-Collector nicht direkt kontrolliert werden kann. Der während der Ausführung genutzte Speicher könnte deshalb schon freigegeben sein, wenn die Werte ausgegeben werden. Weiterhin könnte zu Beginn der Ausführung noch allokierter Speicher von vorherigen Programmteilen während der Ausführung freigegeben werden. Die ausgelesenen Werte waren deshalb sehr unzuverlässig, und unterschieden sich bei wiederholter Ausführung eines Testfalls teils stark. Als Kompromiss wird deshalb der von dem javaw-Prozess unter Windows allokierte Speicher angegeben. Alle Testfalle werden in einer frisch gestarteten Eclipse-Instanz mit ScenarioTools-Support ausgeführt, in einer 32-bit JVM. Es ist wichtig anzumerken, dass der so ausgelesene Speicher nur ein grober Richtwert sein kann, da Java Speicher oft in größeren Blöcken allokiert. Vergleichen lassen sich also allein die Größenordnungen der Werte, nicht der exakte Verbrauch. Eine neu gestartete Eclipse-Instanz, bei der alle

Testfall	OTF	DFSBased	DFSBased (POR)	DFSBased (F)	DFSBased (F+POR)	GOTF	Attractor	Attractor (POR)	Attractor (F)	Attractor (F+POR)
car-to-x 1-0	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
car-to-x 1-1	✗	✗	✓	✓	✓	✓	✗	✗	✓	✓
car-to-x 2-0	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
car-to-x 3-0	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Tabelle 6.2: Ergebnisse der Algorithmen

Initialisierungen abgeschlossen wurden, hatte als Vergleichswert zwischen 690 MB und 720 MB Speicher allokiert.

Für den Attractor-Algorithmus werden darüber hinaus noch weitere Werte erhoben und diskutiert. Da hier der komplette Zustandsraum vor dem eigentlichen Algorithmus erstellt wird, wird die Laufzeit in beide Aspekte aufgebrochen. Die Größe des entstehenden Controllers unter Nutzung des Verfahrens aus Abschnitt 5.2 wird ebenfalls betrachtet.

6.2.2 Ergebnisse

Zur besseren Visualisierung der Ergebnisse werden soweit anwendbar Diagramme verwendet. Die exakten Ergebnisse der einzelnen Testdurchläufe befinden sich nach Algorithmen sortiert in Anhang A.

Korrektheit

Zunächst ist in Tabelle 6.2 gelistet, für welche Testfälle die Algorithmen eine Strategie für das System finden konnten. ✓ steht dabei für eine gefundene Systemstrategie, ✗ für eine Negativantwort. Für die Testfälle 1-0, 2-0 und 3-0 haben alle Algorithmen eine Strategie finden können. Bei Testfall 1-1 gibt es jedoch verschiedene Ergebnisse. OTF, DFSBased, sowie Attractor und Attractor mit Partial-Order-Reduction konnten hier keine Strategie finden, alle anderen Algorithmen hingegen schon. Aktivierte Fairness führt dazu, dass sowohl der DFSBased- als auch der Attractor-Algorithmus eine Strategie finden können. Ist jedoch nur Partial-Order-Reduction aktiv, so unterscheiden sich die Ergebnisse. Während der Attractor-Algorithmus weiterhin keine Strategie findet, gibt der DFSBased-Algorithmus nun eine Strategie für das System zurück. Dies deutet auf ein False-Positive des

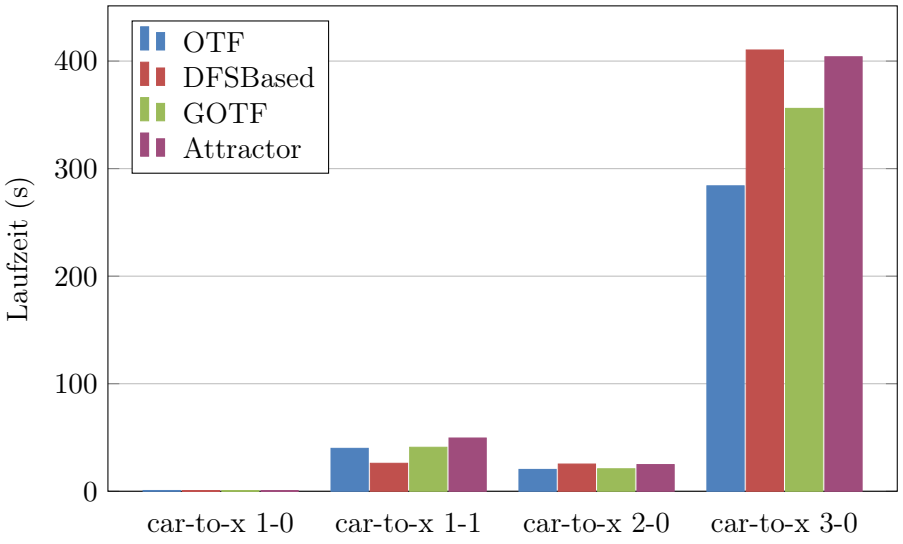

Abbildung 6.1: Laufzeit der Algorithmen

DFSBased-Algorithmus hin. Der GOTF-Algorithmus gibt ebenfalls eine Systemstrategie zurück, weshalb hier vermutlich ebenfalls ein False-Positive vorliegt. Hierauf wird in Abschnitt 6.3 weiter eingegangen.

Laufzeit

Die Laufzeit der Algorithmen ist in Abbildung 6.1 dargestellt. Bei wiederholter Ausführung der Testfälle ergaben sich Laufzeitschwankungen im niedrigen Sekundenbereich. Diese dürften größtenteils dem Scheduling des Betriebssystems zuzuschreiben sein. Für den Testfall 1-0 ist die Laufzeit bei allen Algorithmen vernachlässigbar gering. Bei dem Testfall 1-1 hat der DFSBased-Algorithmus mit 26s die geringste Laufzeit, da er frühzeitig erkennen kann, dass keine Strategie existiert und somit nur einen kleinen Teil des Zustandsraums konstruiert. Dies ist den anderen Algorithmen nicht möglich. Der Unterschied zwischen dem OTF-Algorithmus mit 39.9s und dem GOTF-Algorithmus mit 40.9s liegt hier bei nur einer Sekunde. Dies deutet darauf hin, dass der Overhead des GOTF-Algorithmus bei dieser Größe noch vernachlässigbar ist. Der Attractor-Algorithmus hat mit 49.5s die längste Laufzeit, da er den kompletten Zustandsraum erstellen muss. Er benötigt für diesen Testfall deshalb fast doppelt so lange, wie der DFSBased-Algorithmus.

Die Laufzeiten für den Testfall 2-0 liegen alle innerhalb von 5s. Die On-the-Fly-Algorithmen haben hier einen leichten Vorteil, da sie einige Zustände weniger erstellen müssen als der DFSBased- und Attractor-Algorithmus. Die größten Unterschiede gibt es beim Testfall 3-0. Der OTF-Algorithmus hat hier die mit Abstand geringste Laufzeit, 284s. Der Testfall nutzt

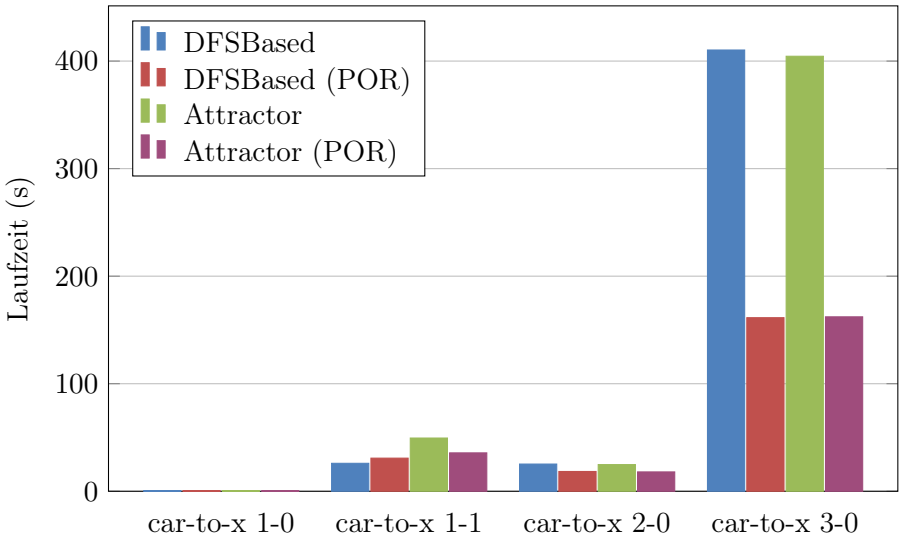

Abbildung 6.2: Laufzeit der Algorithmen mit Partial-Order-Reduction

jedoch Requirement-Szenarien, sodass nicht garantiert war, dass er zu dem korrekten Ergebnis kommt. Der Laufzeitvorteil ergibt sich größtenteils aus der Einfachheit des Algorithmus. Der GOTF-Algorithmus hat mit 356s eine deutlich höhere Laufzeit. Hier zeigen sich die Kosten der zusätzlichen Operationen im Vergleich zum einfachen OTF-Algorithmus. Dennoch ist er um einiges schneller, als die restlichen Algorithmen. Der DFSBased-Algorithmus erzielt eine Laufzeit von 410s, während der Attractor-Algorithmus 404s benötigt. Der Attractor-Algorithmus ist somit marginal schneller als der DFSBased-Algorithmus, jedoch ist der Unterschied in Relation zur Laufzeit sehr gering, sodass er auch auf Schwankungen bei der CPU-Nutzung zurückgeführt werden könnte. Bei diesem Testfall zeigt sich besonders gut, welche Laufzeitvorteile On-the-Fly-Algorithmen bei großen Zustandsräumen haben können.

Nun soll die Auswirkung der Erweiterungen auf die Laufzeit betrachtet werden. Die Laufzeitunterscheide bei aktivierter Fairness sind relativ gering, sodass diese Ausprägungen der Algorithmen hier nicht weiter berücksichtigt werden. Die Werte können ebenfalls in Anhang A nachgeschlagen werden. Abbildung 6.2 stellt die Laufzeit des DFSBased- und Attractor-Algorithmus jeweils mit und ohne aktivierter Partial-Order-Reduction dar.

In Testfall 1-1 benötigt der DFSBased-Algorithmus mit Partial-Order-Reduction nun mehr Zeit, 30.8s statt 25s. Dies kommt daher, dass er nun eine Siegstrategie im Graphen findet und daher nicht mehr frühzeitig terminieren kann. Der Attractor-Algorithmus benötigt mit 35.7s nun deutlich weniger Zeit als vorher mit 49.5s. Die Anzahl erstellter Zustände konnte durch POR von 2264 auf 1646 reduziert werden, die Laufzeitersparnis ist also proportional zur Reduzierung des Zustandsraums. Auffällig war, dass der

Testfall	Gesamt	Zustandsraum	Algorithmus
car-to-x 1-0	0.5s	0.5s	0.002s
car-to-x 1-1	49.5s	49.48s	0.08s
car-to-x 2-0	24.8s	24.78s	0.09s
car-to-x 3-0	404.5s	403.4s	1.05s

Tabelle 6.3: Zeitaufteilung des Attractor-Algorithmus

DFSBased-Algorithmus nur 1389 Zustände erstellte. Da beide Algorithmen die selbe Partial-Order-Reduction-Implementierung nutzen, und sie bei erkannter Systemstrategie beide den kompletten Zustandsraum erzeugen müssen, ist dies ein weiteres Indiz für ein Fehlverhalten. In Testfall 2-0 haben beide Algorithmen eine vergleichbare Ersparnis und benötigen nun etwa 18s statt 25s.

Die größte Laufzeitersparnis gibt es im Testfall 3-0. Beide Algorithmen benötigen mit Partial-Order-Reduction nur noch 162s, eine deutliche Reduzierung gegenüber den 410s bzw. 404s ohne Partial-Order-Reduction. Die erstellten Zustände wurden in beiden Fällen von 11964 auf 4595 reduziert. Die Laufzeit ist also bei beiden Algorithmen weiterhin proportional zur Größe des Zustandsraums. Mit aktivierter Partial-Order-Reduction haben beide Algorithmen eine deutlich geringere Laufzeit als die On-the-Fly-Algorithmen. Jedoch wurde für diese die Erweiterung nicht implementiert. Es ist anzunehmen, dass sie mit implementierter Partial-Order-Reduction weiterhin schneller arbeiten würden, als die anderen beiden Algorithmen.

Zuletzt wird betrachtet, wie sich die Laufzeit des Attractor-Algorithmus in die Erstellung des Zustandsraum und die eigentliche Durchführung des Algorithmus aufteilt. Hier werden nur die Werte ohne Erweiterungen betrachtet, da die restlichen Werte dem gleichem Muster folgen. Dies ist in Tabelle 6.3 aufgelistet. Es ist leicht zu erkennen, dass die überwiegende Mehrheit der Laufzeit mit der Erstellung des Zustandsraum verbracht wird. Selbst beim größten betrachteten Testfall 3-0 benötigt der Algorithmus selbst nur 1.05s, was 0.26% der Laufzeit entspricht. Bei den verbleibenden Algorithmen lässt sich die Erstellung des Zustandsraums nicht so einfach von den Berechnungen der Algorithmen selbst trennen, jedoch wird die Erstellung dort vergleichbar viel Aufwand erfordern. Optimierungen der Synthese-Algorithmen dürften deshalb auf die Gesamtlaufzeit nur eine untergeordnete Auswirkung haben, solange sie den Zustandsraum nicht verringern.

Speicherverbrauch

In Abbildung 6.3 ist der Speicherverbrauch der Algorithmen dargestellt. In den Testfällen 1-0, 1-1 und 2-0 sind die Unterschiede sehr gering, was darauf hindeutet, dass für diese der initial von Java allokierte Speicher ausreichend

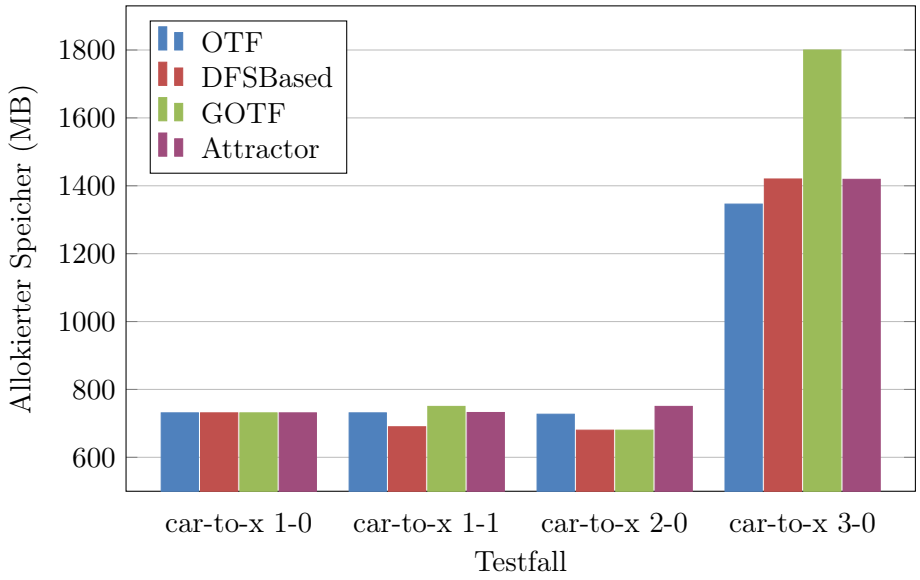

Abbildung 6.3: Allokierter Speicher der Algorithmen

ist. Für den Testfall 3-0 gibt es jedoch interessante Unterschiede. Am wenigsten wurde hier mit 1346 MB durch den OTF-Algorithmus allokiert. Das kann dadurch begründet werden, dass er keine weiteren Daten verwalten muss und nicht den kompletten Zustandsgraphen erstellt hat. Die DFSBased- und Attractor-Algorithmen allokierten jeweils 1420 MB und 1419 MB, der Speicherverbrauch dieser Algorithmen ist also gleichwertig. Mit 1800 MB allokiert der GOTF-Algorithmus mit Abstand am meisten Speicher. Dies ist den zusätzlichen Daten geschuldet, die dieser Algorithmus verwaltet. Setzt man den zusätzlich allokierten Speicher ins Verhältnis zu der Anzahl Zustände im Graphen, so ergibt dies für diesen konkreten Testfall einen zusätzlichen Speicherverbrauch von ungefähr 33 KB pro Zustand. Hierbei sind allerdings noch die beobachten Schwankungen zu berücksichtigen. Bei größeren Zustandsräumen könnte dies ein Problem darstellen, da der verfügbare Speicher neben der Laufzeit dort zur begrenzenden Ressource wird.

Mit aktivierter Partial-Order-Reduction verringert sich der Speicherverbrauch des DFSBased- und Attractor-Algorithmus proportional zu der Anzahl eingesparter Zustände. Der DFSBased-Algorithmus benötigte für den Testfall 3-0 noch 1116 MB, während für den Attractor-Algorithmus 1030 MB reichten. Fairness hatte keinen Einfluss auf den Speicherverbrauch.

Sonstiges

In Tabelle 6.4 ist die Größe der vom Attractor-Algorithmus erzeugten Controller aufgelistet, zusammen mit der Anzahl Zustände des Zustands-

Testfall	Zustände	Gewinnend	Controller
car-to-x 1-0	44	38	38
car-to-x 1-1	2264	1216	44
car-to-x 2-0	1224	972	928
car-to-x 3-0	11964	9612	9212

Tabelle 6.4: Größe der erzeugten Controller des Attractor-Algorithmus

raum und der Anzahl Zustände, von denen aus das System gewinnt. Die Größe des Controllers ist in keinem der Testfälle größer als die Anzahl gewinnender Zustände. Durch die Überführung in einen Graphen mit einer gedächtnislosen Strategie wird also, anders als es der Worst-Case vermuten lässt, bei diesem Beispiel sogar eine Verbesserung erreicht. Da für den Testfall 1-1 keine Siegstrategie gefunden werden konnte, ist der Controller ein Gegenbeispiel. Dies fällt sehr kompakt aus, mit 44 Zuständen ist es mit Abstand das kleinste erzeugte Gegenbeispiel der betrachteten Algorithmen.

Durch Partial-Order-Reduction verringert sich die Größe des Controllers erwartungsgemäß. Er ist jedoch auch hier in jedem Fall kleiner als der gewinnende Subgraph. Durch Fairness erhöht sich die Anzahl gewinnender Zustände, wodurch auch der Controller an Größe zunimmt. Auch hier ist er jedoch immer ein kleinerer Graph als vor der Umwandlung.

6.3 Auffälligkeiten

Die Implementierung des GOTF-Algorithmus hat sich aufgrund des False-Positive beim car-to-x 1-1 Testfall als fehlerhaft erwiesen. Es wird vermutet, dass der Fehler in der Zyklenerkennung liegt, die sich wie in Abschnitt 4.7 erklärt als komplex und fehleranfällig herausgestellt hat. Da sich die zyklenschließenden Transitionen und die Vorgänger bei Übergängen zwischen Konfigurationen ändern können, müssen dort Berechnungen durchgeführt werden, bei denen vermutlich ein Sonderfall nicht korrekt behandelt wird. Als Folge werden möglicherweise einige Zyklen nicht mehr korrekt berücksichtigt. Der Fehler konnte leider nicht mehr rechtzeitig lokalisiert werden, da er nur bei sehr großen Zustandsräumen beobachtet wurde. Es könnte zugunsten einer simpleren Implementierung auf eine integrierte Zyklenerkennung verzichtet werden, und die Zyklen stattdessen als zusätzlicher Schritt während der Verifikation gesucht werden. Wie in Abschnitt 4.3.2 beschrieben sollte dies für eine durchschnittliche Spezifikation in relativ effizienter Zeit möglich sein.

Mit aktivierter Partial-Order-Reduction haben der DFSBased- und der Attractor-Algorithmus unterschiedliche Ergebnisse beim car-to-x 1-1 Testfall. Der DFSBased-Algorithmus gibt an, dass eine Siegstrategie für das System existiert, während der Attractor-Algorithmus dies verneint. Das legt na-

he, dass einer der beiden Algorithmen unter diesen Umständen inkorrekt arbeitet. Da *ohne* Partial-Order-Reduction das System mit beiden Algorithmen verliert, könnte dies ein Fehler im DFSBased-Algorithmus sein. Dies wird weiterhin dadurch bekräftigt, dass der DFSBased-Algorithmus in dieser Konfiguration deutlich weniger Zustände erstellt als der Attractor-Algorithmus, obwohl für beide die selbe Partial-Order-Reduction-Implementierung verwendet wird. Die Ursache konnte jedoch leider nicht identifiziert werden.

Beide Algorithmen müssen den kompletten Zustandsraum erstellen, um eine Siegstrategie für das System zu berechnen. Jedoch wurden in einigen Testläufen eine geringe Anzahl weniger Zustände erstellt als in den übrigen. Dies geschah sowohl beim DFSBased- als auch beim Attractor-Algorithmus zufällig und ohne nachvollziehbaren Grund. Da es bei beiden Algorithmen beobachtet wurde, liegt die Ursache vermutlich bei ScenarioTools.

6.4 Auswertung

Die neu entwickelten bzw. implementierten Algorithmen können insgesamt als wirksam angesehen werden. Beide Algorithmen haben Vorzüge gegenüber anderen Algorithmen ihrer Art.

Für den GOTF-Algorithmus hat sich gezeigt, dass seine Laufzeit trotz der zusätzlichen Operationen deutlich vor den Verfahren liegt, die nicht on-the-fly arbeiten. Er konnte wichtige Testfälle für GBGs korrekt umsetzen, jedoch werden insbesondere GR(1)-Spiele nur teilweise unterstützt. Zwar werden nicht-akzeptierende Assumption-Zyklen für alle sekundären Siegbedingungen erkannt, jedoch ist dies nicht der Fall für die primäre Siegbedingung. Hier wäre es denkbar, dass dieser Fall während der initialen Suche behandelt wird, indem dort wie später Zyklen analysiert werden. Dieser Aspekt konnte leider aus Zeitgründen nicht umgesetzt werden. Der Speicherbrauch beim car-to-x 3-0 Testfall ist jedoch im Vergleich zu den anderen Verfahren sehr hoch, was die Skalierbarkeit des Verfahrens einschränken könnte. Die bestehende Implementierung ist allerdings noch fehlerhaft.

Die Laufzeit des Attractor-Algorithmus ist der des DFSBased-Algorithmus ebenbürtig, ebenso der Speicherverbrauch. Die einzigen Fälle, in denen der DFSBased-Algorithmus klar schneller ist, sind Spezifikationen ohne Siegstrategie, bei denen er frühzeitig terminieren kann. Dass er in jedem Fall den kompletten Zustandsraum aufbauen muss ist die größte Schwäche des Attractor-Algorithmus. Selbst wenn andere Algorithmen nach wenigen Zuständen schon entscheiden könnten, dass es in diesem Graphen keine Strategie für das System geben kann. Es ist jedoch denkbar, dass man zumindest einige dieser Fälle schon während des Aufbaus des Zustandsraum erkennen kann. Dies würde jedoch die Komplexität erhöhen und eventuell weitere Rechenzeit benötigen, die sich negativ auf die Performance von Graphen auswirkt, in denen es eine Strategie gibt.

Die Partial-Order-Reduction hat gezeigt, wie viel Zeit eingespart werden kann, wenn der Zustandsraum verringert wird. Die Aufteilung der Laufzeit auf die Erstellung des Zustandsraums und der eigentlichen Durchführung des Algorithmus in Tabelle 6.3 zeigt jedoch, dass neben der Reduzierung der Anzahl erstellter Zustände auch die Geschwindigkeit der Erstellung selbst zu vielversprechenden Laufzeitoptimierungen führen kann. Über 99.74% der Laufzeit werden mit der Erstellung des Zustandsraums verbracht. Neben allgemeiner Optimierung wäre eine mögliche Forschungsrichtung die Parallelisierung des Aufbaus des Zustandsraums. Dies ist für die anderen Algorithmen unpraktisch, da sie alle mehr oder weniger on-the-fly arbeiten, könnte sich hier jedoch schon bei moderatem Speed-Up auszahlen. Selbst wenn auf diese Art nur wenige Prozent der Laufzeit eingespart werden, könnte es eine spürbare Laufzeitverbesserung gegenüber den Worst-Case-Fällen der anderen Algorithmen sein (Positive Antwort beim DFSBased-Algorithmus, negative Antwort bei den OTF-Algorithmen).

Da der Attractor-Algorithmus Strategien mit Gedächtnis erzeugen kann, ist er der erste vollständige Algorithmus für ScenarioTools. Somit können mit ihm viele neue Spezifikationen umgesetzt werden, bei denen die anderen Algorithmen scheitern, bei vergleichbarer Worst-Case-Laufzeit.

Kapitel 7

Fazit

ScenarioTools ermöglicht den Entwurf und die Analyse szenariobasierter Spezifikationen. Dazu wird eine formale aber vergleichsweise intuitive Notation genutzt, sodass es auch ohne umfangreiches Fachwissen für den formalen Entwurf nutzbar ist. Jedoch schränkt die Größe der erzeugten Zustandsräume die Anwendbarkeit ein. Dem können auch bestehende On-the-Fly-Algorithmen aufgrund fehlender Unterstützung für komplexere Büchi-Spiele nicht entgegenwirken. Darüber hinaus konnte bisher noch nicht jede Spezifikation auch korrekt verifiziert werden. In dieser Arbeit wurden deshalb Verfahren entwickelt und betrachtet, mit denen die Synthese verbessert werden kann.

Zunächst wurde ein On-the-Fly-Algorithmus für gedächtnislose Generalized Büchi-Spiele entwickelt, der darüber hinaus für jede außer der primären Siegbedingung Anforderungen in der Assumption-Guarantee-Form unterstützt. Dieser neue On-the-Fly-Algorithmus kann weitaus mehr Spezifikationen korrekt verifizieren, als bestehende On-the-Fly-Algorithmen. Weiterhin wurde in ScenarioTools ein vollständiger Attractor-Algorithmus für GR(1)-Spiele von Chatterjee et al. implementiert. Für diesen Algorithmus wurde eine Erweiterung entwickelt, mit der das Prinzip der Fairness umgesetzt werden kann. Dies entspricht einem abgeschwächtem Umgebungsverhalten, wodurch Spezifikationen korrekt geprüft werden können, für welche die strikte Semantik eines GR(1)-Spiels keinen Sinn ergibt.

Die Evaluation hat gezeigt, dass diese Algorithmen sinnvoll für szenariobasierte Spezifikationen anwendbar sind. Der Attractor-Algorithmus ist hinsichtlich der Laufzeit und des Speicherverbrauchs dem DFSBased-Algorithmus in vielen Fällen ebenbürtig. Weiterhin hat sich die Fairness-Erweiterung als konsistent mit der bestehenden Umsetzung für den DFSBased-Algorithmus erwiesen. Die Referenzimplementierung des GOTF-Algorithmus hat sich als fehlerhaft erwiesen, dennoch konnte gezeigt werden, dass die On-the-Fly-Suche für GBGs in akzeptabler Zeit umsetzbar ist. Er konnte für GBGs wichtige Testfälle korrekt lösen. Die Laufzeit

ist den Verfahren, die nicht on-the-fly arbeiten, teils deutlich überlegen und unterliegt selbst nur dem einfachen OTF-Algorithmus. Einzig die Speichernutzung dieses Verfahrens könnte die Skalierbarkeit beeinflussen.

Insgesamt stellen die konzipierten bzw. implementierten Algorithmen eine sinnvolle Erweiterung für ScenarioTools dar. Mit dem Attractor-Algorithmus können erstmals Spezifikationen umgesetzt werden, die Strategien mit Gedächtnis benötigen. Der neue On-the-Fly-Algorithmus ist trotz größerer Ausdrucksstärke weiterhin effizienter als andersartige Verfahren.

7.1 Ausblick

Neben der offensichtlichen Korrektur der Implementierung bietet der GOTF-Algorithmus noch Raum für Erweiterungen. Momentan unterstützt er für die erste Siegbedingung keine Akzeptanz im Sinne einer GR(1)-Bedingung. Diese könnte wie erwähnt schon während der initialen Suche geprüft werden. Weiterhin sind Verbesserungen bzw. alternative Ansätze bei der Erzeugung von Konfigurationen denkbar, indem etwa nur ein verlierender Zyklus auf einmal betrachtet wird. Auf diese Art könnten vorzeitig verlierende Pfade identifiziert werden, die im bestehenden Ansatz schlimmstenfalls in jeder erzeugten Konfiguration gesondert gefunden werden müssen. Darüber hinaus könnte die blinde Suche durch Heuristiken erweitert werden, um so vielversprechendere Konfiguration zuerst zu analysieren. Im Beispiel in Abbildung 4.2 wurden beispielsweise bewusst ungünstige Entscheidungen getroffen, die durch eine gute Heuristik vermieden werden könnten. Eine mögliche Reduktion des Speicherverbrauchs würde ebenfalls eine lohnenswerte Verbesserung darstellen. Dies könnte entweder durch effizientere Speicherung der Konfigurationen erreicht werden, oder durch eine Methode, die nicht explizit jeden gewinnenden Pfad an jedem Zustand hinterlegen muss.

Für den Attractor-Algorithmus könnte die Parallelisierung der Zustandsraumerstellung eine lohnenswerte Forschungsrichtung sein. Die Evaluation hat gezeigt, dass eine überwiegende Mehrheit der Laufzeit für die Erstellung des Zustandsraums aufgebracht wird. Selbst ein kleiner Speedup dieser Erstellung würde eine weitaus größere Verbesserung der Laufzeit bewirken, als jegliche Verbesserung des eigentlichen Algorithmus. Die Implementierung der optimierten Algorithmen mit niedrigerer Laufzeitkomplexität von Chatterjee et al. wäre zwar möglich, würde deshalb jedoch keine spürbare Verbesserung darstellen.

Anhang A

Evaluationsergebnisse

In den folgenden Abschnitten sind die Ergebnisse der Evaluation anhand der car-to-x Testfälle gelistet. In Abschnitt A.1 sind die Ergebnisse aller Algorithmen ohne jegliche Erweiterungen gelistet, in jeweils einer Tabelle pro Algorithmus. Abschnitt A.2 listet die Ergebnisse der DFSBased- und Attractor-Algorithmen mit Partial-Order-Reduction. In Abschnitt A.3 sind die Ergebnisse mit aktivierter Fairness und in Abschnitt A.4 zuletzt die Ergebnisse mit aktivierter Partial-Order-Reduction und Fairness.

Für jeden Algorithmus wird neben dem Testfall das Ergebnis (Erg.) gelistet. Das erwartete Ergebnis ist zusätzlich in der selben Spalte in Klammern angegeben. Dies ist *true* für alle Testfälle mit Ausnahme von car-to-x 1-1 ohne Fairness. Weiterhin werden die erstellten Zustände (Zust.), die für das System gewinnenden Zustände (Gew. Zust. bzw. Gew.), die benötigte Zeit und der allokierte Speicher gelistet. Für den Attractor-Algorithmus wird darüber hinaus die Anzahl Zustände im Controller (Contr.) gelistet.

Die Tests wurden unter Windows 7 in einer 32-Bit Eclipse-Mars-Umgebung durchgeführt. Die verwendete Java-Virtual-Machine war ebenfalls 32-Bit. Die Voreinstellungen von ScenarioTools wurden ohne Änderungen übernommen, mit Ausnahme von Fairness bzw. Partial-Order-Reduction in den betreffenden Testläufen. Der verwendete Prozessor ist ein Intel Core i5-2500K ohne Übertaktungen. Die Testfälle wurden über das „ScenarioTools Synthesis“-Menü gestartet. Für jeden Testfall wird der erste Durchlauf aufgelistet, jedoch wurden sie jeweils fünfmal wiederholt, um Ausreißer auszuschließen. Die Ergebnisse unterschieden sich allerdings nur geringfügig. Die Laufzeitunterschiede waren im einstelligen Sekundenbereich, wodurch die Ursache vermutlich größtenteils beim Scheduling des Betriebssystems liegt. Bei den kleineren Testfällen waren sie noch weitaus geringfügiger. Die Unterschiede beim allokierten Speicher bewegten sich innerhalb der beobachteten 40 MB.

A.1 Ohne Erweiterungen

Testfall	Erg.	Zust.	Gew. Zust.	Zeit	Speicher
car-to-x 1-0	true (true)	38	38	0.7s	731 MB
car-to-x 1-1	false (false)	2211	1222	39.9s	731 MB
car-to-x 2-0	true (true)	1177	876	20.3s	727 MB
car-to-x 3-0	true (true)	11623	8652	284s	1346 MB

Tabelle A.1: Ergebnisse des OTF-Algorithmus

Testfall	Erg.	Zust.	Gew. Zust.	Zeit	Speicher
car-to-x 1-0	true (true)	44	34	0.55s	731
car-to-x 1-1	false (false)	1094	784	26s	690
car-to-x 2-0	true (true)	1224	972	25.3s	680
car-to-x 3-0	true (true)	11964	9612	410.3s	1420

Tabelle A.2: Ergebnisse des DFSBased-Algorithmus

Testfall	Erg.	Zust.	Gew. Zust.	Zeit	Speicher
car-to-x 1-0	true (true)	44	38	0.6s	731
car-to-x 1-1	true (false)	2143	1806	40.9s	750
car-to-x 2-0	true (true)	1125	1024	20.9s	680
car-to-x 3-0	true (true)	11240	10220	356s	1800

Tabelle A.3: Ergebnisse des GOTF-Algorithmus

Testfall	Erg.	Zust.	Gew.	Contr.	Zeit	Speicher
car-to-x 1-0	true (true)	44	38	38	0.5s	731
car-to-x 1-1	false (false)	2264	1216	44	49.5s	732
car-to-x 2-0	true (true)	1224	972	928	24.8s	750
car-to-x 3-0	true (true)	11964	9612	9212	404s	1419

Tabelle A.4: Ergebnisse des Attractor-Algorithmus

A.2 Mit Partial-Order-Reduction

Testfall	Erg.	Zust.	Gew. Zust.	Zeit	Speicher
car-to-x 1-0	true (true)	42	36	0.5s	731
car-to-x 1-1	true (false)	1389	975	30.8s	680
car-to-x 2-0	true (true)	922	722	18.8s	691
car-to-x 3-0	true (true)	4595	3837	161.4s	1116

Tabelle A.5: Ergebnisse des DFSBased-Algorithmus mit Partial-Order-Reduction

Testfall	Erg.	Zust.	Gew.	Contr.	Zeit	Speicher
car-to-x 1-0	true (true)	42	34	34	0.5s	727
car-to-x 1-1	false (false)	1646	896	27	35.7s	737
car-to-x 2-0	true (true)	922	696	600	18s	688
car-to-x 3-0	true (true)	4595	3698	3554	162.2s	1030

Tabelle A.6: Ergebnisse des Attractor-Algorithmus mit Partial-Order-Reduction

A.3 Mit Fairness

Testfall	Erg.	Zust.	Gew. Zust.	Zeit	Speicher
car-to-x 1-0	true (true)	44	34	0.5s	731
car-to-x 1-1	true (true)	2264	1820	49.3s	738
car-to-x 2-0	true (true)	1224	1080	25.3s	732
car-to-x 3-0	true (true)	11964	10692	410s	1467

Tabelle A.7: Ergebnisse des DFSBased-Algorithmus mit Fairness

Testfall	Erg.	Zust.	Gew.	Contr.	Zeit	Speicher
car-to-x 1-0	true (true)	44	38	38	0.5s	730
car-to-x 1-1	true (true)	2264	1880	1802	51.3s	730
car-to-x 2-0	true (true)	1224	1080	1024	24.9s	753
car-to-x 3-0	true (true)	11964	10692	10172	402s	1390

Tabelle A.8: Ergebnisse des Attractor-Algorithmus mit Fairness

A.4 Mit Partial-Order-Reduction und Fairness

Testfall	Erg.	Zust.	Gew. Zust.	Zeit	Speicher
car-to-x 1-0	true (true)	42	36	0.49s	719
car-to-x 1-1	true (true)	1645	1301	36.1s	748
car-to-x 2-0	true (true)	923	815	18.4s	719
car-to-x 3-0	true (true)	4595	4187	165.3s	1082

Tabelle A.9: Ergebnisse des DFSBased-Algorithmus mit Partial-Order-Reduction und Fairness

Testfall	Erg.	Zust.	Gew.	Contr.	Zeit	Speicher
car-to-x 1-0	true (true)	42	34	34	0.5s	727
car-to-x 1-1	true (true)	1646	1194	1098	35.3s	727
car-to-x 2-0	true (true)	923	792	528	18.7s	737
car-to-x 3-0	true (true)	4593	4086	3874	161.7s	1032

Tabelle A.10: Ergebnisse des Attractor-Algorithmus mit Partial-Order-Reduction und Fairness

Anhang B

Inhalt der CD

Die beiliegende CD enthält folgenden Inhalt:

- Diese Ausarbeitung als PDF-Dokument
- Die Ergebnisse der Evaluation
- Den Source-Code der umgesetzten Algorithmen
- Die für die Evaluation erstellte Test-Suite
- Eine Textdatei mit Verwendungshinweisen

Literaturverzeichnis

[1] R. Alur and S. La Torre. Deterministic generators and games for ltl fragments. *ACM Transactions on Computational Logic (TOCL)*, 5(1):1–25, 2004.

[2] J. R. Büchi. Symposium on decision problems: On a decision method in restricted second order arithmetic. *Studies in Logic and the Foundations of Mathematics*, 44:1–11, 1966.

[3] K. Chatterjee, W. Dvorák, M. Henzinger, and V. Loitzenbauer. Conditionally optimal algorithms for generalized büchi games. *CoRR*, abs/1607.05850, 2016.

[4] K. Chatterjee and T. A. Henzinger. Strategy improvement for stochastic rabin and streett games. In *CONCUR 2006–Concurrency Theory*, pages 375–389. Springer, 2006.

[5] Y. Choueka. Theories of automata on ω-tapes: A simplified approach. *J. Comput. Syst. Sci.*, 8(2):117–141, Apr. 1974.

[6] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT press, 1999.

[7] G. De Giacomo, F. Patrizi, and S. Sardiña. Automatic behavior composition synthesis. *Artificial Intelligence*, 196:106–142, 2013.

[8] R. Ehlers. Generalized Rabin (1) Synthesis with Applications to Robust System Synthesis. *NASA Formal*, (1003):1–15, 2011.

[9] J. Greenyer, D. Gritzner, N. Glade, T. Gutjahr, and F. König. Scenario-based specification of car-to-x systems1. *URL: http://ceur-ws. org*, 1559, 2016.

[10] J. Greenyer and E. Kindler. Compositional synthesis of controllers from scenario-based assume-guarantee specifications. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8107 LNCS:774–789, 2013.

[11] D. Harel and R. Marelly. *Come, let's play: scenario-based programming using LSCs and the play-engine*, volume 1. Springer Science & Business Media, 2003.

[12] D. B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, 4(1):77–84, 1975.

[13] M. Jurdzinski, M. Paterson, and U. Zwick. A deterministic subexponential algorithm for solving parity games. *SIAM Journal on Computing*, 38(4):1519–1532, 2008.

[14] H. Kugler and I. Segall. Compositional Synthesis of Reactive Systems. *Synthesis*, pages 77–91, 2009.

[15] D. A. Martin. Borel determinacy. *Annals of Mathematics*, pages 363–371, 1975.

[16] N. Piterman, A. Pnueli, and Y. Sa. Synthesis of Reactive (1) Designs. (106), 2004.

[17] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive (1) designs. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 364–380. Springer, 2006.

[18] A. Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977.

[19] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 179–190. ACM, 1989.

[20] S. Safra. On the complexity of ω-automata. In *Foundations of Computer Science, 1988., 29th Annual Symposium on*, pages 319–327. IEEE, 1988.

[21] H. Tauriainen. Nested Emptiness Search for Generalized Büchi Automata. 70:127–154, 2006.

[22] W. Thomas, T. Wilke, et al. *Automata, logics, and infinite games: a guide to current research*, volume 2500. Springer Science & Business Media, 2002.

[23] S. Tripakis and K. Altisen. On-the-fly controller synthesis for discrete and dense-time systems. In *International Symposium on Formal Methods*, pages 233–252. Springer, 1999.

[24] W. Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theor. Comput. Sci.*, 200(1-2):135–183, June 1998.

Abbildungsverzeichnis

Tabellenverzeichnis

Liste der Algorithmen