Albert-Ludwigs-Universität Freiburg
Institut für Informatik
Lehrstuhl für Grundlagen der Künstlichen Intelligenz
Wintersemester 2007/2008
Diplomarbeit

Zufälliges Erstellen von Realzeit-Automaten im Uppaal-Format

Erstgutachter: Prof. Dr. Bernhard Nebel
Zweitgutachter: Prof. Dr. Andreas Podelski

Vorgelegt von:

Jan Rehm
Luisenstrasse 4
79312 Emmendingen
E-Mail: rehm@informatik.uni-freiburg.de

Matrikelnummer 1146881

Abgabe: 4. Februar 2008

Abstract

In dieser Arbeit geht es um den Entwurf und Vergleich verschiedenen Methoden, Realzeit-Automaten (engl. timed automata) zufällig derart zu generieren, dass die entstehenden Automaten als Testmuster interessant sind, mit denen man die Performanz des Model Checker Uppaal testen kann. Dazu werden bereits vorhandene Ansätze zum zufälligen Generieren von zufälligen deterministischen und nicht-deterministischen endlichen Automaten betrachtet und diese mit dem Generieren von Realzeit-Automaten verglichen, um Unterschiede in der Problemstellung aufzuzeigen und anschließend mögliche Lösungen für diese Probleme zu präsentieren. Das aus diesen Lösungen resultierende Programm generiert bei passenden Eingaben Testmuster und dazugehörige Queries für Uppaal, die mit einer 50/50 Wahrscheinlichkeit entweder als „satisfied“ oder „not satisfied“ gelten, eine Verteilung, die für das Model Checking interessant ist, da keine vorherige Aussage gemacht werden kann, ob das gestellte Problem lösbar ist oder nicht.

Erklärung

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Jan Rehm Freiburg, den 04.02.2008

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Idee, Motivation

Realzeit-Automaten (engl. timed automata) sind ein Konzept, vorgeschlagen von Rajeev Alur und David L. Dill [11], um reale Systeme mittels Automatentheorie modellieren und simulieren zu können, und gleichzeitig, anders als bei anderen Ansätzen wie Realzeit-Petrinetzen [6], Realzeit-Transitionssystemen [5, 17] oder Realzeit-I/O-Automaten [13], eine Sprachentheorie dieser Realzeit-Sprachen zu erhalten.
Ein Beispiel für eine solche Steuerung ist eine Steuerung für Strassenbahnen [14]. Dabei werden die simulierten Vorgänge nicht von ihrer zeitlichen Komponente auf eine Abfolge von Zuständen und Ereignissen abstrahiert; anstelle dessen wird das Konzept der endlichen Automaten um eine quantitative Zeitkomponente erweitert, um so Systeme zu modellieren, die der Interaktion mit zeitabhängigen physikalischen Prozessen abhängig sind.

Um dabei möglichst nahe an den physikalischen Vorbildern zu bleiben, wurde eine Repräsentation der Zeit als reelle Zahlen [1] gewählt; obwohl dies einfacher zu bewerkstelligen gewesen wäre, wurde eine Repräsentation durch Integer-Werte als zu ungenau und das System beschränkend zurückgewiesen, da Vorkommnisse in der Realität selten genau zeitlich festlegbar sind, wie es für eine Darstellung mit Integern nötig wäre.

1.1.1 Automaten-Solver als Model Checker

Eine Erweiterung dieses Konzepts durch Bengtsson/Yi [16], in Abwandlung einer Erweiterung durch Henziger [7] fügt diesem Model weitere Elemente hinzu, um reale Systeme darstellen zu können, aber die Definition der Automaten intuitiver zu gestalten.
Schließlich werden Realzeit-Automaten in der Anwendung im Programm *Uppaal* um ganzzahlige Variablen ergänzt, die neben den Zeit einen Einfluss auf den Ablauf des Automaten haben können.

[1] Zumindest konzeptionell sind Zeitwerte in Realzeit-Automaten reelle Zahlen. Bei der Darstellung von Zeitwerten im Rechner finden Methoden Anwendung, um ganze *Bereiche* von Zeitwerten zusammenzufassen. siehe [16]

Dies erlaubt es, dieses Konzept zur Handlungsplanung heranzuziehen, da modellierte physikalische Systeme nun auf ihre Eigenschaften überprüft und ihr Verhalten simuliert werden kann.

Verwendung findet dieses Verfahren unter anderem im AVACS-Projekt (engl. Automatic Verification and Analysis of Complex Systems) [1], wo es zur Systemverifikation von Modellen komplexer sicherheitskritischer Computer-betriebener Systeme benutzt wird, wobei komplexe System erst in einen Realzeit-Automaten abstrahiert werden, und dann mittels geeigneter Softwaretools auf die gewünschten Eigenschaften überprüft werden.

Die Idee, Realzeit-Automaten zufällig zu generieren, beruht auf dem Bedarf, ausreichend Testmuster zur Erprobung und Entwicklung solcher Testverfahren zu gewinnen. Dabei wird in dieser Ausarbeitung das Format von Uppaal [15] verwendet, um Realzeit-Automaten darzustellen.

2 Grundlagen

2.1 Endliche Automaten

Realzeit-Automaten (engl. timed automata) sind eine Erweiterung von endlichen Automaten, daher ist es nötig, diese vorher zu erklären.

Ein endlicher Automat ist ein Modell des Verhaltens, bestehend aus Zuständen, Zustandsübergängen und Aktionen.

Endlich heißt der Automat deshalb, weil die Anzahl seiner inneren Zustände endlich ist. Zudem ist das Alphabet, über dem dieser Automat arbeitet, endlich.

Ein endlicher Automat akzeptiert Wörter einer zugehörigen regulären formalen Sprache $\mathcal{A}$, wobei ein Wort als eine Folge von Symbolen definiert ist [**?**, 8].

Wir werden mehrere Arten von Automaten definieren; diese unterscheiden sich in der Regel nur durch die Definition der Endzustände und Akzeptanzbedingungen.

Wir halten uns daher an die Definition von Alur/Dill [11], indem wir die sonst üblichen Komponenten eines Automaten als sogenannte *Übergangstabelle* zusammenfassen.

2.2 Formale Sprachen

Sei Σ ein Alphabet, also eine endliche Menge von Symbolen. Eine formale Sprache über Σ ist jede beliebige Teilmenge von Σ^*.

Ein Wort σ der Länge n ist eine Folge $(x_1, ..., x_n)$, $x_i \in \Sigma$, $1 \leq i \leq n$.

2.3 Übergangstabelle

Definition 1. Eine *Übergangstabelle* (engl. transition table) $\mathcal{A}$ ist ein Tupel (Σ, S, S_0, E):

- Σ, ist das Alphabet;
- S ist eine nicht-leere, endliche Menge an Zuständen;
- S_0, ist eine nicht-leere, endliche Menge der Start-Zustände, $S_0 \subseteqq S$

- $E \subseteq S \times \ S \times \ \Sigma$ ist die Menge der Kanten, geschrieben $< s, s', a > \in$ E.

Der Automat startet in s_0 und falls $< s, s', a > \in$ E kann der Automat von Zustand s in Zustand s' wechseln, wenn er das Symbol a einliest.

Eine Übergangstabelle ist *deterministisch*, wenn:

1. Es nur einen einzigen Startzustand gibt, d.h. $|S_0| = 1$.
2. Für jedes $s \in S$ und $a \in \Sigma$ existiert höchstens ein s' sodass $< s, s', a > \in E$.

Andernfalls ist sie nicht-deterministisch.

Eine Übergangstabelle, und der daraus resultierende Automat, ist *vollständig*, wenn es für jeden Zustand $s \in S$ und jedes Symbol $a \in \Sigma$ eine Kante in E gibt.

Definition 2. Ein *Lauf* (engl. *run*) r von $\mathcal{A}$ über das Wort $\sigma = (\sigma_1, ..., \sigma_n)$, gegeben $s_0 \in S_0$ und $< s_{i-1}, s_i, \sigma_i > \in E$ für alle $i \geq 1$, ist eine Folge von Zuständen aus S.

2.4 Deterministische endliche Automaten

Definition 3. Ein *deterministischer endlicher Automat* (engl. deterministic finite automata, DFA) $\mathcal{A}$ ist eine deterministische Übergangstabelle (Σ, S, S_0, E) mit einer zusätzlichen endlichen, nichtleeren Menge $F \subseteq S$ an *Endzuständen*.

Ein Wort wird dann von $\mathcal{A}$ angenommen, wenn ein Lauf von $\mathcal{A}$ über dieses Wort in einem Zustand aus F endet.

Abb. 2.1 zeigt einen DFA. *Z0* ist hier der Startzustand, *Z3* der Endzustand, jeweils speziell gekennzeichnet durch den Kasten um den Endzustand und den Pfeil, der auf den Startzustand zeigt.

Die von diesem Automaten akzeptierte Sprache ist:

$$\mathcal{L} := \{X \in \{a, b\}^* | (\text{Anzahl a's in x})\text{- }(\text{Anzahl b's in x}) \equiv 3(\mod 4)\}$$

2.5 Nicht-deterministische endliche Automaten

Definition 4. Ein *nicht-deterministischer endlicher Automat* (engl. nondeterministic final automata, NFA) $\mathcal{A}$ ist eine nicht-deterministische Übergangstabelle (Σ, S, S_0, E) mit einer zusätzlichen endlichen, nichtleeren Menge $F \subseteq S$ an Endzuständen.

Ein Wort wird dann von $\mathcal{A}$ angenommen, wenn ein Lauf von $\mathcal{A}$ über dieses Wort in einem Zustand aus F endet.

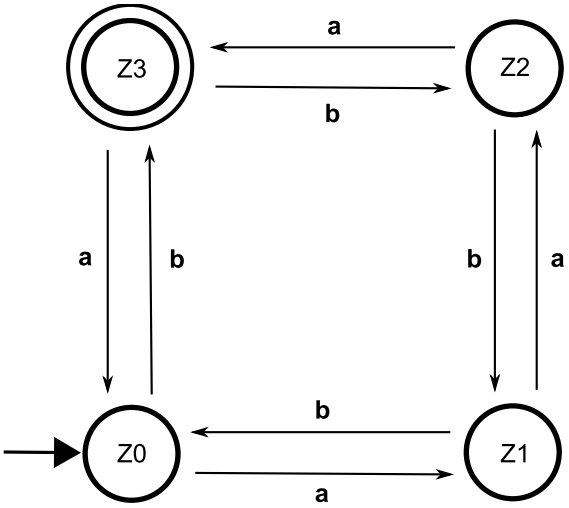


Abbildung 2.1: Ein einfacher DFA

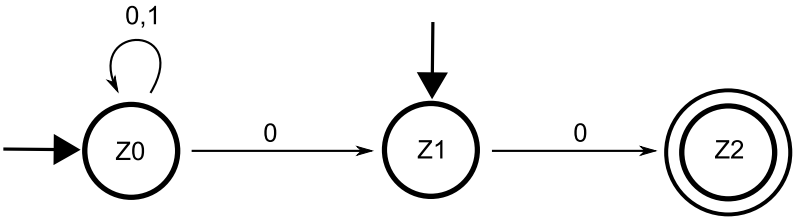


Abbildung 2.2: Ein einfacher NFA

Wird eine Sprache von einem DFA oder NFA akzeptiert, d.h jedes ihrer Wörter wird akzeptiert, ist die Sprache *regulär*.

Abb. 2.2 zeigt einen NFA. *Z0* und *Z1* sind hier Startzustände, *Z2* der Endzustand, jeweils speziell gekennzeichnet.

Die von diesem Automaten akzeptierte Sprache ist:

$$\mathcal{L} := \{X \in \{0,1\}^* | \text{endet mit } 00, \text{ oder ist nur } 0\}$$

2.6 ω-Sprachen

Entgegen der eher bekannten Definition von formalen Sprachen, besteht eine ω-Sprache aus unendlichen Wörtern.

Daher ist eine ω-Sprache über einem endlichen Alphabet Σ eine Teilmenge von Σ^ω — der Menge aller unendlichen Worte über Σ.

Definition 5. Für einen *Lauf* besteht die Menge *inf(r)* aus den Zuständen $s \in S$ derart, dass $s = s_i$ für beliebig viele $i \geq 0$, d.h. ein beliebig langes Wort führt zu einem beliebig langen *Lauf*.

2.7 ω-Automaten

Ein ω-Automat (Omega-Automat) ist ein mathematisches Modell, das eine Erweiterung des endlichen Automaten auf die Eingabe unendlicher Wörter darstellt.

Ein ω-Automat ist im Prinzip dasselbe wie ein entsprechender endlicher Automat, allerdings sind die Akzeptanz-Bedingungen derart angepasst, dass sie unendliche Wörter behandeln.

2.8 Büchi-Automaten

Der *Büchi-Automat* (engl. Büchi automata) (nach dem Schweizer Mathematiker Julius Richard Büchi) ist eine spezielle Form der ω-Automaten[11].

Definition 6. Ein *Büchi-Automat* $\mathcal{A}$ ist eine entweder deterministische oder nicht-deterministische Übergangstabelle (Σ, S, S_0, E) mit einer zusätzlichen Menge $F \subseteq S$ an Endzuständen.

Ein *Lauf* r von $\mathcal{A}$ über das Wort σ wird dann von $\mathcal{A}$ angenommen, wenn *inf(r)* $\cap$ $F \neq \emptyset$, d.h. ein *Lauf* r wird dann akzeptiert, wenn einer der Zustände aus F entlang r unendlich oft durchlaufen wird.

2.9 Muller-Automaten

Definition 7. Ein *Muller-Automat* (engl. Muller automata) $\mathcal{A}$ ist eine entweder deterministische oder nicht-deterministische Übergangstabelle (Σ, S, S_0, E) mit einer zusätzlichen Menge $F \subseteq 2^S$ an Endzustandsmengen, der so genannten *Akzeptanz-Familie* (engl. acceptance family).

Ein *Lauf* r von $\mathcal{A}$ über das Wort σ wird dann von $\mathcal{A}$ angenommen, wenn *inf(r)* $\in F$, d.h. ein *Lauf* r wird dann akzeptiert, wenn die Menge der Zustände die entlang r unendlich oft durchlaufen wird gleich einer der Mengen in F ist.

2.10 Die Sprache eines Realzeit-Automaten

Wir definieren die Sprache derart, dass jedes Wort der Sprache des Automaten wie bei DFA oder NFA mit einem Verhalten des Automaten korreliert.

Dazu wird ein nicht-negativer Zahlenbereich $\mathbb{R}$ als Bereich der Zeit gewählt; jedes Wort w einer Zeit-Sprache (engl. timed language) ist mit einer zugehörigen Sequenz an Zeitwerten z gepaart.

Definition 8. Eine Zeitsequenz $\tau = \tau_1, \tau_2...$ ist eine unbegrenzte Folge an Zeitwerten $\tau_i \in \mathbb{R}, \tau_i > 0$, mit folgenden Einschränkungen:

- *Monotonie*: τ steigt streng monoton; $\tau_i \leq \tau_{i+1}$ für alle $i \geq 1$
- *Fortschritt*: $\forall t \in \mathbb{R}\ \exists i \geq 1, \tau_i > t$

Ein Zeitwort über einem Alphabet Σ ist ein Tupel (σ, τ); $\sigma = \sigma_1, \sigma_2...$ ist ein unbeschränktes Wort über Σ und τ eine Zeitsequenz. Eine Zeitsprache ist somit eine Sammlung an Zeitwörtern über Σ.

In einem Realzeit-Automaten repräsentiert das Wort (σ, τ) die Eingabe des Symbols σ_i zur Zeit τ_i.

2.11 Transitions-Tabelle mit Zeiteinschränkungen

Diese nötige Erweiterung der Übergangstabelle fügt die Möglichkeit hinzu, dass die Wahl des Folgezustandes eines Automaten durch die verstrichene Zeit zwischen den Eingaben der Symbole eines Wortes beeinflusst wird.

Wenn im Vergleich mit einem DFA die Wahl des Folgezustandes nur von der Kombination des aktuellen Zustandes und des eingegebenen Symbols abhängt, ist diese Wahl im Realzeit-Automaten vom aktuellen Zustand, eingegebenem Symbol und verstrichener Zeit abhängig.

Um dies zu ermöglichen, werden dem System eine endliche Anzahl an Uhren mit jeder Übergangstabelle hinzugefügt.

Eine Uhr kann während einer Transition auf Null zurückgesetzt werden; ansonsten enthält die Uhr den Wert der verstrichenen Zeit seit dem letzten Reset.

So erhält jede Transition eine Uhr-basierte Zeit-Beschränkung, die besagt, dass die Transition nur dann gewählt werden darf, wenn der Wert der benannten Uhr der geforderten Bedingung entspricht.

2.12 Uhren-Bedingungen und Uhren-Interpretation

Die einfachste Form einer *Uhren-Bedingung* (engl. clock constraint) ist ein einfacher numerischer Werte-Vergleich zwischen Uhr und Bedingungs-Wert.

Definition 9. Für eine Menge C an Uhren wird die Menge $\phi(C)$ an Uhren-Bedingungen δ folgendermaßen definiert:

$$\delta := x \leq c \mid x \geq c \mid \ -\delta \mid \delta_1 \ \wedge \ \delta_2$$

x ist in diesem Fall eine Uhr aus C und c eine Konstante aus $\mathbb{R}^+$.

Eine *Uhren-Interpretation* ν für eine Menge C an Uhren weist jeder Uhr einen reellen Wert zu; sie ist eine Funktion von C nach $\mathbb{R}$.

Es wird gesagt, eine Uhren-Interpretation ν für C genügt einer Uhren-Bedingung δ über C falls δ wahr wird gegeben den Werten von ν.

2.13 Realzeit-Übergangstabelle (timed transition table)

Definition 10. Eine Realzeit-Übergangstabelle $\mathcal{A}$ ist ein Tupel (Σ, S, S_0, C, E).

- Σ ist ein endliches Alphabet
- S ist eine endliche Menge an Zuständen, auch Lokationen genannt.
- S_0 ist die Menge der Startzustände, $S_0 \subseteq S$
- C ist endliche Menge an Uhren (clocks)
- $E \subseteq S \times \ S \times \ \Sigma \times \ 2^C \times \ \phi(C)$ ist die Menge der Kanten.

Eine Kante $(s, s', a, \lambda, \delta)$ ist eine Kante von s nach s' mit Eingabe-Symbol a. Die Menge $\lambda \subseteq C$ ist die Menge an Uhren die bei der Transition auf Null zurückgesetzt werden, und δ ist eine Uhren-Bedingung über C.

Eine Realzeit-Übergangstabelle ist *deterministisch*, wenn:

1. Es nur einen einzigen Startzustand gibt, d.h. $|S_0| = 1$.
2. Für alle $s \in S$, für alle $a \in \Sigma$, für jedes Paar von Kanten der Form $< s, -, a, -, \delta_1 >$ und $< s, -, a, -, \delta_2 >$, sind die Uhren-Bedingungen δ_1 und δ_2 gegenseitig ausschließend, d.h. $\delta_1 \wedge \delta_1$ kann nicht erfüllt werden.

Andernfalls ist sie nicht-deterministisch. Ein Zeitautomat ist deterministisch, wenn seine Übergangstabelle deterministisch ist.

Eine Realzeit-Übergangstabelle, und der daraus resultierende Automat, ist *vollständig*, wenn es für jeden Zustand $s \in S$ und jedes Symbol $a \in \Sigma$ eine Kante in E gibt.

Entsprechend muss die Definition des *Laufs* angepasst werden:

Definition 11. Ein *Lauf* r, beschrieben durch $(\bar{s}, \bar{\nu})$, von $\mathcal{A}$ über das Realzeit-Wort (σ, τ), gegeben $s_i \in S$ und $\nu_i \in [C \to \mathbb{R}]$ für alle $i \geq 0$, mit den folgenden Bedingungen:

- *Beginn*: $s_0 \in S_0$ und $\nu_0(x) = 0$ für alle $x \in C$.
- *Abfolge*: für alle $i \geq 1$ gibt es eine Kante in E der Form $< s_{i-s}, s_i, \sigma_i, \lambda_i, \delta_i >$ so dass $(\nu_{i-1} + \tau_i - \tau_{i-1})$ der Bedingung δ_i genügt und $\nu_i = [\lambda_i \to 0](\nu_{i-1} + \tau_i - \tau_{i-1})$.

Für einen solchen *Lauf* besteht die Menge *inf(r)* aus den Zuständen $s \in S$ derart, dass $s = s_i$ für unendlich viele $i \geq 0$, d.h. ein beliebig langes Wort führt zu einem unendlich langen *Lauf*.

Es muss beachtet werden, dass in der Definition von Alur/Dill [11] ein RZA entweder ein Büchi-Automat oder ein Muller-Automat ist, d.h. ein Wort wird nur dann angenommen, wenn es abzählbar unendlich ist, und der Lauf des Wortes einen der Endzustände unendlich oft durchläuft.

Dies ist nötig, da es in der vorliegenden Definition möglich ist, dass ein Automat beliebig lange in einem Zustand verharrt, und nicht auf den Endzustand zustrebt.

2.14 Realzeit-Büchi-Automaten

Definition 12. Ein *Realzeit-Büchi-Automat* (engl. timed Büchi automata, TBA) $\mathcal{A}$ ist eine entweder deterministische oder nicht-deterministische Realzeit-Übergangstabelle (Σ, S, S_0, C, E) mit einer zusätzlichen endlichen Menge $F \subseteq S$ an Endzuständen. Ein *Lauf* $r = (\bar{s}, \bar{v})$ von $\mathcal{A}$ über das Wort (σ, τ) wird dann von $\mathcal{A}$ angenommen, wenn *inf(r)* $\cap\, F \neq \emptyset$, d.h. ein *Lauf* r wird dann akzeptiert, wenn einer der Zustände aus F entlang r unendlich oft durchlaufen wird.

2.15 Realzeit-Muller-Automaten

Definition 13. Ein *Realzeit-Muller-Automat* (engl. timed Muller automata, TMA) $\mathcal{A}$ ist eine entweder deterministische oder nicht-deterministische Realzeit-Übergangstabelle

(Σ, S, S_0, C, E) mit einer zusätzlichen endlichen Menge $F \subseteq 2^S$ an Endzustandsmengen, der *Akzeptanz-Familie* (engl. acceptance family).

Ein *Lauf* $r = (\bar{s}, \bar{v})$ von $\mathcal{A}$ über das Wort (σ, τ) wird dann von $\mathcal{A}$ angenommen, wenn *inf(r)* $\in F$, d.h. ein *Lauf* r wird dann akzeptiert, wenn die Menge der Zustände die entlang r unendlich oft durchlaufen wird gleich einer der Mengen in F ist.

2.16 Realzeit-Sicherheits-Automaten

Bengtsson/Yi [16] wandeln Henzinger [7] folgend, die Definition von RZA folgendermaßen ab: Diese Automaten sind, um einen mehr intuitiven Aufbau zu gestatten, nicht länger Büchi-Automaten, sondern weisen ein zusätzliches Element auf, um den Ablauf des Automaten zu steuern, indem sie lokal, also auf Knotenebene, den Fortschritt des Automaten erzwingen.

So ist es nun möglich, Uhren-Bedingungen auch an Knoten zu vergeben; dadurch kann verhindert werden, dass der Automat in einem Knoten verharrt. Diese Bedingungen werden *Invarianten* genannt.

Man interessiert sich bei diesen Automaten weniger für die Sprache, als vielmehr für die Erreichbarkeit bestimmter Zustände, da etwa die Erreichbarkeit bestimmter kritischer Zustände einem realen Unglückszustand (also einem Fehler des Systems) entsprechen kann.

Definition 14. Ein *Realzeit-Sicherheits-Automat* (engl. timed safety automata, RZSA) $\mathcal{A}$ ist eine entweder deterministische oder nicht-deterministische Realzeit-Übergangstabelle (Σ, S, S_0, C, E) mit folgenden Zusätzen und Änderungen:

- $\mathcal{B}(\mathcal{C})$ ist die Menge der möglichen Uhren-Bedingungen, in folgenden Guards genannt
- $E \in N \times \mathcal{B}(\mathcal{C}) \times \Sigma \times 2^C \times N$ ist die Menge der Kanten
- $I : N \to \mathcal{B}(\mathcal{C})$ ist die Zuweisung von Invarianten an Knoten

Bei Alur/Dill [11] dient das Konzept der Endzustände (mit der Bedingung, dass einer von ihnen unendlich oft durchlaufen werden muss) dazu, Fortschritt zu erzwingen, also zu verhindern, dass ein Automat ewig in einer Lokation bleiben kann. Durch Invarianten kann man den Fortschritt lokal (also pro Lokation) erzwingen; somit kann man auf das Konzept Endzustand verzichten (vergleiche Abb.2.3).

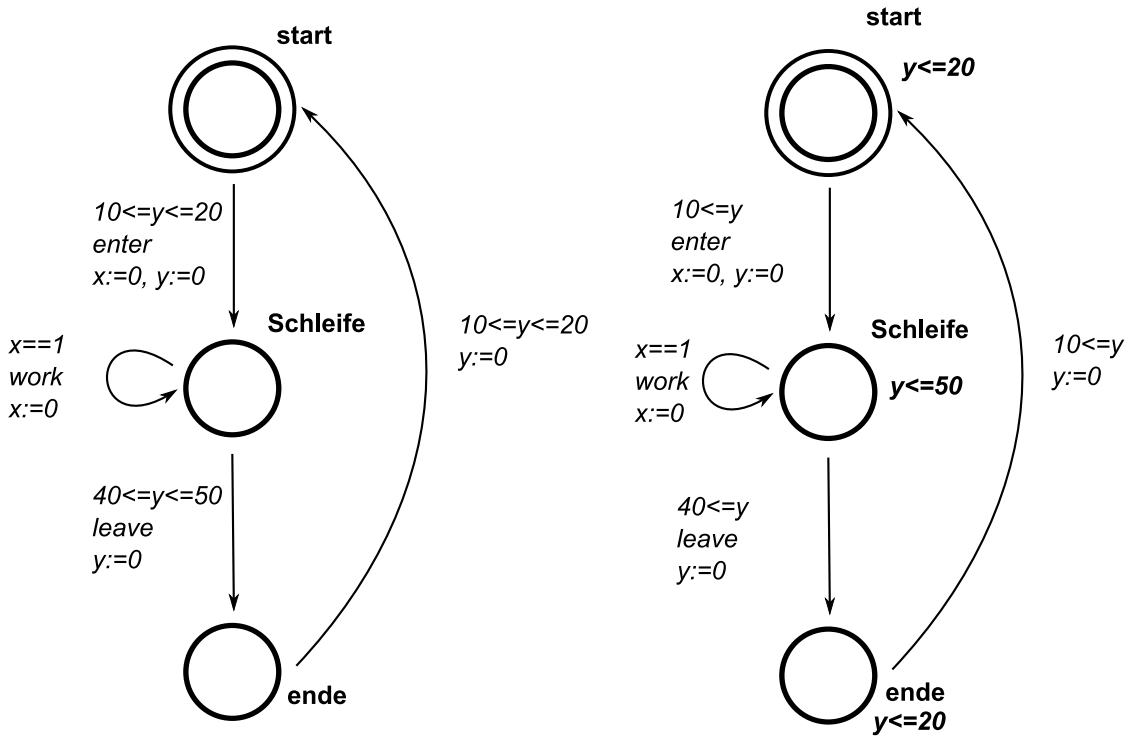


Abbildung 2.3: links ein RZA, rechts der entsprechende RZSA

2.17 Uppaal

Uppaal ist ein Werkzeug zum Modellieren, Simulieren und Überprüfen von Realzeit-Automaten, entwickelt in Zusammenarbeit an den Universitäten Uppsala und Aalborg [15].

Um die Modellierung von realen Systemen (insbesondere die Kommunikation zwischen Teilprozessen) zu erleichtern, kommt in der Uppaal-Version der RZA ein neues Element hinzu: das der Integer-Variablen. Diese beinhalten, wie der Name schon sagt, Integer-Werte, und sind auf während der Definition des Systems bestimmte Wertebereiche festgelegt.

Des weiteren wird der Begriff der Guards erweitert; er betrifft nun nicht mehr nur Bedingungen an die Uhren des Systems, sondern auch Bedingungen, die auf Integer-Variablen basieren.

Definition 15. Für eine Menge X an Variablen wird die Menge $\phi(X)$ an Variablen-Bedingungen δ folgendermaßen definiert:

$$\delta := x \leq c \mid x \geq c \mid \ -\delta \mid \delta_1 \ \wedge \ \delta_2$$

x ist in diesem Fall eine Variable aus X und c eine Konstante aus $\mathbb{R}$.

Dabei stellt den Kern der Uppaal-Modellierung die Modellierung von Netzwerken von RZA dar, um so reale Systeme darstellen zu können, indem den verschiedenen beteiligten Elementen (Agenten, Umgebung etc.) eines Systems jeweils ein eigener Automat zugeordnet wird.

Diese laufen parallel; die einzelnen Automaten (in der Uppaal-Terminologie: Prozesse) einer Komposition aus Automaten (in der Uppaal-Terminologie: System) werden gleichzeitig gestartet und jeder Teilprozess kann entweder eine interne Transition machen, oder zwei Prozesse können sich synchronisieren, wie wir bald erläutern.

Um dies zu ermöglichen, weist die Modell-Sprache von Uppaal zusätzliche Werkzeuge auf, um auch Kommunikation zwischen Automaten zu erlauben:

Synchrone Kommunikation mittels CCS parallel composition operator [4] und *asynchrone Kommunikation* mittels gemeinsamer (globaler) Variablen.

Um die synchrone Kommunikation zu modellieren, enthält das Alphabet Eingabe-Symbole der Form $a?$ und Ausgabe-Symbole der Form $a!$. Ein solches Paar von $(a?, a!)$ wird im folgenden auch Channel genannt. Diese Symbole werden im Ablauf des Systems benutzt, um entsprechende Kanten zu sperren oder zu entsperren.

Dabei können mit einem Eingabesymbol ($a?$) versehene Kanten nur dann von einem Automaten benutzt werden, wenn zeitgleich von einem anderen Automaten eine mit

einem ($a!$) versehene Kante benutzt wird. Zusätzlich enthält das Alphabet das speziellen Symbol τ, es repräsentiert eine interne Aktionen.

Asynchrone Kommunikation findet über globale Variablen statt; darunter fallen auch die Uhren, man kann sie in dieser Hinsicht als Variablen vom Typ „Uhr“ betrachten. Globale Variablen werden wie lokale Variablen in den Guards und Invarianten der Automaten verwendet, und so kann es vorkommen, dass Kanten eines Automaten ge- oder entsperrt werden, obwohl dieser im gleichen Zustand verblieben ist. Um die Werte der Variablen des Systems beeinflussen zu können, ist jede Kante zudem in der Lage, die Werte von Uhren und Integer-Variablen mittels *Zuweisungen* zu verändern. Dies entspricht der internen Aktion, die mit τ bezeichnet ist, und hat die Form $x = c$ oder $x = x \pm c$, wobei x eine Integer-Variable sei, und c eine natürliche Zahl.

Definition 16. Ein Uppaal-Automat $\mathcal{A}$ ist eine nicht-deterministische Realzeit-Übergangstabelle (Σ, S, S_0, C, E) mit folgenden Zusätzen und Änderungen:

- Σ ist ein endliches Alphabet, das Aktionen der Form $a!$ oder $a?$ oder τ enthält, wobei zu jedem $a!$ ein $a?$ existiert und umgekehrt;
- V ist eine endliche Menge an Integer-Variablen;
- $\mathcal{B}(\mathcal{C})$ ist die Menge der möglichen Uhren-Bedingungen, in folgenden (Uhren-)Guards genannt;
- $\mathcal{B}(\mathcal{V})$ ist die Menge der möglichen Variablen-Bedingungen, im folgenden (Variablen-)Guards genannt;
- $E \in N \times \mathcal{B}(\mathcal{C}) \times \mathcal{B}(\mathcal{V}) \times \Sigma \times 2^C \times N$ ist die Menge der Kanten
- $I : N \to \mathcal{B}(\mathcal{C})$ ist die Zuweisung von Invarianten an Knoten

Wie auch andere Realzeit-Automaten setzen Uppaal-RZA (im folgenden *URZA*) Uhren ein, d.h. globale Variablen, die einen Wert $x \geq 0$ haben.

Um die Überprüfung von modellierten Systemen zu erlauben, ist Uppaal in der Lage, eine Teilmenge der TCTL Formeln [9] für RZA zu überprüfen.

Die Formeln sind dabei der folgenden Form:

- A — Es gilt in jedem Pfad ϕ.
- $E \diamond \phi$ — Es existiert ein Pfad, für den irgendwann einmal ϕ gilt.
- $A \diamond \phi$ — Es gilt für alle Pfade irgendwann ϕ.

- E — Es gilt möglicherweise immer ϕ.
- $\phi -- > \psi$ — ϕ führt immer zu ψ.

Dabei sind ϕ und ψ lokale Eigenschaften, die in einem Zustand überprüft werden können, also boolesche Ausdrücke bezüglich Knoten und Integer-Variablen, oder Uhren-Bedingungen aus $B(C)$.

Dabei werden gewöhnlich $A[]\psi$ und $E <> \phi$ für die Überprüfung von Systemen benutzt, um festzulegen, dass eine gewisse Kombination von Knoten in den Automaten (also der Fehler, siehe Abschnitt 3.3, Seite 20) nicht gleichzeitig eingenommen werden kann — der Fehler also nicht eintritt.

3 Bereits existierende Zufallsgenerator-Ansätze für endliche Automaten

3.1 Zufallsgenerierungs-Ansätze für NFA

Champarnaud et al. [12] schlagen die Erstellung von NFA mittels einer geeignet langen binären Zahl (hier Bitstream genannt) vor. Basierend auf der Länge des Alphabet (m) und der gewünschten Anzahl an Zuständen (n) wird hierzu eine Binärzahl der Länge $m * n^2$ benötigt.

Ohne Beschränkung der Allgemeinheit geht der Ansatz davon aus, dass sowohl die Alphabetsymbole als auch die Zustände einfach durchnummeriert sind, also $\Sigma = \{1, .., m\}$ und $Q = \{1, .., n\}$. In diesem binären Tupel $(x_1, .., x_{m*n^2})$ kommen 0 und 1 in jedem x_i mit gleicher Wahrscheinlichkeit vor, also 0,5.

Basierend auf diesem Tupel wird nun festgelegt, welche Kanten im Automaten vorkommen:
Steht eine 1 an Stelle $(l-1) * n^2 + (i-1)n + j$, besteht eine Kante von i nach j mit Symbol l, $l \in \Sigma$, bei einer 0 besteht keine Kante. Der Anfangszustand ist immer Zustand 1, der Endzustand wird per Gleichverteilung aus allen Zuständen gewählt.

Ein Beispiel:

Gegeben sei die Zahl 100101101100001101.

Eine der resultierenden Rechnungen ist:

$$l = 1, i = 1, j = 1 : \ (1-1)3^2 + (1-1)3 + 1 = 1$$

Da Stelle x_1 des Tupels eine 1 enthält bedeutet dies, dass die Kante E(1,1,1) existiert.

Der daraus resultierende NFA ist in Abb. 3.1 dargestellt.

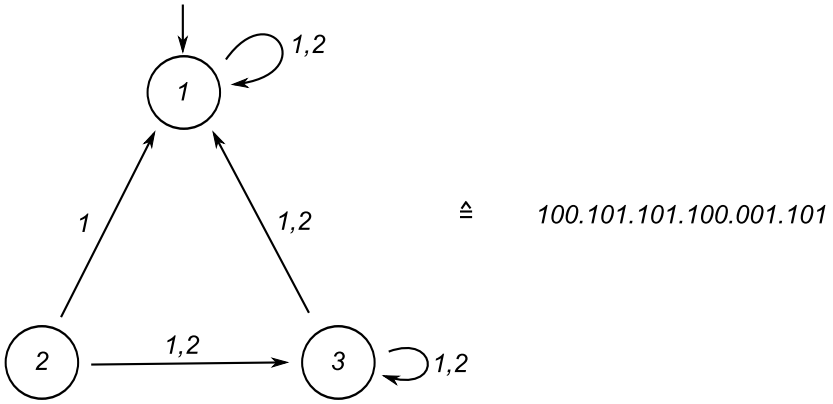


Abbildung 3.1: NFA nach der Bitstream-Methode

3.2 Zufallsgenerierungs-Ansätze für DFA

Bassino und Nicaud [10] stellen einen einfachen Ansatz zum Generieren von vollständigen DFA vor.

Sie bedienen sich dabei dem k-Dyck Box-Diagramm (engl. k-Dyck *boxed diagramm*), einer Struktur die sich mittels der vorgestellten Algorithmen bijektiv in einen zugehörigen DFA umwandeln lässt.

Definition 17. Ein k-Dyck Boxdiagramm der Größe n ist ein Paar von Tupeln von natürlichen Zahlen der Länge $(k-1)*n+1$.

$$(x_1, ..., x_{(k-1)*n+1}), (y_1, ..., y_{(k-1)*n+1})$$

Dabei sind folgende Regeln für die Zahlen in den Tupel zu beachten:

$$x_i \geq i$$

$$1 \leq y_i \leq n$$

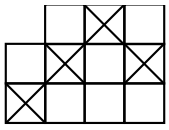


Abbildung 3.2: ein 2-Dyck Diagramm Größe 3

Abb 3.2 zeigt das Diagramm zu ((2,3,3,3),(1,2,3,2)).

Der Algorithmus 1 ist der zugehörige Vorgang, um solche Diagramme in DFA umzuwandeln.

Algorithm 1 k-DyckBoxedToTransitionStrukture

Require: Max[], Boxed[], A = Alphabet des Automaten

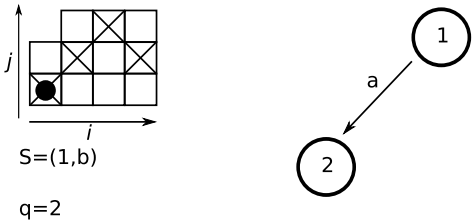
```
1:  S := empty stack; q:=1                          ▷ q ist letzter erzeugter Zustand
2:  Erzeuge Anfangszustand 1
3:  for all a ∈ A in umgekehrter lexikalischer Reihenfolge do
4:      Push(q,a) into S                ▷ fügt die fehlenden Übergange des Startzustandes ein
5:  end fori:=1;j:=1
6:  while S nicht leer do
7:      (p,a)=Pop(S)
8:      if j<Max[i] then //neuer Zustand wird erzeugt
9:          q:=q+1
10:         Erzeuge neuen Zustand q
11:         Erzeuge Kante E(p,q,a)
12:         for all a ∈ A in umgekehrter lexikalischer Reihenfolge do
13:             Push(q,a) into S        ▷ fügt die fehlenden Übergange des Zustandes q ein
14:         end for
15:         j:=j+1
16:     else if                         ▷ thenKante führt zu bereits existierendem Zustand
17:         Erzeuge Kante E(p,Boxed[i],a)
18:     end if
19: end while
```

Ein Beispiel eines solchen Vorganges:

Gegeben sei das Tupelpaar $((2,3,3,3),(1,2,3,2))$. Die beiden Tupel sind jeweils in den beiden Arrays Max[] und Boxed[] enthalten. A ist das Alphabet des Automaten der Länge k=2. Der DFA wird die Größe n=3 haben. Der daraus resultierende Ablauf ist in Abb. 3.3 bis 3.6 veranschaulicht.

S=(1,a);(1,b)

q=1



S=(1,b)

q=2

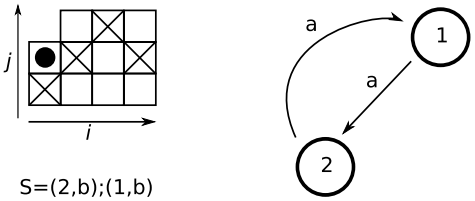
(p,a)=(1,a)

neuer Knoten q = 2

neue Kante E(p,q,a)=E(1,2,a)

j:=j+1

Abbildung 3.3: Schritt 1 und Schritt 2

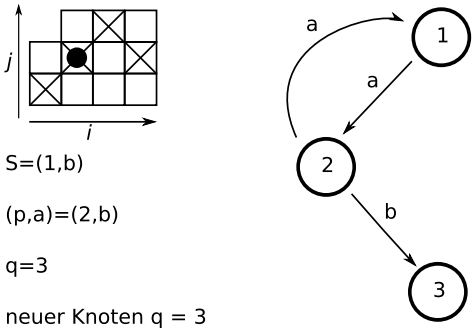


S=(2,b);(1,b)

(p,a)=(2,a)

j=Max[1]

neue Kante E(p,Boxed[1],a)
=E(2,1,a)



S=(1,b)

(p,a)=(2,b)

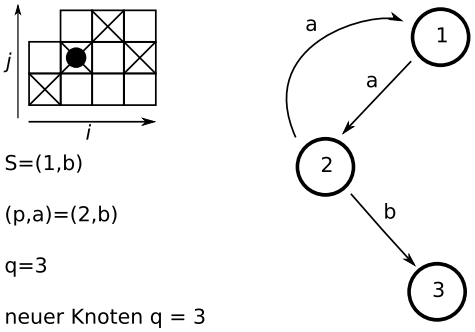
q=3

neuer Knoten q = 3

neue Kante E(p,q,a)=E(2,3,b)

S=(3,a);(3,b);(1,b)

Abbildung 3.4: Schritt 3 und Schritt 4



S=(1,b)

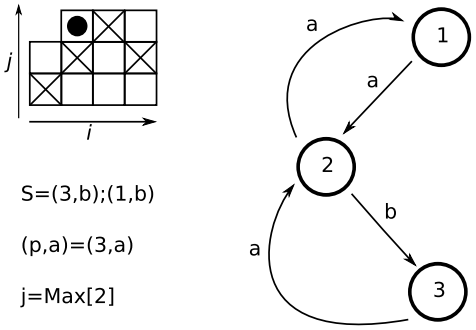
(p,a)=(2,b)

q=3

neuer Knoten q = 3

neue Kante E(p,q,a)=E(2,3,b)

S=(3,a);(3,b);(1,b)



S=(3,b);(1,b)

(p,a)=(3,a)

j=Max[2]

neue Kante E(p,Boxed[2],a)
=E(3,2,a)

Abbildung 3.5: Schritt 5 und Schritt 6

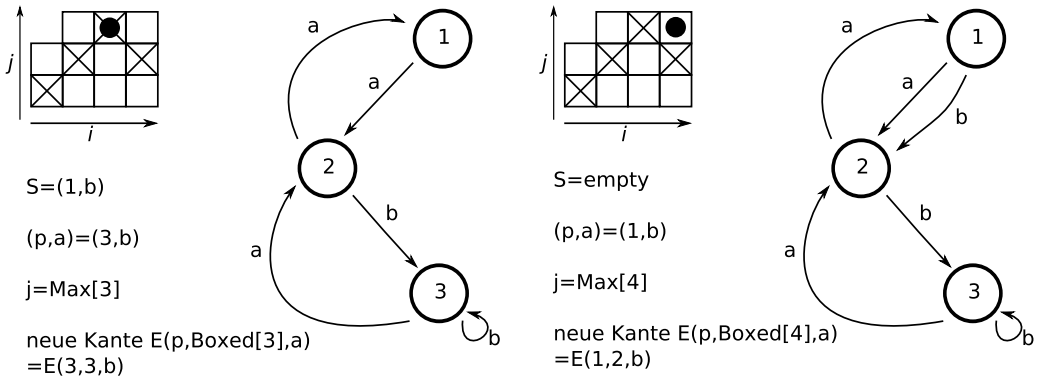


Abbildung 3.6: Schritt 7 und Schritt 8

3.3 Unterschiede bei der Generierung eines URZA

Der wesentliche Unterschied zwischen einem RZA und einem strukturell vergleichbaren NFA oder DFA ist letztendlich die Existenz von Variablen, ähnlich den hybriden Automaten[3], und die Tatsache, dass sowohl Knoten als auch Kanten abhängig von den Werten dieser Variablen offen oder gesperrt sein können.

Das von Uppaal verwendete Format verhält sich weitestgehend wie die Definition von Henzinger [7], d.h. das Alphabet des Automaten ist in Ein- und Ausgabewörter und ein internes Wort aufgeteilt, wobei die ersten beiden primär dem Zweck der Kommunikation zwischen Automaten desselben Systems dienen.

Des weiteren gibt es keinen Endzustand, aber einen Fehler, ein Kreuzprodukt aus je einem Zustand pro Automat des Systems, der ausdrückt, dass alle Automaten ihren jeweiligen Zustand des Fehlers gleichzeitig einnehmen müssen, damit es zum Fehler kommt. Obwohl es kein Ziel sein sollte, einen Fehler zu provozieren, ist es doch Absicht die Robustheit eines Systems dahingehend zu testen, ob es möglich ist, dass ein Fehler eintritt; somit wird der Fehler zum Ziel der Suche.

Die grundlegende Schwierigkeit beim Design eines sicheren Systems besteht darin, Automaten zu generieren, die zwar derartige Guards und Invarianten generieren, dass alle Zustände des Fehlers nicht gleichzeitig eingenommen werden können, ohne den grundsätzlichen Ablauf des Systems zu verhindern.

Es müssen also Guards und Invarianten erzeugt werden, die erfüllbar sein können, um den Ablauf eines echten Systems zu simulieren.

Das Problem mit den untersuchten Ansätzen ist, dass diese zwar Methoden zum Generieren der Struktur eines Automaten bieten, jedoch keine Möglichkeit, Guard und

Invarianten in geeigneter Weise wie oben beschrieben hinzuzufügen.

Eine pure Zufallszuweisung der Guards, basierend auf den gegebenen Wertebereichen der Variablen, würde sich nur in einem System lohnen, in dem alle Variablen auf sehr kleine Intervalle beschränkt sind - ansonsten wird, mit steigender Komplexität des Systems, die Erfüllbarkeit eines einzelnen Guards schnell gegen Null wandern, und somit das generierte System verstümmelt.

Es ist nötig, die Kanten bei Erstellen derart mit Guards zu versehen, dass die erstellten Guards eine wenn auch nur grobe Affinität zu den Werten haben, die das System bei Erreichen der Ursprungsknotens dieser Kanten vorraussichtlich einnehmen wird.

Auf diese Weise kann eine voraussichtliche Aussage über die Erfüllbarkeit einzelner Guards gemacht werden, und somit über die Möglichkeit, ob einzelne Kanten des Automaten während des Ablauf wählbar sind.

4 Zielsetzung - gewünschte Eigenschaften

4.1 Der Phasenübergang, die gewünschte Verteilung der Lösbarkeit der Testmuster

Wir interessieren uns für gewisse Erreichbarkeitseigenschaften der generierten Systeme; Es soll überprüft werden, ob ein Fehler im System, bestehend aus dem gleichzeitigen Einnehmen eines bestimmten Knotens je Automaten des Systems, eintreten kann oder nicht. Wir benutzen im Folgenden dann, wenn der Fehlerzustand in einem beteiligten Automaten erreichbar ist, die abkürzende Sprechweise „der Automat ist erreichbar".

Es nicht damit getan, Testmuster zu generieren, die in jedem Fall erreichbar oder unerreichbar sind; es ist vielmehr von Interesse, eine Gleichverteilung von Testmustern zu erhalten, so dass der Model Checker sowohl an positiven und negativen Beispielen getestet werden kann.

Deshalb ist es nötig, die Erzeugung von Testmustern derart zu kalibrieren, dass mit annähernd 50% Wahrscheinlichkeit ein Testmuster einer der beiden Arten erstellt wird.

4.2 Warum nur eine Zusammenhangskomponente?

Definition 18. Ein *Pfad* $\mathcal{L} = (a_1, ..., a_n)$ in einem Automaten (Σ, S, S_0, E, F) (siehe Kapitel 2) ist eine Liste an Knoten $\mathcal{L} \subseteq S$, für die gilt: für alle Paare $a_i, a_{i+1} \in \mathcal{L}$: es gibt eine Kante $\in$ E, die a_i und a_{i+1} verbindet.

Definition 19. Eine *Zusammenhangskomponente* in einem Automaten (Σ, S, S_0, E, F) (siehe Kapitel 2) ist eine Teilmenge von Knoten $M \subseteq S$, für die gilt: für alle paarweise verschiedenen $a, b \in M$ gilt: $\exists$ Pfad P der a und b verbindet. Wir sprechen von der *ersten Zusammenhangskomponente*, wenn für alle $s_0 \in S$ auch $s_0 \in M$ gilt.

Zwei Faktoren beeinflussen, ob ein Testmuster lösbar ist, d.h. ob der Fehler erreichbar ist: Zum einen muss ein Pfad an Kanten innerhalb jedes am System beteiligten Automaten bestehen, der vom Start-Knoten zum jeweiligen Fehler-Knoten führt.

Da der jeweilige Fehler-Knoten der Automaten beliebig gewählt wird, und somit jeder andere Knoten als Fehlerknoten in Frage kommt, muss ein solcher Pfad also zu jedem Knoten eines Automaten führen, d.h. die Knoten eines Automaten bilden innerhalb des gerichteten Graphen dieses Automaten eine einzige Zusammenhangskomponente.

Nur falls dies gewährleistet ist, ist die Existenz eines Pfades vom Start zum Fehler auf Graphenebene garantiert, also noch ohne Berücksichtigung von Guards und Invarianten.

Der andere beeinflussende Faktor sind die Guards der Kanten und die Invarianten der Lokationen; sie bestimmen durch einfache If-Then-Entscheidungen, wann und ob eine Lokation oder eine Kante des Automaten wählbar ist, um einen Pfad zum Fehler zu bilden.

Da wir in erster Linie einen Model Checker testen wollen, hat es wenig Sinn, einen Fehler dort zu deklarieren, wo er niemals eintreten kann, d.h. zuzulassen, das sich der jeweilige Fehlerzustand in einer zweiten zusammenhangskomponente befindet, in der er vom Start aus niemals, egal welcher Umstände, erreicht werden kann.

Daher wird in den hier dargestellten Lösungsansätzen in erster Linie sichergestellt, dass alle erzeugten Automaten voll zugänglich sind, d.h. sie nur eine einzige Zusammenhangskomponente besitzen, und die Erreichbarkeit des Fehlers einzig durch die Verlegung der Guards und Invarianten gesetzt wird.

4.3 Wie erzeugt man Guards und Invarianten?

Das größte Problem beim Erzeugen eines URZA: Es muss ein Weg gefunden werden, Guards und Invarianten zufällig zu generieren.

Dabei muss zwischen Integer-Guards, Uhren-Guards und Invarianten[1] eine Unterscheidung gemacht werden, da sowohl Uhren-Guards als auch Invarianten sich auf die Uhren des Systems beziehen, und damit auf einen reellwertigen Zahlenbereich.

Zudem verändern sich die Werte der Uhren mit dem Vergehen der Zeit, so dass es möglich ist, abzuwarten bis ein entsprechendes Guard erfüllt ist, so dass auch ein Guard der Form $x = c$, (x sei eine Uhr, c eine Konstante) eine realistische Chance hat, erfüllt zu werden, da es für das Nehmen einer Kante mit einem solchen Guard ausreicht, wenn x den gewünschten Wert c innerhalb eines verstreichenden Zeitraumes annimmt.

Daher ist das Erstellen von Uhren-Guards und Invarianten wesentlich unproblematischer als im Falle der Integer-Guards, da ein wesentliches Problem der Integer-Guards wegfällt.

Integer-Guards bestehen aus der Integer-Variablen nach der sie fragen, einem Vergleichsoperator aus der Menge $\{<, >, \leq, \geq, =\}$ und dem konstanten Integer-Wert mit dem verglichen wird. Uhren-Guards sind ähnlich aufgebaut; sie verwenden statt der Integer-Variablen eine der Uhren des Systems, vergleichen jedoch auch mit einem konstanten Integer-Wert.

Invarianten sind wie Uhren-Guards aufgebaut, allerdings ist die Auswahl der Operatoren auf $<$ oder $\leq$ beschränkt; da Zeit wächst und nicht schrumpft, ist eine Invariante $x > c$ sinnlos, denn sie kann nie falsch werden(sondern nur von vornherein falsch sein), und dient somit nicht der lokalen Flusssteuerung des Automaten. Das Wählen der Variablen kann mit einfacher Gleichverteilung entschieden werden; genauso der Vergleichsoperator (1 aus 5) - das wesentliche Problem stellt die Auswahl des Vergleichwertes dar. Sollte dafür eine Gleichverteilung über den Wertebereich der Variablen verwendet werden, ist die Wahrscheinlichkeitsverteilung wie im folgenden aufgeführt:

4.3.1 Vergleich möglicher Zufalls-Wahlen für die Werte der Guards

Im folgenden ist x der Name der Variablen, auf die sich das Guard bezieht und $(n-1)$ der zugehörige Maximal-Wert der Variablen. Per Konvention beginnen alle Wertebereiche der Variablen bei 0, der Bereich von 0 bis $n-1$ hat also n Elemente.

Es sei m der konstante Wert, der für das Guard per Gleichverteilung aus dem

[1] Diese sind normalerweise Uhren-Invarianten. Es existieren auch Integer-Invarianten; da diese aber weniger interessant sind, werden sie hier nicht betrachtet

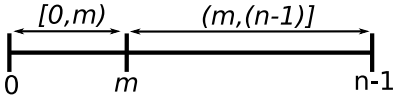


Abbildung 4.1: Wertebereiche bei einfacher Gleichverteilung

Wertebereich der Variablen gewählt wurde. Die Chance, dass ein Vergleich (und damit ein Guard) erfüllbar ist, hängt vom Operator ab:

$$P(x = m) : \frac{1}{n}$$

$$P(x > m) : \frac{(n-1) - m}{n}$$

$$P(x < m) : \frac{m}{n}$$

$$P(x >= m) : \frac{(n-1) - m + 1}{n} = \frac{(n - m}{n}$$

$$P(x <= m) : \frac{m+1}{n}$$

Da es fünf verschiedene Operatoren gibt, ist es nötig diese bei der Summierung jeweils mit $\frac{1}{5}$ zu multiplizieren. Somit ist die Gesamt-Chance, dass ein mit einfacher Gleichverteilung erzeugtes Guard erfüllt ist:

$$\frac{1}{5n} + \frac{(n-1) - m}{5n} + \frac{m}{5n} + \frac{(n-1) - m + 1}{5n} + \frac{m+1}{5n} = \frac{2n+1}{5n} = \frac{1}{5} * \frac{1}{n} + \frac{2}{5}$$

$$= \text{ Wahrscheinlichkeit, dass } (x = m) \text{ erfüllt ist } + \frac{2}{5}$$

Wie man leicht erkennen kann, sind, vor allem bei Variablen mit großen Wertebereichen, die Chancen im Fall $(x = m)$ sehr niedrig.

$$\lim_{n \to \infty} P(x = m) = \frac{1}{n} \to 0$$

Wird also m gleichverteilt aus dem definierten Bereich von x gewählt, ist die Chance, dass ein solches Guard tatsächlich erfüllt ist, mit $\frac{1}{n}$ umso kleiner, je größer der definierte Bereich von x ist.

Obwohl dies trotzdem eine Chance von mindestens 40% bedeuten sollte dass jedes Guard erfüllbar ist, muss bedacht werden, dass mehrere Guards an Kanten existieren können, und deren Chancen multipliziert werden müssen, da alle Bedingungen erfüllt

sein müssen. Ein einziges $(x = m)$-Guard mit $P(x = m) : \frac{1}{n}$ wäre also genug, um eine solche Kante sehr unwahrscheinlich zu machen. Für die Wahrscheinlichkeit einer Kante mit l Guards gilt folgendes:

$$P(G_1, ..., G_l) = \prod_{i=1}^{l} G_i$$

Falls ein $0 \leq i \leq l$ existiert mit $G_i \to 0$, gilt $\prod_{i=1}^{l} G_i \to 0$.

Da alle Operatoren gleich wahrscheinlich sind, kann davon ausgegangen werden, dass ein Fünftel der Integer-Guards eine derart geringe Wahrscheinlichkeit haben, dass der Automat trivial wird, da ein Großteil seiner Kanten blockiert sind.

Weniger gravierend ist dies bei den Uhren-Guards und Invarianten: Im Falle von Uhren-Guards wartet der Automat einfach ab, bis $(x = m)$ erfüllt ist falls $x < m$, ansonsten kann das Guard nicht erfüllt werden.

Bei den Invarianten werden gewöhnlich Uhren verwendet, und der =-Operator ist nicht zugelassen. Die betreffende Uhr schreitet einfach fort bis die Invariante nicht mehr erfüllt ist.

Um dieses Problem der Integer-Guards zu umgehen, kann mit eingeschränkten Bereichen gearbeitet werden: Sollte ein plausibler Wert für x bekannt sein, den die Variable im Ausgangsknoten der Kante des Guards einnimmt, kann ein Intervall um diesen Wert herum gelegt werden, und so die Auswahl für m verkleinert werden.

Wird m nun per Gleichverteilung aus dem eingeschränkten Bereich gewählt, verändern sich die Wahrscheinlichkeiten der Operatoren wie folgt:

$$P(x = m) : \frac{1}{2h+1}$$

$$P(x > m) : \frac{h}{2h+1}$$

$$P(x < m) : \frac{h}{2h+1}$$

$$P(x >= m) : \frac{h+1}{2h+1}$$

$$P(x <= m) : \frac{h+1}{2h+1}$$

Somit ist die Gesamt-Chance, dass ein mit derart beschränkter Gleichverteilung erzeugtes Guard erfüllt ist:

$$\frac{1}{5(2h+1)} + \frac{h}{5(2h+1)} + \frac{h}{5(2h+1)} + \frac{h+1}{5(2h+1)} + \frac{h+1}{5(2h+1)}$$
$$= \frac{4h+2+1}{5(2h+1)} = \frac{4h+2}{5(2h+1)} + \frac{1}{5(2h+1)} = \frac{1}{5} * \frac{1}{2h+1} + \frac{2}{5}$$
$$= \text{ Wahrscheinlichkeit, dass } (x = m) \text{ erfüllt ist } + \frac{2}{5}$$

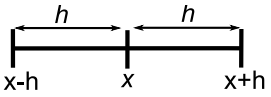


Abbildung 4.2: Wertebereiche bei eingeschränkter Gleichverteilung

Falls $2h + 1 << n$, ist dies eine einfache Methode, die Auswahl der möglichen Werte bei der Gleichverteilung klein zu halten, und so ein Guard zu erzeugen, dass auch im $(x = m)$-Fall noch gute Chancen hat.

Eine Alternative zur Gleichverteilung ist die Normalverteilung, da hier die Mitte der Zufallswahl durch das *Mittelwert* vorgegeben werden kann, und für dieses eine andere Wahrscheinlichkeit gilt.

Auch hier wird wieder ein beschränkendes Intervall um *Mittelwert* gelegt, wobei der plausible Wert für x die Rolle des *Mittelwert* annimmt.

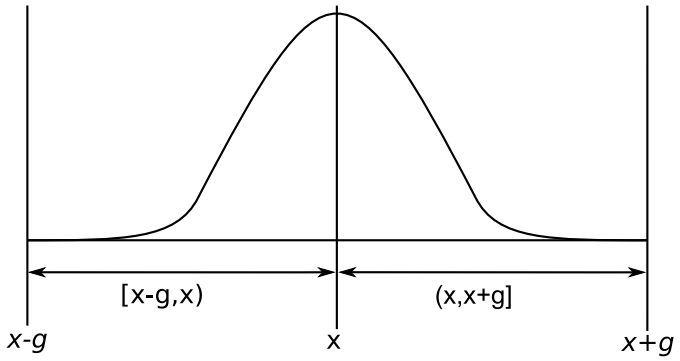


Abbildung 4.3: Wertebereiche bei Normalverteilung

In Abb.4.3 wird die Breite der Intervalle durch g bestimmt, was einer dreifachen Varianz (σ) der Normalverteilung entspricht, also $g = 3\sigma$.

Da laut Definition der Normalverteilung [2] 99,73 % aller Messwerte höchstens eine Abweichung von 3σ vom Mittelwert haben (die restlichen 0.27 % werden hier vernachlässigt; der Faktor 3 ist in diesem Sinne willkürlich gewählt), also sich in einem Intervall der Größe von $2 * 3\sigma + 1$ befinden, der um das *Mittelwert* herum gelegt ist, wird die

Varianz der Normalverteilung so gewählt, dass dieses Intervall deckungsgleich mit dem begrenzenden Intervall ist.

Normalverteilung:

$$P(x = m) : \frac{1}{\sigma\sqrt{2\pi}}$$

$$P(x > m) : \frac{1}{2} - \frac{1}{2\sigma\sqrt{2\pi}} = \frac{\sigma\sqrt{2\pi} - 1}{2\sigma\sqrt{2\pi}}$$

$$P(x < m) : \text{ist symmetrisch zu } >, \text{ also : } \frac{\sigma\sqrt{2\pi} - 1}{2\sigma\sqrt{2\pi}}$$

$$P(x >= m) : \frac{1}{2} + \frac{1}{2\sigma\sqrt{2\pi}} = \frac{\sigma\sqrt{2\pi} + 1}{2\sigma\sqrt{2\pi}}$$

$$P(x <= m) : \text{ist symmetrisch zu } >, \text{ also : } \frac{\sigma\sqrt{2\pi} + 1}{2\sigma\sqrt{2\pi}}$$

Da es auch hier fünf verschiedene Operatoren gibt, ist es wiederum nötig diese bei der Summierung jeweils mit $\frac{1}{5}$ zu multiplizieren. Somit ist die Gesamt-Chance, dass ein mit einfacher Gleichverteilung erzeugtes Guard erfüllt ist:

$$\frac{2}{10\sigma\sqrt{2\pi}} + 2 * \frac{\sigma\sqrt{2\pi} - 1}{10\sigma\sqrt{2\pi}} + 2 * \frac{\sigma\sqrt{2\pi} + 1}{10\sigma\sqrt{2\pi}} = \frac{2 + 4\sigma\sqrt{2\pi}}{10\sigma\sqrt{2\pi}} = \frac{2}{10\sigma\sqrt{2\pi}} + \frac{2}{5}$$

$$= \text{ Wahrscheinlichkeit, dass } (x = m) \text{ erfüllt ist } + \frac{2}{5}$$

Auf diese Weise wird die Wahrscheinlichkeit der Integer-Guards mit =-Operator wesentlich erhöht, und das soeben beschriebene Problem mit der Multiplikation der Wahrscheinlichkeiten der Guards einer Kante vermieden.

Um eine eingeschränkte Gleichverteilung oder eine Normalverteilung möglich zu machen, muss allerdings erst ein plausibler Wert der zu vergleichenden Variable bekannt sein, entweder als Mitte des Intervalls, oder als *Mittelwert* der Normalverteilung.

Es bedarf also einer Konstruktionsweise des Automaten, die die vorraussichtlichen Werte der Variablen und Uhren des Automaten in jedem einzelnen Knoten verfügbar macht.

Wir berechnen für jede Lokation Werte der Integer- und Uhrenvariablen, die für diese Lokation plausibel sind. Diese Werte wiederum benutzen wir zum Generieren 'in-

terressanter' Guards, d.h. Guards, für die sowohl Erfüllung als auch Nichterfüllung eine nicht vernachlässigbare Wahrscheinlichkeit haben. Das Ziel ist, Automaten zu generieren, für die schwer zu sagen ist, ob Erreichbarkeit des Fehlerzustandes gilt oder nicht.

Der wesentliche Unterschied zwischen Guards generiert mittels einer gleichverteilten Wahl innerhalb des Wertebereichs der in Frage kommenden Variable und einer Wahl mittels propagierten Werten und eingeschränkter Gleichverteilung oder Normalverteilung um den angenommenen Wert ist also letztendlich die voraussichtliche Wahrscheinlichkeit, dass ein Guard mit =-Operator erstellt wird, das erfüllbar ist, und die daraus resultierende Veränderung der Wahrscheinlichkeit des Produkts der Wahrscheinlichkeiten aller Guards einer Kante.

Hinzu kommt, dass Invarianten, wenn mittels Gleichverteilung erzeugt, aufgrund des unbegrenzten Wertebereichs der Uhren einem ähnlichen Problem unterliegen, da hier ohne Wertepropagierung nicht abgeschätzt werden kann, ob und wie eine Invariante erfüllbar ist, d.h. es wird unmöglich, zu entscheiden, ob Invarianten die den Knoten pauschal sperren, da sie nie erfüllbar sind, oder solche, die immer als erfüllbar gelten, da sie viel zu hohe Werte vermuten, erzeugt werden.

Da es mit unendlichem Wertebereich auch unendlich viele mögliche Invarianten gibt, ist der einfache Gleichverteilungs-Ansatz nicht brauchbar. Daher werden auch Invarianten mittels einer Normalverteilung in einem Intervall um einen bekannten plausiblen Wert herum erstellt, um so Invarianten zu erzeugen, die tatsächlich auf das Verhalten des Automaten Einfluss nehmen.

5 Verschiedene Lösungen

Die Struktur des fertigen Automaten kann auf mehrere Arten erzeugt werden, wie bereits in früheren Kapiteln erwähnt. Um jedoch gleichzeitig mit dem Erzeugen einer zusammenhängenden Struktur auch das Setzen „sinnvoller“, d.h. mit moderater Wahrscheinlichkeit erfüllbarer Invarianten und Guards zu erreichen, bedarf es eines Konstruktionsvorganges, der gleichzeitig überwacht, dass nur eine Zusammenhangskomponente entsteht, also immer irgendein Pfad zum Start existiert, und die Entwicklung der Werte der Variablen und Uhren des Automaten entlang der entstehenden Kanten derart überwacht, dass das nötige Wissen zum Erstellen von erfüllbaren Guards und Invarianten propagiert wird.

Es gibt zwei Arten dies zu bewirken.

5.1 Einfache Verkettung, mit Garantie einer einzigen Zusammenhangskomponente

In Anlehnung an die zufällige Erstellung von NFA [12] kann auf eine tieferliegende Strukturgenerierung verzichtet werden, und stattdessen können Knoten mit einer festgelegten Anzahl an ausgehenden Kanten erzeugt werden.

Es muss nur durch Rückfragen sichergestellt werden, dass alle so erstellten Knoten Teil der ersten Zusammenhangskomponente sind, und somit ein Pfad vom Start zu diesen Knoten besteht, über den sie die propagierten Werte „erben“ können.

Obwohl dies bereits Ergebnisse erzielt, sind die derart erstellten Automaten sehr einfach aufgebaut: Da jeder Knoten nur einmal erstellt wird, werden seine Invarianten und alle von ihm ausgehenden Kanten und deren Guards basierend auf den „geerbten“ Daten nach Stand zum Zeitpunkt des Erstellens dieses Knotens generiert, also basierend auf der Kante, die den ersten Pfad vom Start zu diesem Knoten abschließt.

Sollten später noch mehr in diesen Knoten eingehende Kanten bei der Konstruktion des Automaten hinzukommen, sind die ausgehenden Kanten des Knotens sehr wahrscheinlich zu diesen eingehenden Kanten nicht so kompatibel wie diese erste eingehende Kante.

Die Kanten sind also übermäßig linear verlegt, und es besteht keine Möglichkeit

Zyklen in diesem Automat gezielt zu generieren — etwas was bei RZA aufgrund der Uhren durchaus erwünscht sein kann (vergleiche Abb. 5.1).

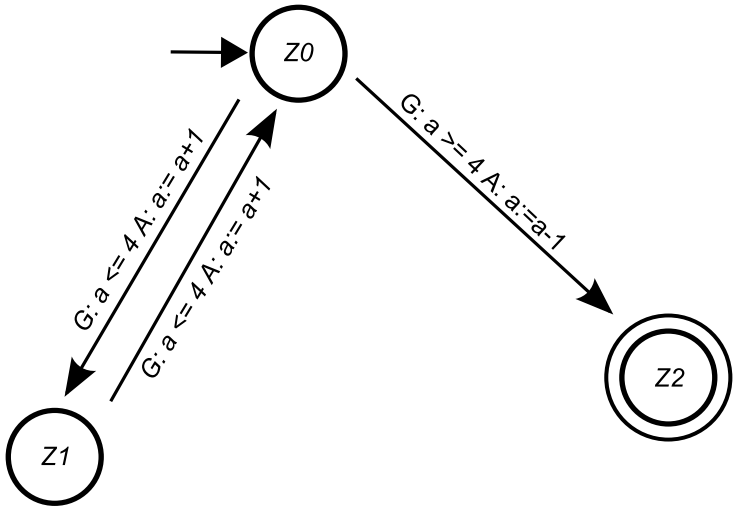


Abbildung 5.1: Der Zyklus zwischen S1 und S2 ist erwünscht.

Um dies zu umgehen, ist ein anderer Ansatz möglich:

5.2 Lange Verkettung, mit Garantie einer einzigen Zusammenhangskomponente

Statt die Knoten der Reihe nach abzulaufen und Kanten pauschal zu erstellen, ist es möglich, gezielt Pfade durch den Automaten zufällig zu verlegen, und anschließend durch Kanten zu realisieren, so dass die beteiligten Knoten ihre Werte entlang des Verlaufs des Pfades „vererben“. Dabei wird darauf geachtet, dass jeder Pfad in der ersten Zusammenhangskomponente beginnt, so dass alle Knoten die Teil eines solchen Pfades werden ebenfalls zur ersten Zusammenhangskomponente gehören, und es Werte gibt, die „vererbt“ werden können.

Dadurch dass diese Pfade eine höhere Lange als nur eine Kante haben, kann es dazu kommen, dass Pfade, entlang derer die Werte propagiert werden, überkreuzen. Sollten die Guards solcher Pfade an diesem Knoten vereinbar sein, kommt es zu einer Verzweigung. Auch kann es passieren, dass ein Knoten der bereits Teil eines oder mehrerer Pfade ist, Beginn eines Pfades wird, was erneut zu einer Verzweigung führt. Dadurch, dass ein Knoten so mehreren Pfaden angehören kann, kann es dazu kommen, dass die von ihm ausgehenden Kanten zu verschiedenen „vererbten“ Werten kompatibel sind.

Dies führt also zur Entstehung von Knoten, deren ausgehende Kanten eine größere Vielfalt an Werten haben, sodass es im entstehenden Automaten mehr unbeabsichtigte Pfade und Zyklen gibt.

5.3 Art der Kanten-Generierung

Die Idee der propagierten Werte basiert darauf, dass sich die Werte der Variablen und Uhren des Automaten entlang der gewählten Pfade im Automaten verändern.

Ist bekannt, welche Kanten seit Start gewählt wurden (oder zumindest was diese Kanten bewirkt haben), kann auch eine Aussage über die Werte der Variablen, und eine Einschränkung der möglichen Werte der Uhren gemacht werden.

Ziel ist es, basierend auf den aufgenommenen Werten neue Kanten mit Guards und Zuweisungen zu konstruieren, um so sowohl die Erfüllbarkeit der Guards zu erhöhen, als auch Zuweisungen zu erstellen, die die Variablen nicht aus deren zugelassenen Definitionsbereichen drängen, um so die Kante wiederum unzulässig zu machen.

Wir simulieren also einige mögliche Pfade im Automaten, um Aussagen über mögliche Werte der Variablen in einem Knoten machen zu können.

In Abbildung 5.2 wird ein solcher Vorgang dargestellt.

Die einzige Variable des Automaten, a, wird entlang der Kanten verändert. Jedesmal, wenn ein Konten das Ziel einer neu erstellten Kante wird, speichert er einen plausiblen Wert für a, basierend auf den Informationen des Ausgangsknoten der Kante und dem Einfluss der Kante.

Dabei kann es durchaus vorkommen, dass ein Knoten mehrere plausible Werte für a enthält. Diese kommen dann zum Einsatz, wenn der Knoten selbst Start eines Pfades zum Zwecke der Automatengenerierung werden sollte; in diesem Fall wird durch Zufall einer dieser soeben genannten plausiblen Wert gewählt, um den Pfad zu beginnen.

Im Beispiel in Abb.5.2 ist Knoten $Z2$ zweimal Teil eines Pfades mit $Z0$ als Ursprung gewesen; in beiden Fällen war $a = 0$ der Ausgangswert des Pfades. Sollte $Z2$ Ursprung eines Pfades werden, kann entweder $a = 8$ oder $a = 2$ als Ausgangswert gewählt werden.

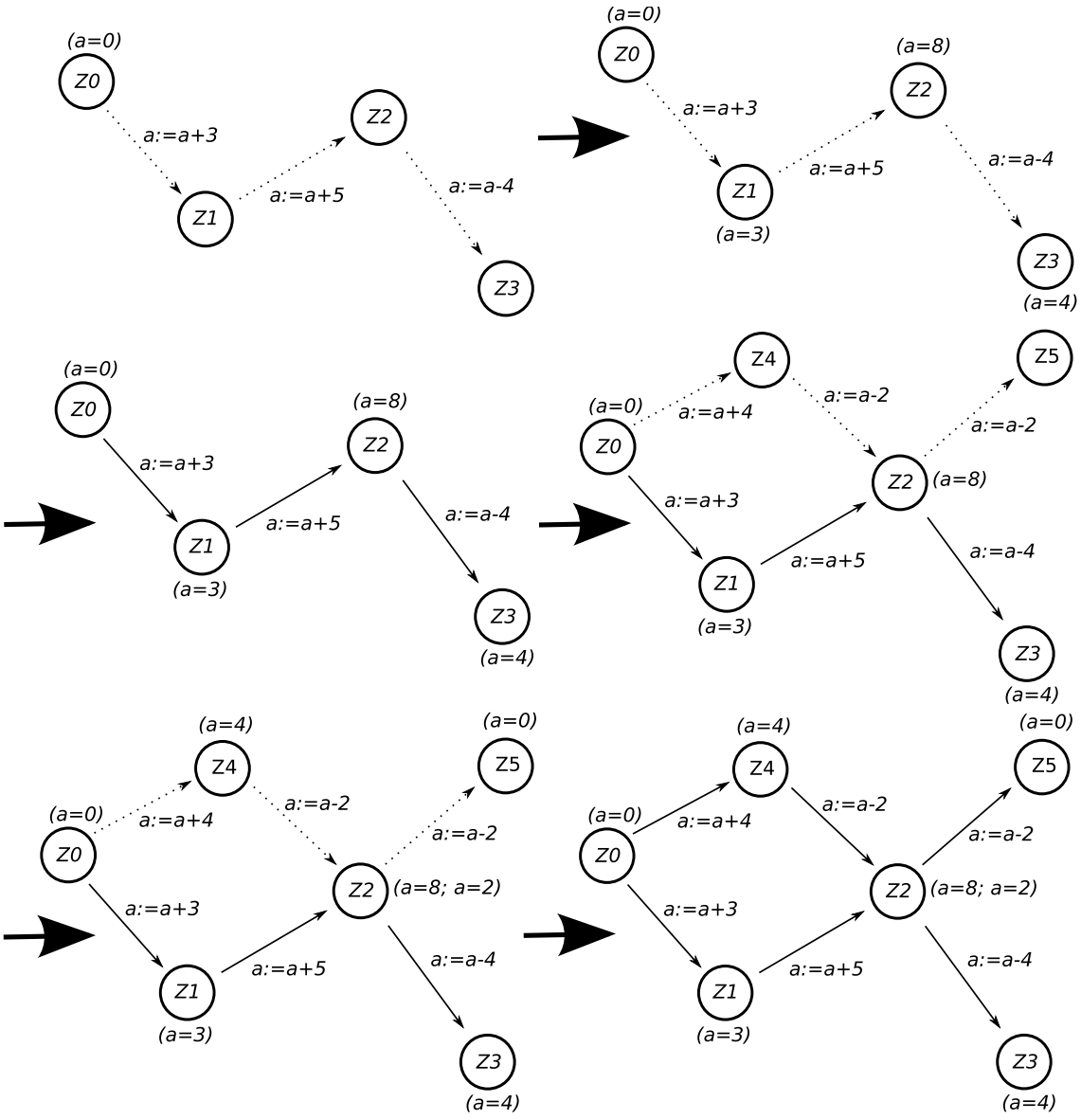


Abbildung 5.2: Ein kurzes Beispiel zur Kantenerstellung

6 Implementierung

6.1 Pattern-Generator

Ein Testmuster ist die Darstellung eines Systems aus mehreren Prozessen (vergleiche Algorithmus 2, Seite 41). Es werden erst die globalen Elemente (globale Variablen und Uhren) erzeugt, und dann an den Prozessgenerators weitergeleitet, der die gewünschte Anzahl an Prozessen generiert.

Außerdem wird auf dieser Ebene die Verteilung von Channels an die einzelnen Prozesse gehandhabt, um so eine gleichmäßige Verteilung der Ein- und Ausgabe-Symbole zu gewährleisten. Dabei hat jeder Prozess nur eine maximale Anzahl an möglichen Ein- oder Ausgabe-Symbolen, basierend auf der Gesamtzahl der Kanten die dieser Prozess besitzt, geteilt durch einen willkürlichen Faktor, um so das Überladen eines Prozesses mit Channels zu vermeiden.

Als Prozessgeneratoren kommen dann drei verschiedene Möglichkeiten in Frage:

6.2 Der Standard-Algorithmus, eine einfache Zufallsverkettung

Als Baseline-Vergleich wurde auch ein einfacher Zufalls-Generator verwendet, in dem alle Knoten der Reihe nach abgelaufen werden, um Ausgangsknoten für k Kanten zu werden (vergleiche Algorithmus 3, Seite 42). Die Ziel-Knoten der einzelnen Kanten werden per Gleichverteilung aus der Menge der Knoten gewählt. Hat ein Knoten bereits Ausgangswerte, einen Means-Bereich, für die Variablen, werden diese zur Kantenerstellung verwendet; hat er diese nicht, werden für jede Variable und Uhr per Gleichverteilung über den möglichen Wertebereich Ausgangswerte zugewiesen.

Im Falle der Uhren wird dabei ein Maximum von 10000 verwendet.

6.3 Erster Algorithmus, kurze Verkettung, mit Garantie einer einzigen Zusammenhangskomponente

Diese Variante beruht auf einem Prinzip ähnlich der Breitensuche (vergleiche Algorithmus 4, Seite 43): Es werden die Knoten der Reihe nach angelaufen, und ein eine Kante langer Pfad erzeugt.

Zu Beginn wird der Startknoten des Automaten (Knoten 1) erzeugt, und mit den festgelegten Startwerten der Variablen und Uhren belegt — dies ist der *Mittelwert-Bereich* des Startknotens, also eine Sammlung der plausiblen Mittelwerte aller Uhren und Variablen in diesem Knoten (vergleiche Abschnitt 4.3).

Dann werden von Knoten 1 aus k weitere Kanten zufällig erzeugt, deren Guards auf den Werten von Knoten 1 basieren.

Dabei wird beachtet, dass der in der Reihenfolge jeweilige nachfolgende Knoten, in diesem Fall Knoten 2, unter den Knoten ist, die auf diese Weise an die erste Zusammenhangskomponente angebunden werden.

Anschließend wechselt die Schleife auf Knoten 2 und erzeugt wiederum k Kanten an zufällige Knoten. Es wird überprüft, ob der Folgeknoten (Knoten 3) bereits durch generierte Kanten angebunden wurde; ist dies nicht der Fall, wird eine der k Kanten auf Knoten 3 gerichtet.

Danach wechselt die Schleife in Knoten 3, und fährt in der gleichen Weise fort, bis alle Elemente der Menge der Knoten abgelaufen sind (vergleiche Abb. 6.1).

Sobald die Kanten erstellt sind, besitzen die Zielknoten alle einen zusätzlichen Mittelwert-Bereich, mittels dem sie selbst ausgehende Kanten erstellen können.

Abschließend werden alle Knoten erneut durchlaufen, und eine festgelegte Prozentzahl von ihnen mit Invarianten versehen, wiederum basierend auf den propagierten Mittelwert-Bereichen.

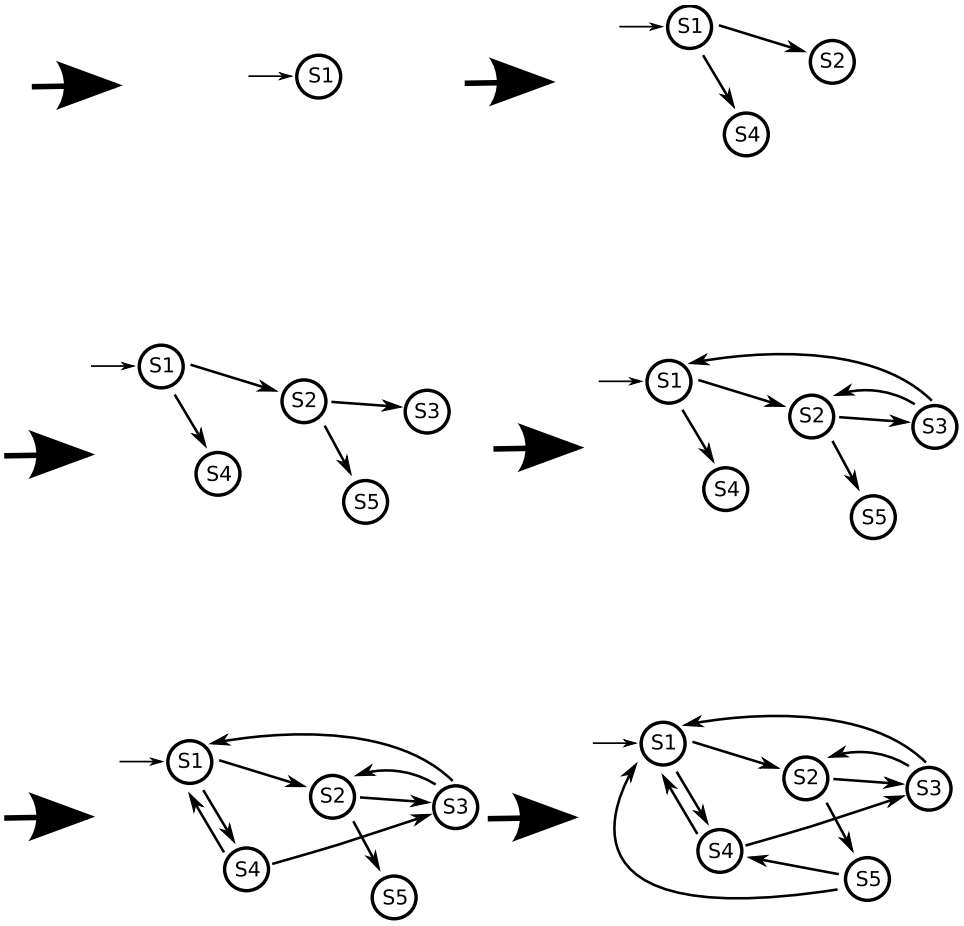


Abbildung 6.1: Ein kurzes Beispiel zur Breitenvariante, Knotenanzahl ist 5, k ist 2.

6.4 Zweiter Algorithmus, lange Verkettung, mit Garantie einer einzigen Zusammenhangskomponente

Diese Variante beruht auf einem Prinzip ähnlich der Tiefensuche (vergleiche Algorithmus 5, Seite 44):

Es werden die Knoten der Reihe nach angelaufen, und ein langer Pfad (also ein Pfad mit einer Länge größer als 1) erzeugt, entlang dessen die Werte propagiert werden; die Knoten, die an der Reihe sind, werden immer eines der beiden Enden eines Pfades.

Dabei soll immer mindestens ein Ende des Pfades ein bereits angeschlossener Knoten (ein Mitglied der ersten Zusammenhangskomponente) sein; dies wird dann der Anfang des Pfades, von dem aus basierend alle anderen Knoten des Pfades ihre Wertebereiche erhalten. Dabei gibt es zwei Fälle:

1. Fall: Ist der Knoten angeschlossen, wird von hier ein neuer, zufälliger Pfad der Länge k erstellt, indem per Gleichverteilung k Elemente aus der Menge der Knoten gewählt werden. Dann werden Kanten entlang dieses Pfades zwischen den Knoten erstellt und die Mittelwert-Bereiche von Knoten zu Knoten propagiert.

Anschließend gelten alle Knoten des Pfades als angeschlossen.

2.Fall: Ist der Knoten nicht angeschlossen, muss ein Ende gefunden werden, das angeschlossen ist. Es werden x zufällige Knoten an den Pfad angehängt, bis einer von ihnen angeschlossen ist; danach werden noch k Schritte wie in Fall 1 angehängt — hierbei wird darauf geachtet, dass der letzte Schritt der ist, der ursprünglich als Mitglied der ersten Zusammenhangskomponente gefunden wurde, um sicherzustellen, dass dieses Ende, an die erste Zusammenhangskomponente angeschlossen ist.

Danach wird die Liste umgedreht, so dass das an die erste Zusammenhangskomponente angeschlossene Ende zum Anfang des Pfades wird, und wie im 1. Fall behandelt, d.h. es werden Kanten erstellt und Mittelwert-Bereiche propagiert.

Der wesentliche Unterschied zum ersten Algorithmus ist hier, dass Knoten mehrmals Anfang eines Pfades werden können, und zu diesem Zeitpunkt mehrere Mittelwert-Bereiche besitzen können, also mehrere plausible Mittelwerte für die Variablen und Uhren des Systems, je nachdem, über welchen Pfad der Knoten erreicht wurde — aus diesen Bereichen wird dann per Gleichverteilung gewählt.

Dies ermöglicht das Entstehen von Zyklen, und erhöht die Chance, dass verzweigte Pfade durch Teile von sich kreuzenden Pfaden erzeugt werden, deren Guards zueinander passen.

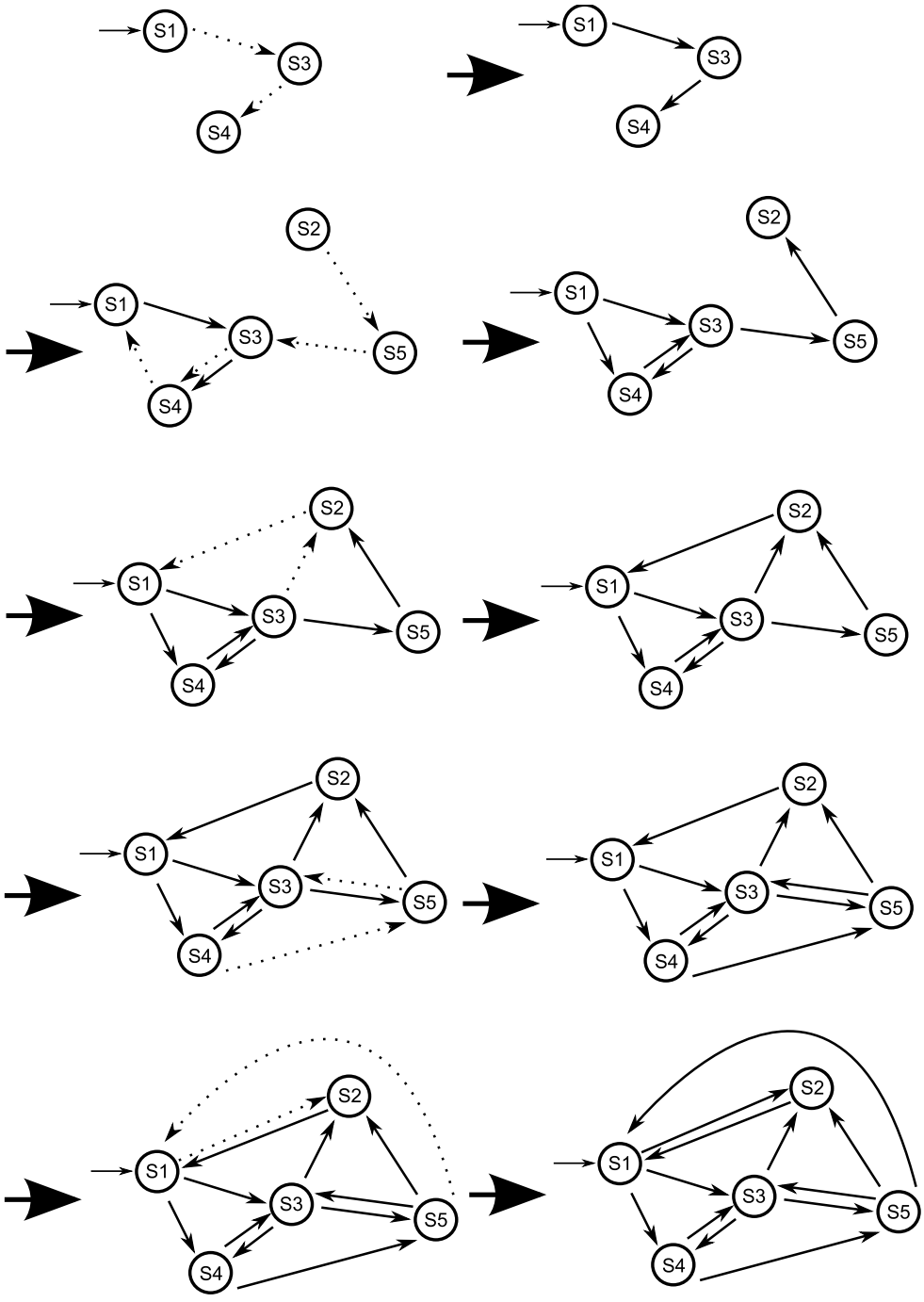


Abbildung 6.2: Ein kurzes Beispiel zur Tiefenvariante, Knotenanzahl ist 5, k ist 2.

6.5 Der Kantengenerator

Der Kantengenerator basiert darauf, dass gegeben Anfangs- und Endknoten der Kante, mit Hilfe der übergebenen Mittelwert-Bereiche die gewünschte Menge plausibler Guards erzeugt werden kann (vergleiche Algorithmus 7, Seite 46). Anschließend werden die Werte-Zuweisungen dieser Kante per Zufall entschieden: Die Veränderung der Werte der betreffenden Variable v, global oder lokal, werden zufällig gewählt; sie entsprechen entweder der Form $v = v \pm w$, oder der Form $v := m$, mit $w \in [v - v * r, v + v * r]$ und m innerhalb des definierten Wertebereich von v. Dabei ist r ein eine reelle Zahl mit $r > 0$; dieser Faktor bestimmt, wie schnell sich die Integer-Variablen des Systems ändern können.

Auch wird entschieden, ob in dieser Kante ein Reset einer oder mehrerer der Uhren durchgeführt wird; dies wird als zusätzliche Zuweisungen der Kante notiert.

Zuletzt wird entschieden, wie viel Zeit in den einzelnen Uhren vergeht; Uppaal simuliert das Vergehen von Zeit nur in den Location, nicht entlang der Kanten, deshalb ist das erzeugte Zeitintervall immer als zum Ursprungsknoten der Kante gehörend zu betrachten, wird aber mit der Kante erzeugt.

6.6 Guard-Generator und Invarianten-Generator

Die Erzeugung von Guards und Invarianten läuft ähnlich ab, der wesentliche Unterschied ist, dass Invarianten sich in Uppaal nur auf Uhren beziehen können, und der Operator immer entweder $\leq$ oder $<$ sein muss (vergleiche Algorithmus 8 und Algorithmus 9, Seite 47).

Es wird ein Operator gewählt und anschließend ein Vergleichswert basierend auf dem übergebenen plausiblen Mittelwert erzeugt.

Wie bereits in Abschnitt 4.3 beschrieben werden die Werte, gegen die der Vergleichsoperator gerichtet ist, mittels Normalverteilung gewählt; es wird ein Intervall $[a, b]$ um den Mittelwert von v erzeugt, und anschließend die Varianz σ der Normalverteilung so gewählt, dass $6 * \sigma = |b - a|$ gilt, da laut Definition der Normalverteilung $99,73\%$ aller Messwerte eine Abweichung von höchstens 3σ vom Mittelwert haben. Die restlichen 0.27% werden mittels einer Schleife abgefangen.

Um a und b zu wählen, wird ähnlich wie in der Werte-Zuweisungen der Kanten vorgegangen: Dabei ist die Breite des Intervalls durch einen vorgegebenen Multiplikator, multipliziert mit dem Mittelwert, bestimmt, also $a = v - v * p, b = v + v * p$. Je größer dieser Multiplikator gewählt wird, desto mehr wächst die Größe des resultierende Intervall, und damit die Varianz der Normalverteilung. p ist ein willkürlicher Faktor $0 \leq p \leq 1$. Sollte

p größer als 1 gewählt werden, würde das resultierende Integer-Intervall in den negativen Bereich hineinreichen, was nicht zweckdienlich ist; auch ist ein Wert kleiner als 0.1 nicht sinnvoll, da das resultierende Integer-Intervall auf zu kleine Bereiche beschränkt wird, und im schlimmsten Falle nur 3 Elemente enthält. Diese zum Mittelwert relative Intervall-Größe wurde gewählt, um Intervalle relativ zur Veränderlichkeit des Wertes der Variablen zu erhalten.

Algorithm 2 Test-Pattern Generator

Require: NrA=Anzahl Automaten, NrCh=Anzahl Channels, NrVar=Anzahl Variablen, NrGVar=Anzahl globaler Variablen, NrCl=Anzahl Uhren, NrLoc=Anzahl Locations pro Automat

1: **for** k ← 1 to NrCl **do**
2: ListofClocks[k] ← New_Clock
3: **end for**
4: **for** k ← 1 to NrGVar **do**
5: ListofGlobals[k] ← New_Global_Variable
6: **end for**
7: **for** j ← 1 to NrA **do**
8: Process[j] ← PROZESS-GENERATOR(NrVar, ListofClocks, ListofGlobals, NrLoc)
9: **end for**
10: **for all** ch ← 1 to chNr **do**
11: **for all** $Pr \in Process[]$ **do**
12: $Slots[Pr] \leftarrow (NumberofLocations(Pr) * MaximalerAusgangsgrad/l)$ ▷ l ist Konstante, vom User festgelegt
13: **end for**
14: **while** $a, b \in Processes; a \neq b; Slot[a] > 0; Slot[b] > 0$ **do** ▷ suche ein Paar Prozesse, die beide noch genügend Platz für einen weiteren Channel haben.
15: ch ← random_Channel
16: loc_a ← random_location ∈ Locations(a)
17: loc_b ← random_location ∈ Locations(b)
18: edge_a ← random_edge ∈ loc_a
19: edge_b ← random_edge ∈ loc_b
20: edge_a+channel?(ch)
21: edge_b+channel!(ch)
22: $Slot[a]--; Slot[b]--$
23: **end while**
24: **end for**
25: **return** Pattern

Algorithm 3 Process-Generator, Variante 0, Baseline

Require: NrVar=Anzahl Variablen des Prozesses, ListofCl=Liste der Clocks des Systems, ListofGlobals=Liste der globalen Variablen des Systems, NrLoc=Anzahl Locations des Prozesses

```
1: function PROCESS-GENERATOR(NrVar, ListofCl, ListofGlobals, NrLoc)
2:     for k ← 1 to NrVar do
3:         ListofVar[k] ← New_Variable
4:     end for
5:     Wertebereich1 ← Liste an Tupeln (x,0), x ∈ ListofVar+ListofCl+ListofGlobals
6:     location[1...NrLoc] ← empty
7:     location[1].append(New_Location(1, Wertebereich1))      ▷ Knoten 1 wird bereit gemacht
8:     for j ← 1 to NrLoc do
9:         for k ← 1, Ausgrad do                ▷ Ausgrad ist Konstante, siehe maximaler Ausgangsgrad
10:            if location[j]==empty then
11:                alte_werte ← Liste an Tupeln (x,r_x), x ∈ ListofVar+ListofCl+ListofGlobals, r_x zufälliger Wert innerhalb der gültigen Werte von x
12:            else
13:                alte werte ← location[k].random_Wertebereich
14:            end if
15:            Ziel ← randominteger(1,NrLoc)
16:            neue Werte, neue kante ← KANTEN-GENERATOR(ch1, ch2, alte werte, ListofVar+ListofGlobals, ListofClocks)
17:            location[ch1] add neue kante
18:            if location[Ziel]==empty then
19:                location[Ziel] ← New_Location(Ziel, neue werte)
20:            else
21:                location[Ziel] add Wertebereich(neue Werte)
22:            end if
23:        end for
24:    end for
25:    for j ← 1 to NrLoc do
26:        if randomint(1, 100) <= h  then ▷ h ist Konstante, bestimmt die Häufigkeit von Invarianten
27:            rd_cl ← random(clock) ∈ clocks
28:            loc ← location[j]
29:            location[rd_loc] ← INVARIANTENGENERATOR(rd_cl, loc)
30:        end if
31:    end for
32:    return Processobjekt(ListofVar, ListofCl, location)
33: end function
```

Algorithm 4 Process-Generator, Variante 1

Require: NrVar=Anzahl Variablen des Prozesses, ListofCl=Liste der Clocks des Systems, ListofGlobals=Liste der globalen Variablen des Systems, NrLoc=Anzahl Locations des Prozesses

```
1: function PROCESS-GENERATOR(NrVar, ListofCl, ListofGlobals, NrLoc)
2:     for k ← 1 to NrVar do
3:         ListofVar[k] ← New_Variable
4:     end for
5:     Wertebereich1 ← Liste an Tupeln (x,0), x ∈ ListofVar+ListofCl+ListofGlobals
6:     location[1...NrLoc] ← empty
7:     location[1].append(New_Location(1, Wertebereich1))      ▷ Knoten 1 wird bereit gemacht
8:     for j ← 1 to NrLoc do
9:         for k ← 1, Ausgrad do              ▷ Ausgrad ist Konstante, siehe maximaler Ausgangsgrad
10:            if location[j+1]==empty then
11:                Ziel ← k+1
12:            else
13:                Ziel ← randominteger(1,NrLoc)
14:            end if
15:            alte werte ← location[ch1].random_Wertebereich
16:            neue Werte, neue kante ← KANTEN-GENERATOR(ch1, ch2, alte werte, ListofVar+ListofGlobals, ListofClocks)
17:            location[ch1] add neue kante
18:            if location[Ziel]==empty then
19:                location[Ziel] ← New_Location(Ziel, neue werte)
20:            else
21:                location[Ziel] add Wertebereich(neue Werte)
22:            end if
23:        end for
24:    end for
25:    for j ← 1 to NrLoc do
26:        if randomint(1, 100) <= h  then ▷ h ist Konstante, bestimmt die Häufigkeit von Invarianten
27:            rd_cl ← random(clock) ∈ clocks
28:            loc ← location[j]
29:            location[rd_loc] ← INVARIANTENGENERATOR(rd_cl, loc)
30:        end if
31:    end for
32:    return Processobjekt(ListofVar, ListofCl, location)
33: end function
```

Algorithm 5 Process-Generator, Variante 2, Teil 1

Require: NrVar=Anzahl Variablen des Prozesses, ListofCl=Liste der Clocks des Systems, ListofGlobals=Liste der globalen Variablen des Systems, NrLoc=Anzahl Locations des Prozesses

```
1: function PROCESS-GENERATOR(NrVar, ListofCl, ListofGlobals, NrLoc)
2:     for k ← 1 to NrVar do
3:         ListofVar[k] ← New_Variable
4:     end for
5:     Wertebereich1 ← Liste an Tupeln (x,0), x ∈ ListofVar+ListofCl+ListofGlobals
6:     location[1...NrLoc] ← empty
7:     location[1].append(New_Location(1, Wertebereich1))      ▷ Knoten 1 wird bereit gemacht
8:     for j ← 1 to NrLoc do
9:         chain ← emptylist
10:        chain[1] ← j
11:        if location[j] != empty then
12:            for h ← 2 to Ausgrad do          ▷ Ausgrad ist eine vom User vorgegebene Konstante, siehe maximaler Ausgangsgrad
13:                chain[h] ← randominteger(1,NrLoc)
14:            end for
15:        else
16:            repeat
17:                r ← randominteger(1,NrLoc)
18:                chain.append(r)
19:            until location[r] != empty
20:            chain.reverse()
21:            chain2 ← emptylist
22:            chain2.append(chain[1])
23:            for h ← 1 to (Ausgrad-1) do
24:                chain2[h] ← randominteger(1,NrLoc)
25:            end for
26:            chain = chain2.append(chain)
27:        end if
28:        for all ch1,ch2 ∈ chain do
29:            alte werte ← location[ch1].random_Wertebereich
30:            neue werte, kante ← KANTEN-GENERATOR(ch1, ch2, alte werte, ListofVar+ListofGlobals, ListofClocks)
31:            location[ch1] add kante
32:            if location[ch2]==empty then
33:                location[ch2] ← New_Location(ch2, neue werte)
34:            else
35:                location[ch2] add Wertebereich(neue Werte)
36:            end if
37:        end for
38:    end for
```

Algorithm 6 Process-Generator, Variante 2, Teil 2

```
39:     for j ← 1 to NrLoc do
40:         if randomint(1, 100) <= h  then ▷ h ist Konstante, bestimmt die Häufigkeit
    von Invarianten
41:             rd_cl ← random(clock) ∈ clocks
42:             loc ← location[j]
43:             location[rd_loc] ← InvariantenGenerator(rd_cl, loc)
44:         end if
45:     end for
46:     return Processobjekt(ListofVar, ListofCl, location)
47: end function
```

Algorithm 7 Kanten-Generator

Require: StartKnotenNummer, EndknotenNummer, alte Mittelwerte, Liste der Lokalen und Globalen Variablen, Liste der Uhren

```
1: function KANTEN-GENERATOR(StartKnoten, Endknoten, alteWerte, ListofVarand-
   Globals, ListofClocks)
2:     neueWerte=alteWerte
3:     guards ← empty
4:     for h ← 1, random(1, i) do                ▷ i ist Konstante, siehe Abschnitt 7.1
5:         random(ListofVarandGlobals+ListofClocks) → v
6:         newGuard ← GUARDGENERATOR(v, alteWerte[v])
7:         guards.append(newGuard)
8:     end for
9:     assigns ← empty
10:    assigned[] ← empty
11:    for h ← 1, random(1, Anzahl(ListofVarandGlobals + ListofClocks) do
12:        random(ListofVarandGlobals+ListofClocks) → v
13:        assigned.append(v)
14:        g ← randomint(1,100)
15:        if g ≤Häufigk'rnd' then
                                  ▷ Häufigk'rnd' ist Konstante, siehe Abschnitte 8 und 9.2
16:            randomwert = random(Variable.Minimum, Variable.Maximum)
17:            newassign ← assign(v, == , randomwert)
18:            assigns.append(newassign)
19:            neueWerte(v)=randomwert
20:        else if Häufigk'rnd'≤ g ≤Häufigk'rnd'+Häufigk'+' then
                                  ▷ Häufigk'+' ist Konstante, siehe Abschnitte 8 und 9.2
21:            wert ← alteWerte(v)
22:            repeat
23:                delta= randominteger(1, 1+wert*X%)        ▷ X ist Konstante, siehe
   Abschnitte 8 und 9.2
24:            until wert+delta ≤ Variable.Maximum
25:            newassign ← assign(v, '+' , delta)
26:            assigns.append(newassign)
27:            neueWerte(v)=wert+delta
28:        else if Häufigk'rnd'+Häufigk'+'≤ g ≤Häufigk'rnd'+Häufigk'+'+Häufigk'-'
   then
                                  ▷ Häufigk'-' ist Konstante, siehe Abschnitte 8 und 9.2
29:            wert ← alteWerte(v)
30:            repeat
31:                delta=randominteger(1, 1+wert*X%)
32:            until wert-delta ≥ Variable.Minimum
33:            newassign ← assign(v, '-' , delta)
34:            assigns.append(newassign)
35:            neueWerte(v)=wert-delta
36:        end if
37:    end for
38:    Zeitdelta=random(1,50)
39:    for all cl ∈ ListofClocks do
40:        neueWerte[cl]=alteWerte[cl]+Zeitdelta
41:    end for
42:    return neueWerte, Kante(StartKnoten, Endknoten, guards, assigns)
43: end function
```

Algorithm 8 GuardGenerator

Require: Variable, alterWert

```
1: function GuardGenerator(Variable, alterWert)
2:     a ← k * Zeitwert                          ▷ k Konstante, siehe Abschnitt 8
3:     b ← k * Zeitwert                          ▷ k Konstante, siehe Abschnitt 8
4:     x ← max(a,Variable.Minimum)
5:     y ← min(b,Variable.Maximum)
6:     value ← restrictedGauss(x,y)
7:     comparison ← choose from (<, >, ≤, ≥, ==)
8:     return guardobject(v, comparison ,value)
9: end function
```

Algorithm 9 InvariantenGenerator

Require: Clock, Location

```
1: function InvariantenGenerator(Clock, location)
2:     bereich ← randomchoice(location.wertebereiche)
3:     Zeitwert ← bereich[Clock]
4:     a ← k * Zeitwert                          ▷ k Konstante, siehe Abschnitt 8
5:     b ← k * Zeitwert                          ▷ k Konstante, siehe Abschnitt 8
6:     x ← max(a,Variable.Minimum)
7:     y ← min(b,Variable.Maximum)
8:     value ← restrictedGauss(x,y)
9:     comparison ← choose from (<, ≤)
10:    return Invariant(Clock, comparison ,value)
11: end function
```

Algorithm 10 restrictedGauss

Require: a, b = Zahlen aus $\mathbb{R}$

```
1: function restrictedGauss(a, b)
2:     varianz = |a−b|/6                         ▷ siehe Abschnitt 4.3
3:     Mean=a + |a−b|/2
4:     while True do
5:         Gaussergebnis=Normalverteilung(Mean,Varianz)
6:         if a ≤ Gaussergebnis ≤ b then
7:             break
8:         end if
9:     end while
10:    return Gaussergebnis
11: end function
```

7 Mathematische Eigenschaften

7.1 Die Bedeutung des Verhältnis *Ausgangsgrad* zu *maximale Anzahl Guards pro Kante*

Da jeder Knoten eine Menge von Ausgangskanten hat, die durch den *Ausgangsgrad (A)* angegeben wird, ist allein durch die in diesem System maximal erlaubte Anzahl an Guards pro Kante bestimmt, wie wahrscheinlich es letztendlich ist, dass ein Knoten wieder verlassen werden kann.

Die Anzahl der Guards pro Kante werden gleichverteilt bei der Erstellung einer Kante gewählt, es gibt also theoretisch bei beliebig vielen Kanten genausoviele Kanten mit dem Minimum an Guards wie es solche mit dem Maximum gibt, und alle Werte dazwischen.

Somit ist es bei höheren Ausgangsgraden eher möglich zu versichern, dass eine „einfache“ Kante mit minimalen Guards aus dem Knoten heraus existiert.

Des weiteren erhöht sich die Anzahl an Möglichkeiten insgesamt, so dass mit steigendem Ausgangsgrad es immer wahrscheinlicher wird, dass ein Knoten verlassen werden kann.

Zudem bewirkt eine höherer Ausgangsgrad einen dichteren Automaten und somit eine höhere Anzahl an Pfaden durch den Automaten, so dass das Erreichen der Fehlerzustandes eher möglich wird.

Im Gegensatz dazu bewirkt ein Erhöhen der Maximalzahl der Guards das Gegenteil: da die Guards einer Kante alle erfüllt sein müssen, wird eine einzelne Kante immer unwahrscheinlicher, je mehr Guards sie trägt.

Es ist daher von Interesse zu wissen, wie hoch der Ausgangsgrad eines Knoten in einem der erstellten Automaten ist.

7.1.1 Ausgangsgrad des Baseline-Algorithmus

Beim Zufalls-Ansatz ist der Ausgangsgrad der Knoten fest vorgegeben: *(A)*.

7.1.2 Ausgangsgrad des ersten Algorithmus

Beim ersten Ansatz ist der Ausgangsgrad der Knoten fest vorgegeben: *(A)*.

7.1.3 Statistischer Ausgangsgrad des zweiten Algorithmus

Im zweiten Algorithmus ist der Ausgangsgrad der Knoten nicht festgelegt, sondern ergibt sich letztendlich durch die Gleichverteilung der Zielknoten der verlegten Kanten, und damit Pfade, aus der Gesamtzahl an Kanten K_{max}, geteilt durch die Anzahl der Knoten N, also:

$$(\frac{K_{max}}{N})$$

Es ist also nur noch nötig, diese Gesamtzahl der Kanten abzuschätzen (vergleiche [12]).

Per Definition ist der Startknoten s_0 immer der erste Knoten, der bearbeitet wird (vergleiche Algorithmus 5, Seite 44). Deshalb ist er der einzige Knoten eines Automaten/Prozesses, für den schon beim Start plausible Werte existieren, nämlich die Startwerte der Variablen bei Beginn des Systems.

Dies bewirkt die Erzeugung eines Pfades der Länge A, also von A neuen Kanten, was maximal A neue Knoten an die erste Zusammenhangskomponente anschließt.

Zwei Fälle können nun für den nächsten Knoten s_{i+1} in der Reihenfolge bestehen:

1. Der Knoten s_{i+1} ist angeschlossen und generiert einen Pfad der Länge A; dabei produziert er A viele Kanten, die maximal A viele Knoten anschließen können.

2. Der Knoten s_{i+1} ist nicht angeschlossen, produziert also auf seiner Suche nach Anschluß einen „suchenden“ Pfad der Länge x, also x viele Kanten, findet dann Anschluß und verlängert den Pfad um A, also zusätzliche A viele Kanten.
   Dabei sind maximal $x + A$ viele Knoten angeschlossen worden.

   Dieser Vorgang wiederholt sich, bis alle Knoten durchlaufen sind.

Es ergibt sich also eine Mindest-Gesamtzahl von Kanten von $(A * N)$, was, geteilt durch die Anzahl der Knoten, einem Ausgangsgrad von A entspricht. Verändert wird dieser Wert durch die Anzahl der Kanten, die durch die im zweiten Fall erzeugten x Kanten bei jedem Schritt hinzukommen können.

Im Minimalfall ist x immer 1, d.h. es wird eine Kante während der Suche erzeugt, und genau ein neuer Knoten angeschlossen, bevor der Pfad um A verlängert wird (vergleiche Abb. 7.2).

Im Maximalfall kann x unendlich groß werden, indem unendlich oft Knoten die nicht zur ersten Zusammenhangskomponente gehören gewählt werden, was allerdings

sehr unwahrscheinlich ist. Dies kann mittels der geometrischen Verteilung [2] abgeschätzt werden:

Die geometrische Verteilung entspricht einem Problem der Art „Warten auf den ersten Treffer“ in einem Bernoulli-Experiment [2], die dazugehörige Definition ist:

$$p_k = p * (1-p)^{k-1}$$

für k=1,2,3... Dies gibt die Wahrscheinlichkeit dafür an, dass bei einem Bernoulli-Experiment das Ereignis A mit $p = P(X)$ zum ersten Mal beim k-ten Versuch eintritt.

Das Ereignis X ist die Wahl eines Knoten aus der ersten Zusammenhangskomponente. Sei K_{z1} die Anzahl Knoten in der ersten Zusammenhangskomponente. Also sei $p = \frac{K_{z1}}{N}$, d.h. p steigt, je mehr Knoten an die erste Zusammenhangskomponente angeschlossen werden.

Die Wahrscheinlichkeit, dass nach M Versuchen das Ereignis X eintritt, ist daher die Summe:

$$P(M) = \sum_{k=1}^{M} p * (1-p)^{k-1}$$

Der zu dieser Verteilung gehörende Erwartungswert für M ist:

$$E(M) = \frac{1}{p}$$

In schlimmsten Falle ist $p = \frac{1}{N}$, was selten genug ist, da dazu der Startknoten k-mal auf sich selbst verwiesen haben muss, um nicht zusätzliche Knoten anzuschließen. Dies ist möglich, wenn für den Pfad der Länge A den der Startknoten erzeugt, jedesmal s_0 als Ziel der einzelnen Kanten des Pfades gewählt wird.

In diesem Fall gilt:

$$E(M) = \frac{1}{\frac{1}{N}} = N$$

In diesem Extremfall muss man im Mittel also N mal einen Knoten zufällig auswählen, um den Startknoten zu wählen; da p aber in der Regel größer ist, da im ersten Schritt mehr Knoten an die erste Zusammenhangskomponente angeschlossen werden, verringert sich dieses Mittel entsprechend. Da diese Verteilung asymptotisch gegen Null konvergiert, ist $1 \leq x \leq N$ am wahrscheinlichsten.

Die Mindest-Gesamtzahl der Kanten wird durch x also derart verändert, dass im Mittel für jeden Knoten zwischen 0 und N viele Kanten zusätzlich erstellt werden.

Dabei führt das Erzeugen eines langen „suchenden“ Pfades mit großem x dazu, dass

bei diesem Schritt maximal $x + A$ viele Knoten an die erste Zusammenhangskomponente angeschlossen werden, was die Menge der Knoten, die nicht angeschlossen sind, die also noch eine Suche auslösen und nicht beenden entsprechend stark verringert; d.h. p steigt anschließend stark an.

Daraufhin können daher nur im Mittel noch kleine „suchenden“ Pfade mit geringem x gebildet werden, falls überhaupt, da sich der Erwartungswert entsprechend verändert (vergleiche Abb. 7.1).

Zur Beurteilung des daraus resultierenden Ausgangsgrades hilft es, die Extremfälle zu betrachten: Sollte nur der Minimalfall wiederholt eintreten, wird durch jeden solchen Knoten ein Pfad der Gesamtlänge von l, $A \leq l \leq A + 1$ erzeugt, dementsprechend ist der Ausgangsgrad der Knoten in diesem Falle zwischen A und A+1.

Sollte ein Maximalfall eintreten, wird durch diesen Knoten ein Pfad der Gesamtlänge von im Mittel $l \leq A + N$ erzeugt, dies schließt allerdings das Erzeugen von weiteren „suchenden“ Pfaden weitestgehend aus.In diesem Extremfall werden also $(N - 1)$-mal A viele Kanten erzeugt, und einmal im Mittel maximal $N + A$ viele Kanten, was, geteilt durch die Anzahl der Knoten, wieder einem Ausgangsgrad zwischen A und A+1 entspricht.

Es kann also o.B.d.A. geschlossen werden, dass im Mittel der Ausgangsgrad eines Automaten generiert durch Variante 2 zwischen A und A+1 liegt.

1. 1←5←6

2. 2←3←4←5←6←1

➤ 1←6←5←4←3←2

3. 3←6←1

4. 4←2←1

5. 5←2←6

6. 6←3←5

= 15 Kanten

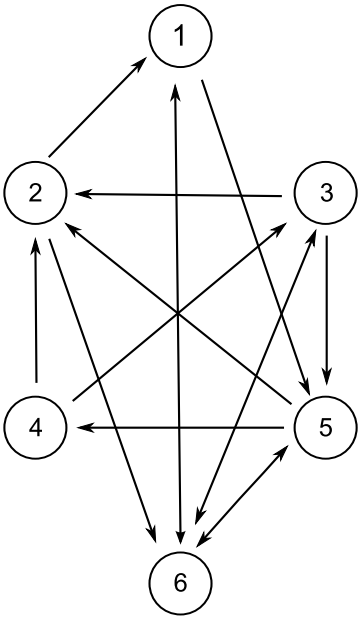


Abbildung 7.1: Ein Beispiel mit einer langen Suche (in Schritt 2) (vergleiche Algorithmus 5), und anschließend keiner zusätzlichen Suche. Der Automat hat die Größe 6 und $A = 2$.

1. 1←5←6

2. 2←1←5←6

3. 3←2←6←1

4. 4←2←3←3

5. 5←4←5

6. 6←4←4

= 15 Kanten

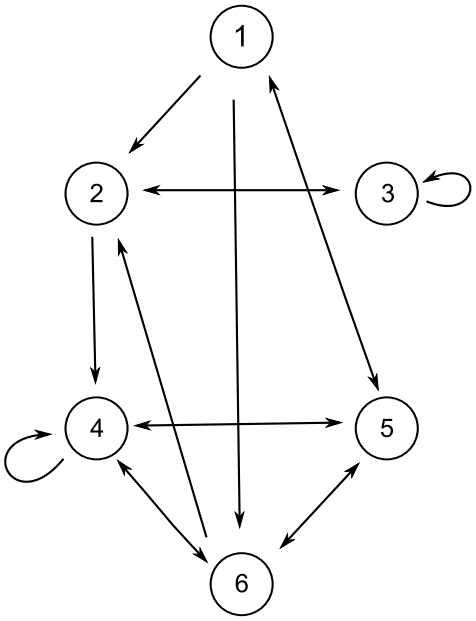


Abbildung 7.2: Ein Beispiel mit vorwiegend kurzen Suchen (vergleiche Algorithmus 5). Der Automat hat die Größe 6 und $A = 2$.

8 Experimente

Zu der hier verwendeten Terminologie:

Ein einzelner *Test* generiert ein einziges System aus den vom Generator erzeugten Dateien: der Musterdatei, die die Beschreibung des zufällig generierten Systems enthält, und der dazugehörigen Aufgaben-Datei (auch Query-Datei), die den zufällig generierten Fehler beschreibt, mit der Anweisung, dass dieser nicht erreicht werden darf.

„verifyta“, die Kommandozeilen-Version von Uppaal, liest diese Dateien ein und gibt entweder „satisfied“ oder „not satisfied“ als Ausgabe zurück, je nachdem ob der Fehler vermieden wurde oder nicht.

Um also die Verteilung von „satisfied“ und „not satisfied“ testen zu können, müssen ganze *Testreihen* gemacht werden; dabei werden immer wieder zufällige Muster und Queries basierend auf in dieser Reihe identischen Eingaben in den Generator erzeugt, die Dateien von verifyta getestet und die Anzahlen an „satisfied“ und „not satisfied“-Ausgaben festgehalten. Eine Testreihe ist also eine Sammlung an Tests mit identischen Eingaben in den Generator.

Eine *Serie* ist eine Abfolge an Testreihen, in der die Eingabe in den Generator systematisch variiert wird, um die Auswirkungen dieser Veränderung auf die Anzahlen von „satisfied“ und „not satisfied“-Ausgaben der so entstehenden Testreihen miteinander vergleichen zu können, und damit Rückschlüsse auf die generelle Auswirkung von auf diese Weise veränderten Eingaben machen zu können.

Anschließend wurde, um die Ergebnisse zu normalisieren, zu jeder Reihe die Anzahlen der „satisfied“-Ausgaben und „not satisfied“-Ausgaben jeweils aufsummiert, und danach der Quotient $\frac{satisfied}{not\_satisfied}$ gebildet, um die Verhältnisse unabhängig von der Gesamtzahl an Tests in dieser Reihe vergleichen zu können. Der sich daraus ergebende Quotient $0 \leq Q$ ist ein Maß dafür, wie gleichmäßig das Verhältnis der Anzahlen der „satisfied“-Ausgaben und „not satisfied“-Ausgaben ist; je näher Q an 1 ist, desto gleichmäßiger. Um dabei eine Division durch Null zu umgehen, wird in dem Fall, dass die Anzahl von „not satisfied“-Ausgaben gleich Null ist, diese auf 1 erhöht.

Alle Testreihen mit Quotienten über 0.8 und unter 1.25 wurden als erfolgreiche Testreihen angenommen, dies entspricht in einer Verteilung von hundert Fällen in etwa

je einem Verhältnis 45/55 oder 55/45 (vergleiche Abschnitt 4.3).

8.1 Testserie 1 — Auswirkung des Verhältnisses „*Ausgangsgrad* zu *maximale Anzahl Guards pro Kante*“ auf die Verhältnisse zwischen den Anzahlen der „satisfied“-Ausgaben und „not satisfied“-Ausgaben

Es wurde von zwei Einstellungen für die Generierung eines Systems ausgegangen:
Einstellung 1 — Testreihe 1:

- durchschnittliche Anzahl Knoten pro Prozess: 10
- Anzahl Prozesse im System: 2
- Anzahl Uhren im System: 1
- Anzahl lokale Variablen pro Prozess: 3
- Anzahl globale Variablen im System: 1
- Anzahl Channels im System: 1
- Ausgangsgrad der Knoten: variabel, siehe unten
- Maximal Anzahl Guards pro Kante: variabel, siehe unten

Einstellung 2 — Testreihe 2:

- durchschnittliche Anzahl Knoten pro Prozess: 10
- Anzahl Prozesse im System: 2
- Anzahl Uhren im System: 1
- Anzahl lokale Variablen pro Prozess: 3
- Anzahl globale Variablen im System: 1
- Anzahl Channels im System: 1
- Ausgangsgrad der Knoten: variabel, siehe unten
- Maximale Anzahl Guards pro Kante: variabel, siehe unten

Dies wurde getan, um Unterschiede beim Generieren eines Systems ohne Kommunikation mit einem System mit Kommunikation zwischen den Prozessen zu vergleichen.

Um die Auswirkungen des Verhältnisses „*Ausgangsgrad* zu *maximale Anzahl Guards pro Kante*“ auf die Verhältnisse zwischen den Anzahlen der „satisfied“-Ausgaben und „not satisfied“-Ausgaben überprüfen zu können, wurde in dieser Testserie alle möglichen Kombinationen von „Ausgangsgrad“ (von 1 bis 10) und „Maximale Anzahl Guards pro Kante“ (von 1 bis 10) in beiden Systemen überprüft.

Alle anderen Werte wurden willkürlich gewählt, um kleine, schnell überprüfbare Systeme zu erhalten.

Auf diese Systeme wurde jeder der drei möglichen Prozessgenerator-Methoden (siehe Kapitel 6) angewendet, jeweils mit folgenden identischen Optionen:

- Häufigkeit der Invarianten (in %): 0
- Breite-Faktor des Invarianten-Integrals: 0.2
- maximale Anzahl an Zuweisungen pro Kante: 1000
- Häufigkeit Zuweisungen an Variablen der Art $x = w$ in %: 14
- Häufigkeit Zuweisungen an Variablen der Art $x = x + w$ in %: 43
- Häufigkeit Zuweisungen an Variablen der Art $x = x - w$ in %: 43
- Breite-Faktor des Integrals, aus dem neue Werte für Zuweisungen gewählt werden, resultiert in mehr oder weniger schnell veränderlichen Integer-Variablen, reele Zahl zwischen -1 und 1: 1.2
- Breite-Faktor des Integrals für die Uhr, resultiert in „vergangener Zeit“ im Knoten: 0.20
- Chance auf Uhr-Reset in einer Kante in %=10
- min. Abstand des aktuellen Wertes vom Minimum der Variablenwerte, um noch $\leq$-Guards zuzulassen (dies verhindert unsinnige Guards, die verlangen, dass eine Variable über ihre Grenzen verändert wird): 3
- min. Abstand des aktuellen Wertes vom Maximum der Variablenwerte, um noch $\geq$-Guards zuzulassen (dies verhindert unsinnige Guards, die verlangen, dass eine Variable über ihre Grenzen verändert wird): 3
- Breite-Faktor des Integrals für Guards (siehe Abschnitt 6.6): 0.5

- maximale Länge der Pfade in Variante 2 (0 für kein Maximum): 0

Dabei wurde jede Kombination in jeder Methode (vergleiche Kapitel6) dreimal getestet, um so ein deutlicheres Ergebnis zu erhalten.

Um die Testserie möglichst unkompliziert zu halten, wurde hier auf das Erstellen von Invarianten verzichtet, d.h. die Häufigkeit in allen Einstellungen auf 0 gesetzt.

Die Breiten-Faktoren wurden in dieser Serie auf 0.5 gesetzt, um einen mittleren Wert zwischen 0 und 1 zu wählen (vergleiche Abschnitt 6.6).

Da sich keinen merklichen Unterschiede in den Ergebnissen von Einstellung 1 und Einstellung 2 gezeigt haben, beide resultierenden Systeme also ähnlich auf die verändernden Verhältnisse zwischen „*Ausgangsgrad* zu *maximale Anzahl Guards pro Kante*“ reagiert haben, wurden die Ergebnisse aufsummiert; in den folgenden Grafiken wird dies dadurch ausgedrückt, dass die verschiedenen Balken der Testreihen gleicher Verhältnisse gestapelt werden.

Da jede der drei Methoden des Generieren dreimal getestet wurde, über 2 verschiedene Einstellungen, ergibt sich für die folgenden Grafiken eine Maximale Anzahl von 18 maximal möglichen erfolgreichen Testreihen zu einem Verhältnis zwischen „*Ausgangsgrad* zu *maximale Anzahl Guards pro Kante*“. Die y-Achse ist entsprechend der Anzahlen der erfolgreichen Testreihen angepasst, um kleine Ergebnisse nicht zu verschlucken.

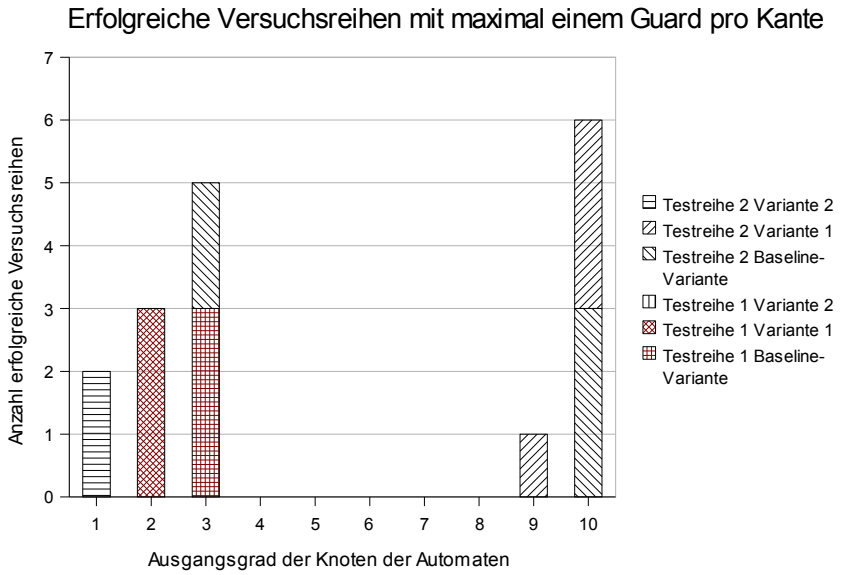


Abbildung 8.1: Auswertung bei *maximale Anzahl Guards pro Kante*=1

Wie leicht zu erkennen ist, werden generell die besten Ergebnisse erzielt, wenn der

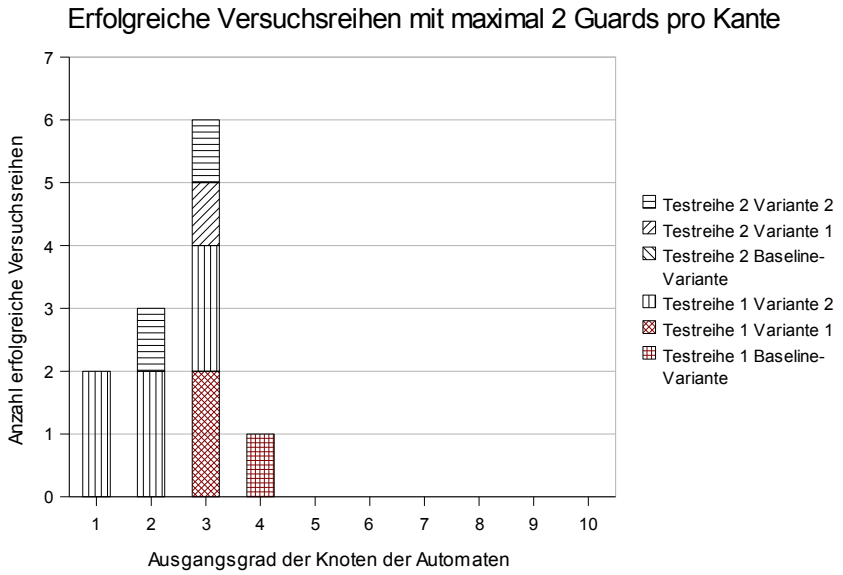


Abbildung 8.2: Auswertung bei *maximale Anzahl Guards pro Kante*=2

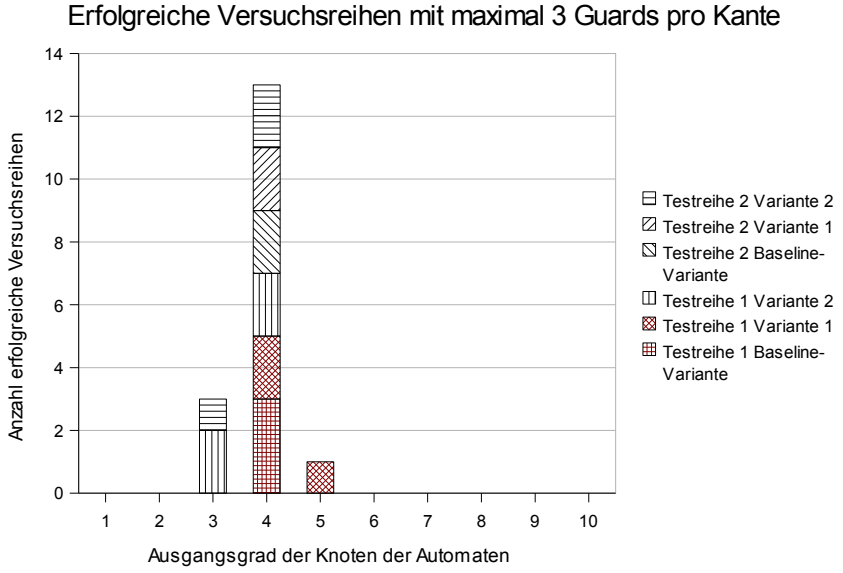


Abbildung 8.3: Auswertung bei *maximale Anzahl Guards pro Kante*=3

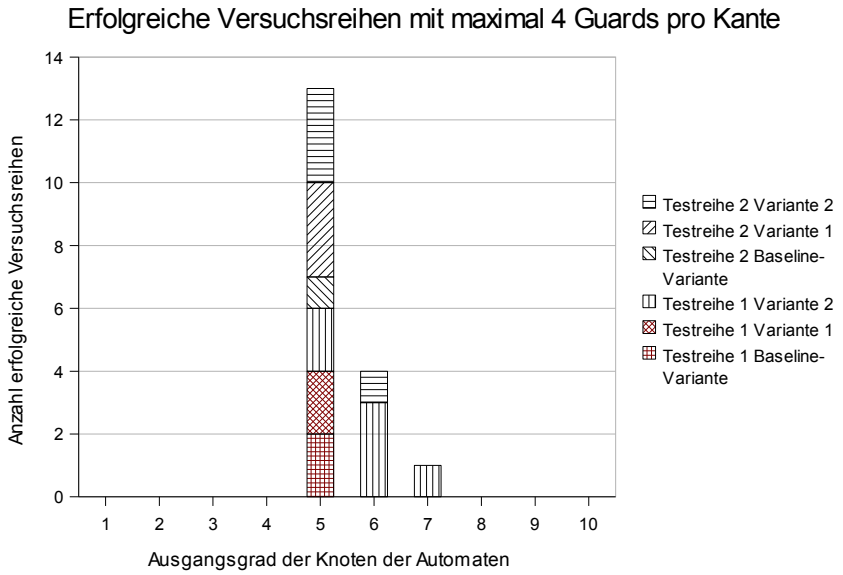


Abbildung 8.4: Auswertung bei *maximale Anzahl Guards pro Kante*=4

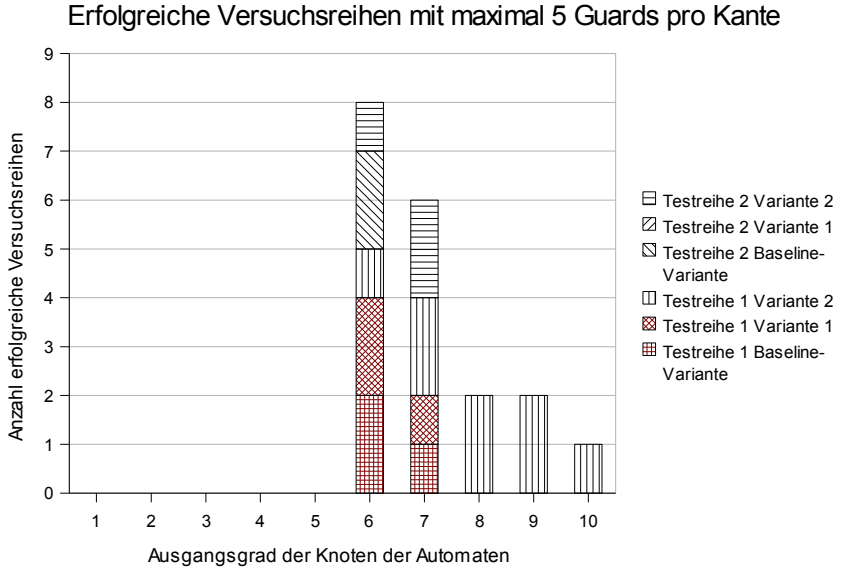


Abbildung 8.5: Auswertung bei *maximale Anzahl Guards pro Kante*=5

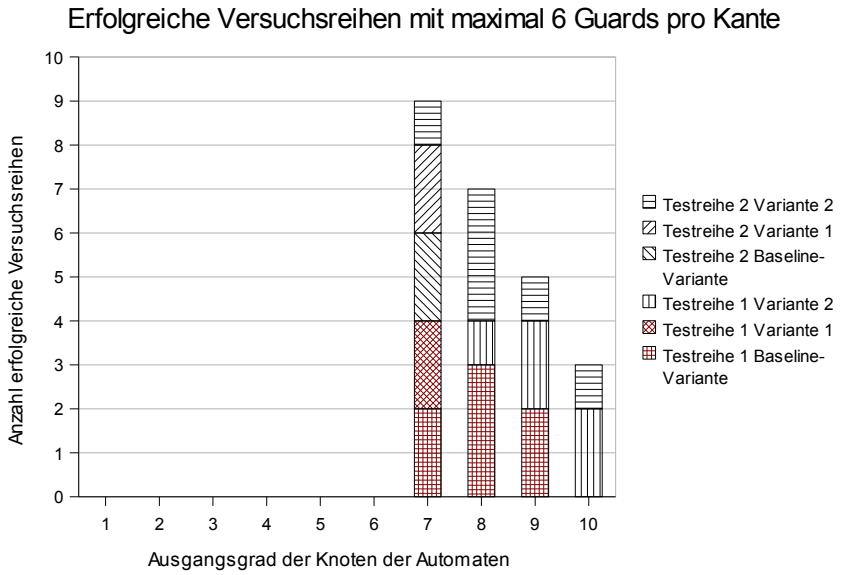


Abbildung 8.6: Auswertung bei *maximale Anzahl Guards pro Kante*=6

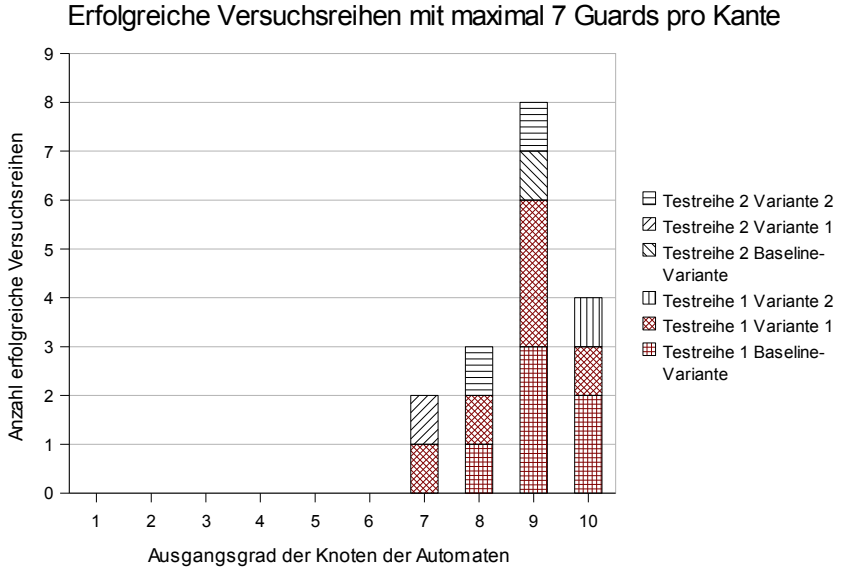


Abbildung 8.7: Auswertung bei *maximale Anzahl Guards pro Kante*=7

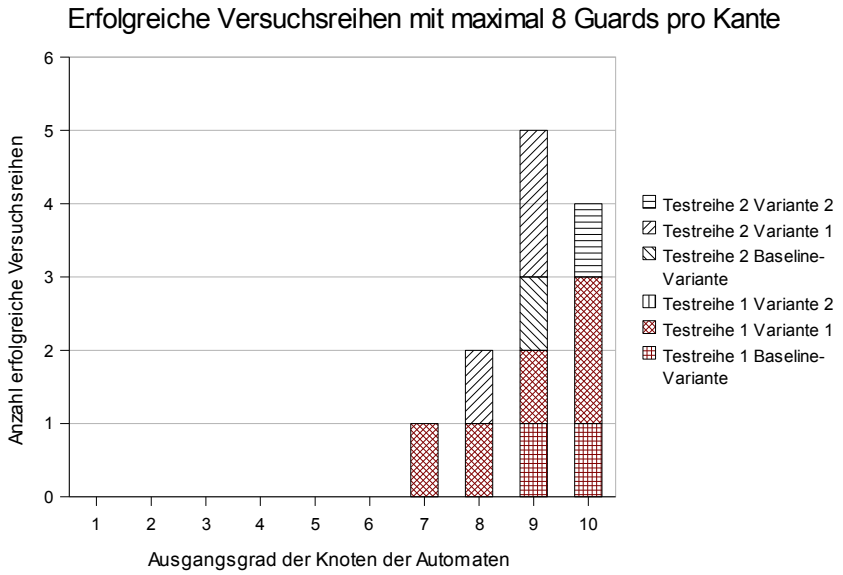


Abbildung 8.8: Auswertung bei *maximale Anzahl Guards pro Kante*=8

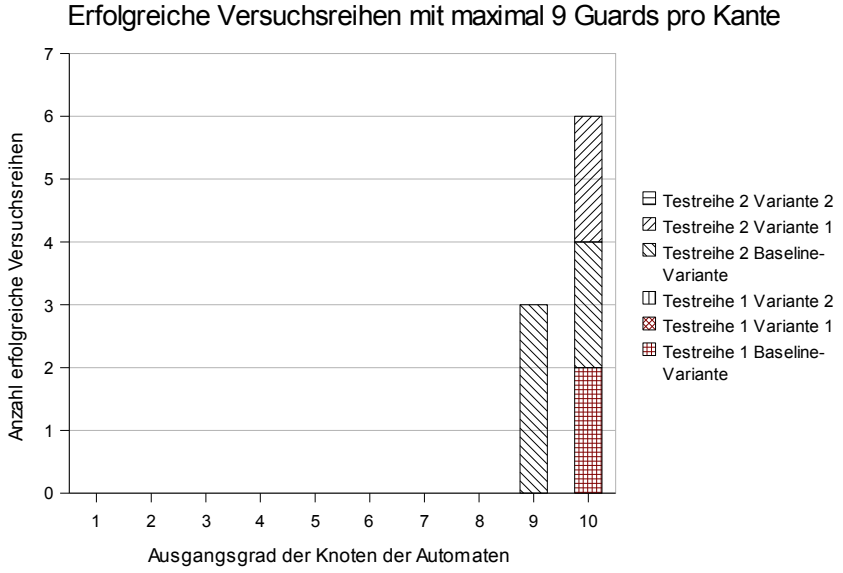


Abbildung 8.9: Auswertung bei *maximale Anzahl Guards pro Kante*=9

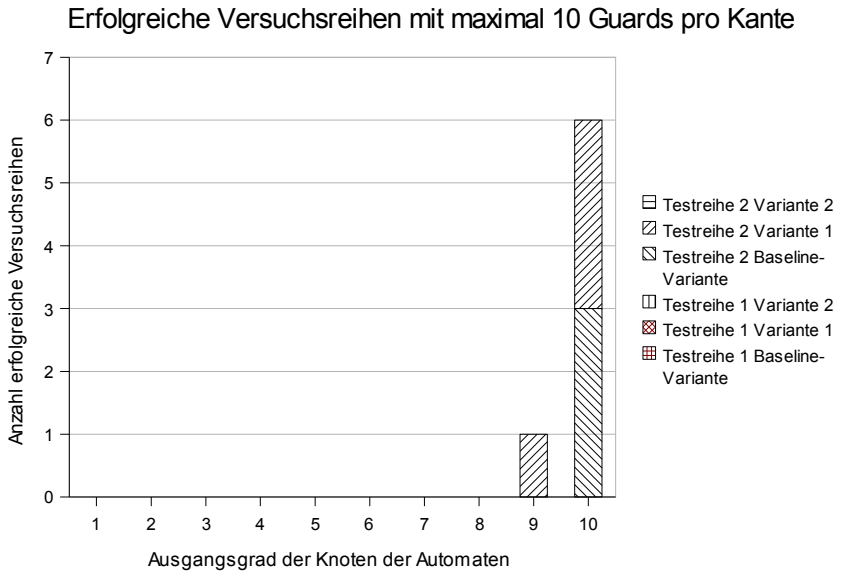


Abbildung 8.10: Auswertung bei *maximale Anzahl Guards pro Kante*=10

Ausgangsgrad in etwas gleich oder nur wenig größer ist als die *maximale Anzahl Guards pro Kante*.

Es wurde auch der Fall für *maximale Anzahl Guards pro Kante*=0 getestet, hier aber nicht aufgeführt, da die Ergebnisse keine Erfolge enthielten.

8.2 Testserie 2 — Auswirkung des Breiten-Faktors auf die Verhältnisse zwischen den Anzahlen der „satisfied“-Ausgaben und „not satisfied“-Ausgaben

Um die Auswirkungen des in Abschnitt 6.6 erwähnten Breiten-Faktor zu testen, wurde von einem System mit folgenden Einstellungen ausgegangen:

- durchschnittliche Anzahl Knoten pro Prozess: 10
- Anzahl Prozesse im System: 2
- Anzahl Uhren im System: 1
- Anzahl lokale Variablen pro Prozess: 3
- Anzahl globale Variablen im System: 1

- Anzahl Channels im System: 1
- Ausgangsgrad der Knoten: 4
- Maximal Anzahl Guards pro Kante: 3

Diese System-Konfiguration hatte sich in der vorangehenden Testreihe als sehr kompatibel mit allen drei Methoden zur Generierung der Prozesse herausgestellt und in jedem Falle ein zufriedenstellendes Verhältnis zwischen „satisfied"-Ausgaben und „not satisfied"-Ausgaben generiert.

Auf dieses System wurde jeder der drei möglichen Prozessgenerator-Methoden angewendet, jeweils mit folgenden Optionen:

- Häufigkeit der Invarianten (in %): 0
- Breite-Faktor des Invarianten-Integrals: 0.2
- maximale Anzahl an Zuweisungen pro Kante: 1000
- Häufigkeit Zuweisungen an Variablen der Art $x = w$ in %: 14
- Häufigkeit Zuweisungen an Variablen der Art $x = x + w$ in %: 43
- Häufigkeit Zuweisungen an Variablen der Art $x = x - w$ in %: 43
- Breite-Faktor des Integrals, aus dem neue Werte für Zuweisungen gewählt werden, resultiert in mehr oder weniger schnell veränderlichen Integer-Variablen, reele Zahl zwischen -1 und 1: 1.2
- Breite-Faktor des Integrals für die Uhr, resultiert in „vergangener Zeit" im Knoten: 0.20
- Chance auf Uhr-Reset in einer Kante in %=10
- min. Abstand des aktuellen Wertes vom Minimum der Variablenwerte, um noch $\leq$-Guards zuzulassen: 3
- min. Abstand des aktuellen Wertes vom Maximum der Variablenwerte, um noch $\geq$-Guards zuzulassen: 3
- Breite-Faktor des Integrals für Guards: variabel, siehe unten
- maximale Länge der Pfade in Variante 2 (0 für kein Maximum): 0

Breite-Faktor ist:	0.1	0.3	0.5	0.6	1.0
Anzahlen-Verhältnis der Ergebnisse in der Testreihe mit Baseline-Variante	2.06	1.94	1.95	1.67	1.43
Anzahlen-Verhältnis der Ergebnisse in der Testreihe mit Variante 1	1.07	1.64	1.12	1.44	0.90
Anzahlen-Verhältnis der Ergebnisse in der Testreihe mit Variante 2	0.59	0.91	0.62	0.68	0.79

Dabei wurden dieselben Optionen fünfmal angewendet, mit jeweils verschiedenen Größen für den Breiten-Faktor (Siehe Kantengenerierung), nämlich 0.1, 0.3, 0.5, 0.6 und 1.0.

Wie man in der Tabelle sieht, stellte sich keine eindeutige Auswirkung ein; die kann damit Erklärt werden, dass aufgrund der Symmetrie der Guards-Wahrscheinlichkeiten (vergleiche Abschnitt 4.3) nur ein Fünftel der Guards direkt von diesem Wert betroffen werden, und selbst in diesem Fall das Produkt von Breite-Faktor und plausibler Mittelwert des Guards (siehe Abschnitt 8) bedeutet, dass die Auswirkung dieses Faktors weiter abgeschwächt wird, bis er keine offensichtlichen Auswirkungen mehr hat.

8.3 Testserie 3 — Auswirkung der Invarianten auf die Verhältnisse zwischen den Anzahlen der „satisfied“-Ausgaben und „not satisfied“-Ausgaben

Wie in Testserie 2 wurde von einem System mit folgender Konfiguration ausgegangen:

- durchschnittliche Anzahl Knoten pro Prozess: 10
- Anzahl Prozesse im System: 2
- Anzahl Uhren im System: 1
- Anzahl lokale Variablen pro Prozess: 3
- Anzahl globale Variablen im System: 1
- Anzahl Channels im System: 1
- Ausgangsgrad der Knoten:4
- Maximale Anzahl Guards pro Kante: 3

Diese System-Konfiguration hatte sich in der vorangehenden Testreihe als sehr kompatibel mit allen drei Methoden zur Generierung der Prozesse herausgestellt und in jedem Falle ein zufriedenstellendes Verhältnis zwischen „satisfied“-Ausgaben und „not satisfied“-Ausgaben generiert.

Auf dieses System wurde jeder der drei möglichen Prozessgenerator-Methoden angewendet, jeweils mit folgenden Optionen:

- Häufigkeit der Invarianten (in %): variabel, siehe unten
- Breite-Faktor des Invarianten-Integrals: 0.2
- maximale Anzahl an Zuweisungen pro Kante: 1000
- Häufigkeit Zuweisungen an Variablen der Art $x = w$ in %: 14
- Häufigkeit Zuweisungen an Variablen der Art $x = x + w$ in %: 43
- Häufigkeit Zuweisungen an Variablen der Art $x = x - w$ in %: 43
- Breite-Faktor des Integrals, aus dem neue Werte für Zuweisungen gewählt werden, resultiert in mehr oder weniger schnell veränderlichen Integer-Variablen, reele Zahl zwischen -1 und 1: 1.2
- Breite-Faktor des Integrals für die Uhr, resultiert in „vergangener Zeit“ im Knoten: 0.20
- Chance auf Uhr-Reset in einer Kante in %=10
- min. Abstand des aktuellen Wertes vom Minimum der Variablenwerte, um noch $\leq$-Guards zuzulassen: 3
- min. Abstand des aktuellen Wertes vom Maximum der Variablenwerte, um noch $\geq$-Guards zuzulassen: 3
- Breite-Faktor des Integrals für Guards: 0.5
- maximale Länge der Pfade in Variante 2 (0 für kein Maximum): 0

Dabei wurden dieselben Optionen zweimal angewendet, mit jeweils verschiedenen Größen für die Verteilung der Invarianten, nämlich an keinem der Knoten (0%) und an 20% der Knoten.

Die Ergebnisse waren wie folgt verteilt:

	ohne Invarianten	mit Invarianten
Anzahlen-Verhältnis der Ergebnisse in der Testreihe mit Baseline-Variante	1.269	2.41
Anzahlen-Verhältnis der Ergebnisse in der Testreihe mit Variante 1	1.11	1.8
Anzahlen-Verhältnis der Ergebnisse in der Testreihe mit Variante 2	0.93	0.8

Wie man in der Tabelle sehen kann, verändern sich die Verhältnisse von „satisfied"-Ausgaben und „not satisfied"-Ausgaben derart, dass die relativen Anzahlen der „satisfied"-Ausgaben steigen, da die Automaten des Systems nun schwieriger zu durchlaufen sind. Ausnahme dazu ist die Variante 2 der Generators (siehe Algorithmus 5), hier bleibt das Verhältnis in dem Bereich, wo noch von einer erfolgreichen Testreihe gesprochen wird (siehe oben).

Eine Erklärung diese Ergebnisse ist im unterschiedlichen Aufbau-Vorgang der verschiedenen Prozess-Generatoren zu finden:

Sowohl die Baseline-Variante als auch Variante 1 basieren die von einem Knoten ausgehenden Kanten und deren Guards auf einem einzigen Werte-Set, dass dieser Knoten bei seiner Erstellung erhalten hat. Dieses Set wurde entweder beim verlegen einer Kante vergeben, im Falle der Baseline-Variante, möglicherweise auch zufällig.

Ist dieses Set nicht mit der Invariante des Knoten vereinbar, dann kann es vorkommen, dass ein Lauf, der von diesem Set unterschiedliche Werte benutzt, zwar nicht von der Invariante gehindert wird, aber dafür durch die Guards der ausgehenden Kanten gesperrt wird.

Effektiv wird der Knoten für das System undurchlaufbar.

Da Pfade in Variante 2 durch die einzelnen Knoten hindurch geplant und verlegt werden, erhöht sich die Anzahl an Paaren von Kanten hinein und heraus aus einem Knoten, d.h. Wege durch den Knoten hindurch.

Deshalb gibt es eine Vielzahl an Kanten, die in den Knoten hineinführen, die auch einen zugehörigen plausiblen, da bei der Konstruktion des Automaten angelegten, Ausgang aus dem Knoten haben — die Chance, dass sich darunter ein Paar befindet, das mit der Invariante vereinbar ist, erhöht sich entsprechend.

Auf diese Weise kommt es zu weniger Knoten, deren Invarianten nicht mit den möglichen Paaren an Ein- und Ausgangskanten vereinbar sind, was zu einer geringeren Beeinträchtigung des Automatenablaufs führt.

Ein Beispiel:
Ein Knoten s_i hat drei eingehende Kanten K_a, K_b, K_c und drei ausgehende Kanten K_x,

K_y, K_z. Der Knoten wurde erstellt, als Kante K_a verlegt wurde.
In Variante 1 und der Baseline-Variante bedeutet dies, dass K_x, K_y und K_z basierend auf den plausiblen Werten über K_a „geerbt“ erstellt wurden, was zur Folge hat, dass ein Lauf L_1, der s_i kommend über K_a betritt, weit höhere Chancen hat, s_i wieder über K_x, K_y oder K_z zu verlassen, als ein anderer Lauf, der s_i über K_b oder K_c betritt.

Hat nun s_i eine Invariante, die mit L_1 nicht vereinbar ist, wird der Knoten s_i gesperrt, d.h. der Lauf endet.

Die einzige Möglichkeit, s_i dennoch durchlaufen zu können, ist s_i über K_b oder K_c zu betreten, was aber die erhöhte Wahrscheinlichkeit birgt, dass K_x, K_y und K_z das Verlassen von s_i verhindern, da sie nicht auf K_b oder K_c abgestimmt sind.

In Variante 2 werden die Guards von K_x, K_y und K_z basierend auf mehreren plausiblen Werten, „geerbt“ durch K_a, K_b und K_c, erzeugt, was bedeutet, dass auch ein Lauf in s_i über K_b oder K_c gute Chancen hat, s_i wieder zu verlassen. Durch die erhöhte Auswahl ist es eher möglich, dass ein Lauf existiert, den die Invariante von s_i nicht sperrt, und der K_x, K_y oder K_z benutzen kann.

9 Das Programm

Ein Programm, das die in dieser Arbeit vorgestellten Methoden verwendet und zusammen mit Uppaal verwendet werden kann, wurde im Verlaufe dieser Arbeit geschrieben. Dabei wurde die Programmiersprache Python verwendet, da es in dieser besonders einfach ist, das Programm unabhängig vom Betriebssystem des verwendeten Computers zu schreiben. Die verwendete Version von Python war 2.5; das Programm ist mit Compilern ab Python 2.4 kompatibel.

Auf Wunsch kann das Programm die generierten Muster gleich an die entsprechende Uppaal-Komponente weiterleiten, und eine Überprüfung durchführen. Dazu muss das im Uppaal-Paket enthaltene Kommandozeilen-Programm „verifyta“ passend zum Betriebssystem aus dem Uppaal-Paket in das Verzeichnis von des Generators kopiert werden. Anschließend können Automaten nach den gegebenen Einstellungen generiert und direkt automatisch verifiziert werden. Das Programm hält dabei automatisch die Anzahlen-Verhältnisse der „satisfied“/„not satisfied“-Probleme fest, und berechnet anschließend den Quotienten (siehe Kapitel 8). Auf diese Weise ist es möglich, passende Einstellungen und Zufallsparameter zu finden, die die gewünschte Verteilung an Ergebnissen bieten.

9.1 Benutzeranleitung

Das Programm verfügt über zwei verschiedene Benutzer-Interfaces:

Ein graphisches Interface, weitestgehend selbsterklärend, kann durch Aufrufen der Datei „GUI.py“ mittels „python GUI.py“ gestartet werden.

Dabei muss eine Datei mit den Zufalls-Parametern, die während der Erstellung benutzt werden sollen, angegeben werden. Standardmäßig ist dies die Datei „Zufallsparameter.ini“.

Ein Kommandozeilen-Interface kann durch Aufrufen der Datei „RTA.py“ mittels „python RTA.py“ gestartet werden. Dem Kommandozeilen-Interface muss als erstes Argument eine Parameter-Datei angegeben werden, die die Konfiguration des Systems, der Automaten des Systems und die Angaben über die verschiedenen Ein- und Ausgabedateien für den Generator enthalten muss. Dazu später mehr.

Optionale Parameter sind „-p“, „-M:[1,2,3]“ und “-l“, wobei „-p“ das Programm dazu veranlasst, das Uppaal-Kommandozeilenprogramm zum sofortigen Testen zu verwenden, und „-M:[1,2,3]“ eine Auswahl der zu verwendenden Prozessgenerator-Variante ermöglicht (in Reihenfolge: der Baseline-Ansatz, Variante 1 und Variante 2, siehe Kapitel 6). Standardmässig wird „-M:3“ angenommen. Der Parameter „-l“ dient dazu, nach einem Programmdurchlauf im Test-Modus die erzeugten Muster-Dateien automatisch zu löschen, um einen Übersichtsverlust durch eine Überzahl an Testmusterdateien zu verhindern. Es werden nur die im Test-Modus erzeugten Auswertungs-Dateien zurückgelassen (siehe unten). Hierbei muss beachtet werden, dass dabei alle Muster-Dateien im Verzeichnis gelöscht werden, weshalb dieser Parameter standardmässig nicht gesetzt ist.

9.2 Ein- und Ausgabe-Dateien

Solange das Programm nicht im Test-Modus gestartet wird, generiert es Textdateien, die RZA in einem Uppaal-kompatiblen Format enthalten und die dazugehörigen zufälligen Queries, die den zu überprüfenden Fehler definieren.

Die dazu verwendeten Eingabe-Dateien sind:

„status.ini“: Diese Datei enthält, in vier Zeilen, die Angaben der gewünschten Benennung, des Formates und der Version der RZA-Dateien und den Stamm-Namen der Ausgabe-Dateien für den Tests-Modus falls das Programm mittels GUI gestartet wird.

Zufallsparameter-Datei: Enthält die Parameter für den Prozessgenerator, die die Wahrscheinlichkeiten des Generators beeinflussen. Ein Beispiel für eine solche Datei ist die Datei „Zufallsparameter.ini“.

- Häufigkeit der Invarianten (in %)
- Breite-Faktor des Invarianten-Integrals: eine reelle Zahl zwischen 0 und 1, gibt an, wie gestreut der Wert einer Invariante um den vermuteten plausiblen Zeitwert generiert wird.
- maximale Anzahl an Zuweisungen pro Kante
- Häufigkeit Zuweisungen an Variablen der Art $x = w$ in %
- Häufigkeit Zuweisungen an Variablen der Art $x = x + w$ in %
- Häufigkeit Zuweisungen an Variablen der Art $x = x - w$ in %

- Breite-Faktor des zusätzlichen Integrals, aus dem neue Werte für Zuweisungen gewählt werden, resultiert in mehr oder weniger schnell veränderlichen Integer-Variablen, reele Zahl zwischen -1 und 1
- Breite-Faktor des Integrals für die Uhr, resultiert in „vergangener Zeit“ im Knoten
- Chance auf Uhr-Reset in einer Kante in %
- min. Abstand des aktuellen Wertes vom Minimum der Variablenwerte, um noch $\leq$-Guards zuzulassen: dies verhindert, dass ein Guard verlangt, das ein nicht erreichbarer Wert vorliegen muss.
- min. Abstand des aktuellen wertes vom Maximum der Variablenwerte, um noch $\geq$-Guards zuzulassen: dies verhindert, dass ein Guard verlangt, das ein nicht erreichbarer Wert vorliegen muss.
- Breite-Faktor des Integrals für Guards: eine reelle Zahl zwischen 0 und 1, gibt an, wie gestreut eine Wert eines Guards um den vermuteten plausiblen Wert generiert wird.
- maximale Länge der Pfade in Variante 2 (0 für kein Maximum): diese Einstellung zwingt den Generator dazu, den die Länge des unbekannt langen Teil eines Pfades (siehe Abschnitt 6.4) auf ein vorgegebenes Maximum zu beschränken. Diese Einstellung ist mit Vorsicht zu behandeln, da sie die Laufzeit des Generators erheblich erhöhen kann.

Die Parameter-Datei des Kommandozeilen-Interface: Diese Text-Datei besteht aus einem oder mehreren Blöcken der folgenden Art:

- durchschnittliche Anzahl Knoten pro Prozess
- Anzahl Prozesse im System
- Anzahl Uhren im System
- Anzahl lokale Variablen pro Prozess
- Anzahl globale Variablen im System
- Anzahl Channels im System
- Ausgangsgrad der Knoten

- Maximal Anzahl Guards pro Kante
- Name der Zufalls-Parameter-Datei, z.B. „Zufallsparameter.ini“
- Name der Ergebnis-Datei (falls Test), z.B. „Ergebnisse.txt“
- Max Delay bei Test: bestimmt, wie viele Sekunden Zeit verifyta gegeben wird, ein einzelnes Problem zu überprüfen.
- Anzahl Durchgänge: bestimmt, wie viele Exemplare dieser Konfiguration generiert oder generiert und überprüft werden
- ein „;“ schließt einen solchen Block ab. Anschließend kann ein weiterer solcher Block folgen, um ganze Testserien zu ermöglichen

Ein Beipiel für eine solche Datei ist mit der Datei „Parameter.ini“ gegeben.

Während des Testens werden mehrere Dateien generiert: Es wird für jede Testreihe eine Zusammenfassung der Ausgaben von „verifyta“ aufgestellt, eine Zählung der Anzahlen der „satisfied“/„not satisfied“-Probleme und eine Datei mit den sich ergebenden Anzahlenverhältnissen pro Zählung. Abschließend wird eine Zusammenfassung für alle Blöcke der Parameter-Datei erstellt, die einen direkten Vergleich zwischen den Parametern und den Ergebnissen erlaubt. Bei Benutzen des graphischen Interfaces wird diese Zusammenfassung nicht erstellt, da sie nur den letzten Durchlauf mit sich selbst vergleichen könnte.

9.3 Gliederung und Aufbau

Im folgenden werden die einzelnen Module des Programms und die darin enthaltenen Klassen und Funktionen kurz vorgestellt und erläutert.

9.3.1 Datei „GUI.py“

Dieses Modul enthält die Funktionen und Klassen der GUI, und startet nach Einlesen der vom User mittels GUI und „Zufallsparameter.ini“ eingegebenen Daten.

Die Funktionen in diesem Modul sind abhängig von den Modulen generator.py, fileops.py und testrunner.py. Sie benutzt die Standard-Module string.py und Tkinter.py.

9.3.2 Datei „RTA.py“

Dieses Modul enthält die Funktion der Kommandozeilen-Eingabe, und startet den Generator nach Einlesen der vom Benutzer mittels Parameter-Datei und „Zufallsparameter.ini“ eingegebenen Daten.

Die Funktionen in diese Modul sind abhängig von den Modulen generator.py, fileops.py, testrunner.py und Auswerter.py. Sie benutzt das Standard-Modul sys.py.

9.3.3 Datei „generator.py“

Dieses Modul enthält die Funktionen, die in Abschnitt 6 beschreiben worden sind, und einige kleinere Hilfsfunktionen.

Die Funktionen in diesem Modul sind abhängig von den Modulen objparser.py, fileops.py und conditional_random.py. Sie benutzt das Standard-Modul random.py.

9.3.4 Datei „fileops.py“

Dieses Modul enthält sämtliche Funktionen, die während des Programmablaufs benötigt werden, um Dateien zu lesen und zu schreiben.

9.3.5 Datei „objparser.py“

Dieses Modul enthält alle Klassen, die zur Konstruktion eines Systems an Realzeit-Automaten vom Generator benötigt werden. Diese Klassen besitzen die nötigen Methoden, ihre Eigenschaften als Text im gewünschten Uppaal-Format und -Version auszulesen.

9.3.6 Datei „testrunner.py“

Dieses Modul enthält die nötigen Funktionen, um die Interaktion zwischen dem Generator und „verifyta“ zu ermöglichen. Dabei ist für jedes der drei Betriebssysteme Windows, Linux und SunOS eine entsprechende, automatisch gewählte Funktion vorhanden.

Dieses Modul benutzt die Standard-Module subprocess.py, time.py, os.py und sys.py.

9.3.7 Datei „querycreator.py“

Diese Modul benutzt das Standard-Modul random.py, und enthält die nötige Funktion, um Queries passend zu den vom Generator erzeugten Mustern zu generieren.

9.3.8 Datei „Auswerter.py“

Diese Modul benutzt das Standard-Modul os.py, und enthält die nötige Funktion, um während einer Testreihe den entsprechenden Quotienten zwischen den „satisfied“ und „not satisfied“-Anzahlen zu bilden.

9.3.9 Datei „conditional_random.py“

Dieses Modul enthält die angepassten Random-Funktionen, die im Verlauf des Programms benötigt werden (vergleiche Algorithmus 10).

Die Funktionen in diesem Modul benutzen das Standard-Modul random.py.

10 Zusammenfassung

Ziel dieser Arbeit war es, einen Zufallsgenerator für Realzeit-Automaten zu entwerfen, der die generierten Automaten und deren umgebenden Systeme im Uppaal-Format erstellen kann.

Dabei wurden verschiedene Ansätze zum Generieren von DFA und NFA hinzugezogen (vergleiche Kapitel 3), die Unterschiede in der Problemstellung beim Generieren von RZA festgestellt und entsprechende Methoden entwickelt, um diese Probleme zu bewältigen.

Wie gezeigt, kann mit den hier vorgestellten Methoden bei geeigneter Konfiguration des Generators mit Zuverlässigkeit zufällige Automaten von gewünschter Form und Format erzeugt werden, die dann die gewünschte Verteilung zwischen „Fehler erreichbar“ und „Fehler nicht erreichbar“ aufweisen.

Es können allerdings keine festen Regeln für die Parameter des Generators angegeben werden, die immer dazu führen, dass ein generiertes System mit bestimmter Wahrscheinlichkeit einen erreichbaren/nicht erreichbaren Fehler hat; im Vergleich zur Analyse endlicher Automaten ist die Analyse von Realzeit-Automaten durch den zusätzlichen Einfluss der Guards und Invarianten bedeutend schwerer, so dass nur grobe Richtlinien für die Eingaben an den Generator vorgegeben werden können (vergleiche Kapitel 8) — eine Feinabstimmung dieser Einstellungen muss dann per Experimentieren mittels der vom Programm zur Verfügung gestellten Funktionen vorgenommen werden, bis das gewünschte System mit zufriedenstellender Verteilung generiert wird.

So sollte der „Ausgangsgrad der Knoten“ immer bei oder knapp über der „maximalen Anzahl Guards pro Kante“ (siehe Abschnitt 9.1) liegen, weil ein erhöhendes Ausgangsgrad die Erreichbarkeit des für das System definierten Fehlers erhöht, und damit die relative Anzahl der „not satisfied“-Ergebnisse, und ein Erhöhen der maximalen Anzahl der Guards pro Kante die Erreichbarkeit verringert, und damit die relative Anzahl der „satisfied“-Ergebnisse. Da die Anwesenheit von Invarianten die Erreichbarkeit des Fehlers ebenfalls verringert, und damit die relative Anzahl der „satisfied“-Ergebnisse, muss entsprechend der Ausgangsgrad erhöht werden, um so die Erreichbarkeit des Fehlers wieder zu erhöhen, und eine zufriedenstellende Verteilung der gewünschten Automaten

zu erhalten.

Die Veränderlichkeit der Werte der Integer-Variablen und Uhren des Systems kann je nach Wunsch verändert werden, um so ein schneller oder langsamer veränderliches System zu simulieren (siehe Abschnitt 9.1).

Die restlichen Einstellungen können so belassen werden, wie sie in den Beispiel-Dateien (siehe Abschnitt 9.1) vorgegeben sind, da sie keinen wesentlichen Einfluss auf die Erreichbarkeit des Fehlers haben, aber genutzt werden können, um Systeme von verschiedenem Aufbau zu generieren.

Literaturverzeichnis

[1] Bernd Becker, Andreas Podelski, Werner Damm, Martin Fränzle, Ernst-Rüdiger Olderog, and Reinhard Wilhelm. Sfb/tr 14 avacs - automatic verification and analysis of complex systems (der sonderforschungsbereich/transregio 14 avacs - automatische verifikation und analyse komplexer systeme). *it - Information Technology*, 49(2):118–, 2007.

[2] Karl Bosch. *Elementare Einführung in die Wahrscheinlichkeitsrechnung.* Vieweg-Verlag, 1999.

[3] Thomas Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS '96)*, pages 278–292, New Brunswick, New Jersey, 1996.

[4] R. Milner. *Communication and concurrency.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[5] Jonathan S. Ostroff. *Temporal logic for real time systems.* John Wiley & Sons, Inc., New York, NY, USA, 1989.

[6] C. Ramchandani. Analysis of asynchronous concurrent systems by timed petri nets. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1974.

[7] T.A. Henzinger, X. Nicollin, J. Sifakis und S. Yovine. Symbolic Model Checking for Real-Time Systems. In *7th. Symposium of Logics in Computer Science*, pages 394–406, Santa-Cruz, California, 1992. IEEE Computer Scienty Press.

[8] Harry R. Lewis und Christos H. Papadimitriou. Elements of the theory of computation. *SIGACT News*, 29(3):62–78, 1998.

[9] Rajeev Alur und Costas Courcoubetis und David L. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.

[10] Frédérique Bassino und Cyril Nicaud. Enumeration and random generation of accessible automata. *Theor. Comput. Sci.*, 381(1-3):86–104, 2007.

[11] Rajeev Alur und David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[12] Jean-Marc Champarnaud und Georges Hansel und Thomas Paranthoën und Djelloul Ziadi. Random generation models for NFA's. *J. Autom. Lang. Comb.*, 9(2-3):203–216, 2004.

[13] Nancy Lynch und Hagit Attiya. Using mappings to prove timing properties. In *PODC '90: Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, pages 265–280, New York, NY, USA, 1990. ACM.

[14] Bernd Krieg-Brückner und Jan Peleska und Ernst-Rüdiger Olderog und Alexander Baer. The UniForM workbench, a universal development environment for formal methods. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems*, volume 1709 of *LNCS*, pages 1186–1205. Springer-Verlag, 1999.

[15] Kim Guldstrand Larsen und Paul Pettersson und Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[16] Johan Bengtsson und Wang Yi. Timed automata: Semantics, algorithms and tools, 2004.

[17] Thomas A. Henzinger und Z. Manna und Amir Pnueli. Temporal proof methodologies for real-time systems. Technical report, Stanford University, Stanford, CA, USA, 1991.