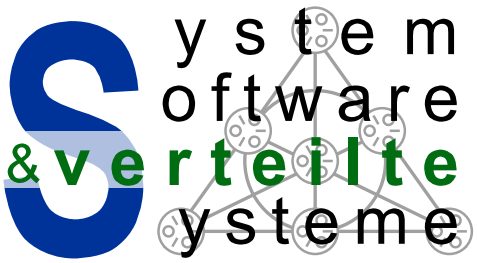
CARL VON OSSIETZKY universität OLDENBURG

DEPARTMENT FÜR INFORMATIK
SYSTEMSOFTWARE UND VERTEILTE SYSTEME

Simulationsumgebung zur Transformation selbststabilisierender Algorithmen

Bachelorarbeit



19. September 2011

Leon Winter
Verbindungsweg 23
26188 Edewecht

Erstprüfer	Prof. Dr.-Ing. Oliver Theel
Zweitprüfer	MSc Abhishek Dhama

Erklärung zur Urheberschaft

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel und Quellen benutzt habe.

Oldenburg, den 19. September 2011

Leon Winter

Diese Abschlussarbeit befasst sich mit der Simulation von transformierten selbststabilisierenden verteilten Algorithmen. Durch einen speziellen Transformationsprozess können selbststabilisierende Algorithmen unter jedem beschränkt fairen Scheduler konvergieren. Zur besseren Klassifizierung der Hilfsalgorithmen des Transformationsvorgangs und zur Analyse des Konvergenzverhaltens wurde daher eine Simulationsumgebung entworfen.

This thesis is about the simulation of transformed self-stabilizing distributed algorithms. Transformed through a special process self-stabilizing algorithms converge under any weakly fair scheduler. In order to classfiy the auxiliary algorithms used in the transformation process and to study the convergence behaviour a simulation environment was developed.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Seit der industriellen Revolution in Großbritannien werden immer mehr Tätigkeiten der Menschen von Maschinen übernommen. Zunächst durch mechanische Geräte, später durch elektronische Schaltungen und heute durch den Mikrochip. Doch im Streben um höhere Leistungsfähigkeit der Rechenmaschinen ist das physikalische Maximum bald ausgeschöpft, sodass eine höhere Leistung anderweitig erzielt werden muss. Diesem Problem soll durch Kombination mehrer Recheneinheiten (Parallelisierung) begegnet werden. Doch lässt sich ein System nicht mit beliebig vielen Prozessoren ausstatten, sodass man das Konzept ausweitete und fortan auch die Rechnersysteme selbst miteinander verband. Die erschaffenen *verteilten Systeme* verfügen somit über eine sehr große kombinierte Rechenleistung.
Durch die ständig wachsende Leistungsfähigkeit und dadurch sinkende Kosten ist die Digitalisierung inzwischen allgegenwärtig. Dies hat unweigerlich zur Folge, dass kritische Vorgänge wie der Betrieb eines Atomkraftwerkes vollkommen computergesteuert ablaufen. Ein Fehlverhalten wie zum Beispiel ein Absturz eines Rechnersystems kann dramatische Folgen für die Menschen und die Umwelt haben. Es ist daher von enormer Wichtigkeit, die richtige Funktionsweise kritischer Systeme dauerhaft sicherzustellen.

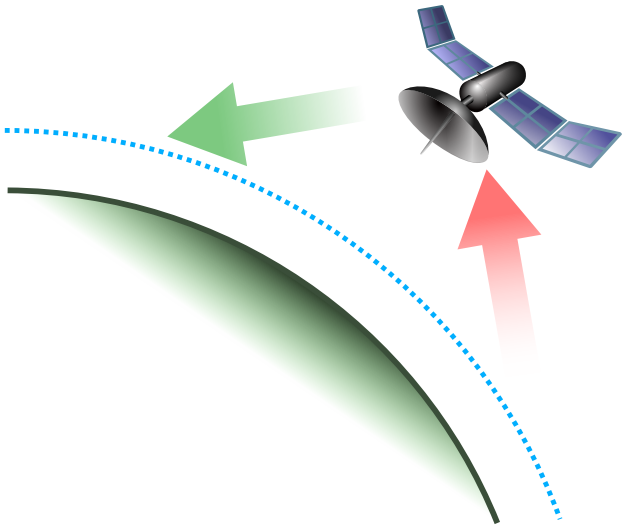


Abbildung 1.1: Ablenkung eines Satelliten durch fehlerhafte Steuerung und Kurskorrektur

Konsequenterweise werden *fehlertolerante* Systeme für kritische Anwendungen eingesetzt. Diese sind in der Lage, mit Fehlern umzugehen und bei diesen die negativen Auswirkungen zu reduzieren. *Fehlersichere* Systeme im Speziellen sind durch ihren internen Aufbau in der Lage, Fehler bis zu einer bestimmten Größenordnung vollkommen zu kompensieren. Zeitkritische Sicherheitssysteme, wie zum Beispiel ein Notbremssystem, müssen stets die richtige Entscheidung treffen und dürfen durch Fehlfunktionen nicht beeinträchtigt werden. Es ist festzustellen, das fehlersichere System das Optimum an Fehlertoleranz bieten und dementsprechend einen sehr großen Entwicklungsaufwand erfordern. Für viele Anwendungsszenarien der Fehlertoleranz reicht jedoch ein System, welches sich nach einem Fehlerfall wieder heilen kann und nach einer bestimmten Zeit wieder voll funktionsfähig ist. Als Beispiel sei hier ein geostationärer Satellit genannt, dessen Anwendungsspeicher durch kosmische Strahlung verfälscht wurde. Das Steuerungssystem des Satelliten wird aufgrund der korrumpierten Daten nun fehlerhafte Lenkungsentscheidungen treffen. Doch solange die fehlerhaften Daten in einer bestimmten Zeit durch korrekte Daten ersetzt werden

können, kann die Steuerung den Kurs korrigieren und der Satellit die Laufbahn halten. Systeme dieser Art heißen *selbststabilisierende* Systeme und sollen in dieser Arbeit besonders betrachtet werden.
Es existieren inzwischen diverse selbststabilisierende Algorithmen für einen bestimmten Zweck. Um jedoch komplexere Probleme zu lösen, ist es nötig, mehrere Algorithmen zu kombinieren. Hierbei muss die Eigenschaft der Selbststabilisierung für das Gesamtsystem erhalten bleiben. Die Beweisführung, dass die Selbststabilisierung erhalten bleibt, ist jedoch sehr kompliziert, da die einzelnen Algorithmen jeweils eine ganz spezielle Ablaufsteuerung (*Scheduler*) erfordern und diese untereinander nicht kompatibel sein können. Der elegante Lösungsansatz nach [DT10] sieht hierfür vor, die Algorithmen zu alternieren und die Ablaufsteuerung lokal zu emulieren. Da die Ablaufsteuerung jedoch Kenntnis über das gesamte System haben muss, darf nur ein Prozess zur Zeit aktiv sein und dieser muss jeweils den aktuellen Systemzustand weiterreichen. Durch diesen *gegenseitigen Ausschluss* ist sichergestellt, dass die Ablaufsteuerung aufgrund von korrekten Informationen eine Entscheidung trifft. Es existieren allerdings keine Algorithmen für den gegenseitigen Ausschluss in einem beliebig vermaschten Rechnernetz, wohl aber für Ring-Strukturen. Zwangsläufig wird zur Zusammenführung (*Komposition*) mehrerer selbststabilisierender Algorithmen zunächst eine aufspannende Baumstruktur gebildet, welche danach ringförmig durchlaufen werden kann. Auf diesem virtuellen Ring wird durch gegenseitigen Ausschluss die spezielle Ablaufsteuerung emuliert und damit die Selbststabilisierung des Gesamtsystems sichergestellt.

1.0.1 Problem

Die Komposition mehrerer selbststabilisierender Algorithmen erfordert mehrere Komponenten. Zunächst muss eine aufspannende Baumstruktur generiert werden. Danach wird diese Struktur ringförmig durchlaufen um gegenseitigen Ausschluss zu realisieren. Damit das Gesamtsystem selbststabilisierend ist, müssen die Algorithmen zur Bildung des aufspannenden Baums und zum gegenseitigen Ausschluss ebenfalls selbststabilisierend sein. Für diese Zwecke existieren eine Vielzahl von Algorithmen. So gibt es beispielsweise lokalen und globalen gegenseitigen Ausschluss.
Die Wahl der Algorithmen und dessen Kombination beeinflusst, wie schnell das Gesamtsystem nach einem Fehler wieder korrekt läuft (*konvergiert*). Doch auch der Aufbau des verteilten Systems hat Einfluss auf die Konvergenz. So können Algorithmen unter bestimmten Topologien schneller konvergieren als andere. Es ist daher erstrebenswert, in Erfahrung zu bringen, welche Algorithmen und Konfigurationen sich für welche Probleme eignen. Durch diese Erkenntnisse könnten bei der Transformation stets eine optimale Auswahl der Algorithmen getroffen werden, sodass der Zeitraum von Fehler zu Konvergenz minimiert und damit die Zeit der korrekten Arbeit erhöht werden kann.

1.0.2 Lösungsansatz

Um Erkenntnisse über die Auswahl und Konfigurationen von selbststabilisierenden Algorithmen für den aufspannenden Baum und gegenseitigen Ausschluss im Rahmen der Komposition selbststabilisierender Anwendungsalgorithmen (*Transformation*) zu finden, wird nun folgende Vorgehensweise vorgeschlagen: Eine Simulationsumgebung für den Transformationsalgorithmus mit Modulen für die Kompositionsalgorithmen mit großer Konfigurierbarkeit soll Aufschluss über das Konvergenzverhalten unter bestimmten Eingabedaten liefern. Bei systematischer Vorgehensweise der Simulation können somit allgemeine Erkenntnisse über die Eignung der einzelnen Algorithmen gezogen werden.
Die Simulationsumgebung sollte dafür modular gestaltet, hoch konfigurierbar, leicht erweiterbar und verständlich in der Bedienung sein. Um das Simulieren weiter zu vereinfachen, sollten die Anwendungsalgorithmen mit möglichst geringer Abweichung gegenüber ihrer originalen Repräsentation eingegeben werden können. Die Ausgabe sollte textuell und graphisch erfolgen können und gepaart mit einer statistischen Analyse der Ergebnisse bei der Interpretation der Simulation unterstützen.

1.0.3 Überblick

Die Abschlussarbeit ist thematisch in mehrere Kapitel unterteilt, die aufeinander aufbauen.
Mit einer kurzen Problembeschreibung und Situationsbeschreibung eröffnet diese Einleitung als erstes Kapitel.
Der zweite Teil der Arbeit befasst sich mit der Theorie der selbststabilisierenden verteilten Systeme (2). Dabei wird zunächst der Forschungsgegenstand des verteilten Systems definiert (2.1). Im weiteren Verlauf wird dann die Eigenschaft der Fehlertoleranz eingeführt (2.2). Das Kapitel schließt mit der Vorstellung der selbststabilisierenden Algorithmen (2.3).
Im dritten Kapitel wird auf die Transformation von selbststabilisierenden Algorithmen eingegangen (3). Konkret wird dabei erst einmal die Absicht hinter der Transformation erläutert (3.1). Danach steht das mehrteilige Vorgehen der Transformation im Vordergrund und wir in Unterkapiteln je erläutert. Besonders ausführlich wird dabei auf die Hilfsalgorithmen eingegangen (3.2.1 und 3.2.2).
Im vierten Teil wird dann die entwickelte Simulationsumgebung beschrieben (4). So wird zu Beginn die Zielsetzung umrissen um dann die Lösungskonzepte vorzustellen.
Die Ergebnisse konkreter Simulationsläufe sind dann im fünften Kapitel zu sehen (5).
Zum Schluss folgt im sechsten Teil der Arbeit eine Zusammenfassung und ein Ausblick in Zukunft (6).

2 Selbststabilisierende Algorithmen

In diesem Kapitel definieren wir zunächst verteilte Systeme (2.1) und ihre Funktionsweise. Im weiteren findet eine Spezialisierung auf Fehlertoleranz (2.2) statt, um abschließend weiter auf selbststabilisierende Systeme (2.3) zu verfeinern, die Betrachtungsgegenstand dieser Arbeit sind.

2.1 Verteilte Systeme

Da einzelnen Rechnersysteme in ihrer Leistungsfähigkeit und Ausfallsicherheit Grenzen gesetzt sind, werden *verteilte Systeme* eingesetzt. Nachfolgend wird analog nach [DT10] das Modell eines verteilten Systems definiert.

Verteiltes System Ein *verteiltes System* ist ein Verbund von n Prozessen $\Pi := \{P_1, \cdots, P_n\}$. Die Prozessoren sind nach einer Topologie angeordnet und tauschen Nachrichten über Kommunikationsregister miteinander aus. Direkt mit einem Prozess verbundene andere Prozesse bilden die Nachbarschaft des erstgenannten Prozesses.

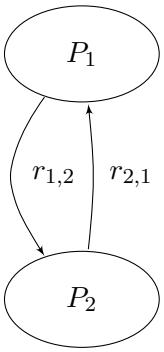


Abbildung 2.1: Bidirektionale Kommunikation der benachbarten Prozesse P_1 und P_2

Kommunikationsregister Prozesse kommunizieren jeweils über Register. Ein *Register* ist nach dem Konzept des Nachrichtenaustauschs (*message passing*) eine sich veränderte Information, auf die benachbarte Prozesse Zugriff haben. Sie werden dabei häufig und auch in dieser Arbeit als gemeinsamer Speicherbereich (*shared memory*) implementiert. Sofern nur ein Register zwischen zwei benachbarten Prozessen existiert, spricht man von einer **unidirektionalen** Kommunikation, gibt es zwei für den gegenseitigen Nachrichtenaustausch von einer **bidirektionalen**. Im Rahmen dieser Arbeit werden lediglich bidirektionale Register verwendet. Zusätzlich wird der Begriff des Registers fortan synonym für Variablen in einem Register verwendet. Wie in Abbildung 2.1 entnommen werden kann, werden die Register mit je $r_{1,2}$ und $r_{2,1}$ bezeichnet wenn sie ausgehend von P_1 und P_2 Nachrichten transportieren. Während die Prozesse sich in der Realität die Register der Nachbarn in lokale Variablen kopieren und von dort aus mit den Werten weiterarbeiten, wird dieser Sachverhalt zukünftig nicht mehr explizit genannt: Bei Nennung eines Registers ist stets die lokale Kopie gemeint.

Prozess Ein *Prozess* ist eine Rechnereinheit innerhalb eines verteilten Systems, der einen Sub-Algorithmus ausführt. Er verfügt über eine eindeutige Identifizierung, ein- und ausgehende Kommunikationsregister und interne lokale Variablen. Im Gegensatz zu den Kommunikationsregistern können die

Nachbarn nicht auf die sich im internen Speicher befindlichen lokalen Variablen des Prozesses zugreifen.

Topologie Die Prozesse eines verteilten Systems sind gemäß einer *Topologie* miteinander verbunden. Analog den Topologien von Rechnernetzen können diese zum Beispiel baumartig, ringförmig oder vermascht sein.

Verteilter Algorithmus Ein verteilter Algorithmus A besteht aus den Sub-Algorithmen aller Prozesse Π, wir schreiben $A := \cup_{P_j \in \Pi} A_{sub_j}$. Sub-Algorithmen sind in *Guarded Command Language* (GCL) annotiert.

Globaler Systemzustand Der lokale Zustand θ_x des Prozesses P_x setzt aus seinen lokalen Variablen und Schreibregistern zusammen. Die Gesamtheit der lokalen Zustände aller Prozesse Π bilden den Vektor $\sigma = [\theta_1, \cdots, \theta_n]$, welcher als *globaler Systemzustand* definiert ist. Die Menge aller möglichen globalen Zustände ist Σ.

Ausführung Eine Ausführung Ξ ist eine Folge von globalen Zuständen, also $\Xi := \langle \sigma_1, \sigma_2, \cdots \rangle$ wobei σ_1 der Startzustand der Folge ist und σ_2 durch einen Rechenschritt des verteilten Systems erreichbar ist. Ein Rechenschritt des verteilten Systems wiederum besteht aus je einem Rechenschritt aller vom Scheduler ausgewählter Prozessoren.

Scheduler Die Ablaufsteuerung, also die Auswahl der Prozesse, die einen Rechenschritt machen dürfen, übernimmt der *Scheduler* $\mathbb{D}$. Da es in jedem Zustand σ_i diverse Möglichkeiten gibt, fortzufahren, lässt sich bei beliebiger Auswahl der Prozesse keine gesicherte Aussage über das Verhalten des Systems machen. Daher werden Scheduler eingesetzt um dem Nicht-Determinismus entgegenzuwirken. Ein Scheduler kann daher als überabzählbare Menge $\mathbb{D} = \{\rho_1, \rho_2, \cdots\}$ dargestellt werden. Scheduler werden in ihrer Auswahl bezüglich Fairness klassifiziert. Wenn ein Prozess unendlich oft aktiviert wurde, sofern der Scheduler unendlich lang lief, so bezeichnet man den Scheduler als unbeschränkt fair (*unconditionally fair*). Sofern zusätzlich gilt, dass ein Prozess, welcher vor einem kritischen Abschnitt wartet, diesen garantiert zu einer späteren Zeit betritt, so zeichnet sich der Scheduler durch schwache Fairness (*weakly fair*) aus. Sollte der Prozess unendlich oft am kritischen Abschnitt warten und auch unendlich oft daraufhin den Zuschlag bekommen, ist der Scheduler sogar dauerhaft fair (*strong fair*).[T10] Das Auswahlverhalten des Schedulers wird auch Auswahlstrategie genannt (*scheduling policy*).

2.1.1 Verteilter Algorithmus

Wie bereits definiert, setzt sich ein verteilter Algorithmus A_x aus den Sub-Algorithmen A_{sub_i} aller Prozesse $P_i \in \Pi$ zusammen. Im folgenden soll nun auf die Sprache, in welcher Sub-Algorithmen beschrieben werden, eingegangen werden: die *Guarded Command Language* (GCL). Entwickelt wurde die Sprache bereits im Jahre 1975 von Dijkstra [Dij75]. Die nachfolgende Definition folgt jedoch [Dha11].
Sub-Algorithmen sind in GCL in einen deklarativen Kopfbereich und dem Bereich der geschützten Anweisungen aufgeteilt. Im deklarativen Kopfbereich werden zunächst lokale Variablen und Konstanten des Prozesses definiert. Darüber hinaus werden Prädikate, sogenannte Makros, definiert, welche im weiteren Code verwendet werden.
Die eigentliche Logik des Sub-Algorithmus im Prozess P_i befindet sich im geschützten Anweisungsbereich. Dort befinden sich mehrere Zuweisungsfunktionen act_{ix}, die je durch eine Bedingung $\mathcal{G}_{ix}$, genannt **Guard**, geschützt sind und optional über einen eindeutigen Namen $label$ verfügen:

$$\langle label \rangle :: \mathcal{G}_{ix} \rightarrow act_{ix}$$

Der Guard $\mathcal{G}_{ix}$ ist hierbei ein Bool'scher Ausdruck, welcher lediglich lokale Variablen und Kommunikationsregistern zur Berechnung referenziert. Sollte der Wahrheitswert eines Guards wahr sein, bezeichnet

man die durch ihn geschützte Anweisung als aktiviert (*enabled*). Wählt der Scheduler den geschützten Befehl des Guards $\mathcal{G}_{ix}$ auf Prozesses P_i aus, so wird die Zuweisungsfunktion act_{ix} ausgeführt. Diese führt Berechnungen durch und weist anschließend ausgehenden Kommunikationsregistern und lokalen Variablen neue Werte zu. Die Begriffe *geschützte Anweisung* und *Guard* werden nunmehr synonym verwendet und die *Bedingung einer geschützten Anweisung* fortan nur noch *Bedingung des Guards* genannt.

κ-lokale Algorithmen

Viele verteilte Algorithmen operieren auf jeweils nur einer Untermenge der Prozesse, um ihre Aufgabe zu erfüllen. Dabei kann für die Entscheidungsfindung eines Prozesses lediglich ein bestimmter Radius in der Topologie vom Bedeutung sein. Dieser Radius umspannt alle Prozesse, welche über maximal κ Knoten (*hops*) zu erreichen ist.

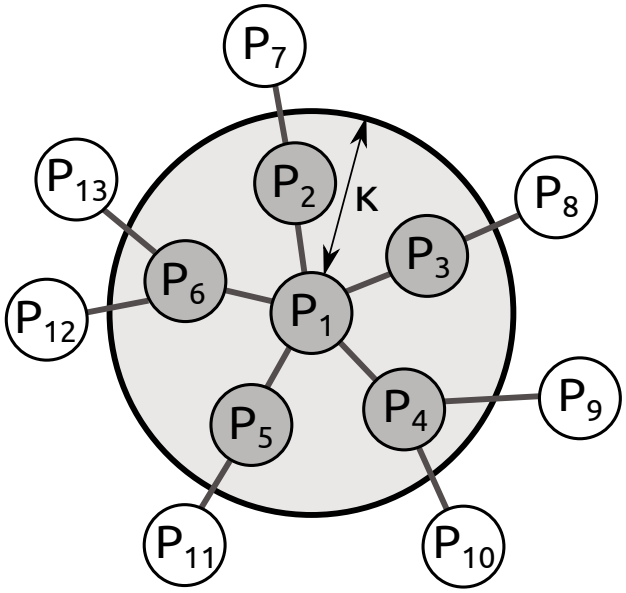


Abbildung 2.2: κ-lokale Nachbarschaft von P_1 mit $\kappa = 1$

Die Prozesse innerhalb dieses Bereich bezeichnet man als κ-lokale Nachbarschaft (*k-hop neighborhood*). In Abbildung 2.2 sehen wir beispielsweise die κ-lokale Nachbarschaft von P_1 mit $\kappa = 1$, welche Prozesse P_2 bis P_6 enthält. Der Sonderfall $\kappa = 1$ umfasst alle direkten Nachbarn, bei größerem κ werden die Prozesse innerhalb der κ-lokalen Nachbarschaft nicht immer direkt mit dem Prozess verbunden sein. Ein Algorithmus, welcher auf einer solchen κ-lokalen Nachbarschaft basiert, nennt sich daher auch κ-lokaler Algorithmus.

Verteiler Algorithmus $\mathbb{SSWMAC}$

Der *slot seeking (algorithm) for wireless medium access control* ($\mathbb{SSWMAC}$) ist ein vereinfachter verteilter Algorithmus zur Suche eines freien Sendeplatzes in einem Funknetzwerk. Diese Algorithmen werden benötigt, da in einem Funknetzwerk nur jeder Teilnehmer zu einer bestimmten Zeit, seinem Sendeplatz (*slot*), Nachrichten verschicken darf. Verschicken zwei oder mehrere Teilnehmer zur gleichen Zeit Nachrichten, so überlagern sich deren Funkwellen und löschen sich gegenseitig aus (destruktive Interferenz). Im folgenden vereinfachten Modell können sich die Teilnehmer durch Kommunikation absprechen, sodass jeder einen eindeutigen Funkplatz kriegt. Es ist jedoch zu beachten, dass die Funkwellen eines Teilnehmers nur eine bestimmte Reichweite κ haben und nur für diesen Bereich Interferenz ausgeschlossen werden muss. Damit handelt es sich bei $\mathbb{SSWMAC}$ um einen κ-lokalen verteilten Algorithmus.

Im folgenden wird $\mathbb{SSWMAC}$ in Listing 2.1 nach [Dha11] beschrieben. Der Sub-Algorithmus beherbergt für jeden Prozess P_i die zwei lokalen Variablen $turn_i$ und $slot_i$, welche für eine Einigung über den Sendeplatz benötigt werden. Während zunächst mit der Variable $turn_i$ ein freier Sendeplatz gesucht wird, zeigt $slot_i$ an, ob der Prozess P_i noch auf der Suche ist ($slot_i = \perp$) oder einen Sendeplatz gefunden hat ($slot_i = turn_i \neq \perp$). Zur Vereinfachung sind im Algorithmus noch mehrere Konstanten definiert. So repräsentiert $\mathcal{N}_i^\kappa$ die κ-lokale Nachbarschaft, die in $\mathcal{N}_i^{\kappa\perp}$ noch weiter auf die suchenden Prozesse innerhalb dieser Nachbarschaft eingeschränkt wird. Die Konstante $\tilde{n}$ ist die maximale Größe der κ-lokalen Nachbarschaft aller Prozesse Π und somit auch die höchste Nummer für einen Sendeplatz, die vergeben werden muss.
Nach den Konstanten folgen die Definitionen der Prädikate, die später in den Bedingungen der Guards abgefragt werden. Das Prädikat min_x überprüft hierbei, ob die Variable $turn_j$ aller suchenden κ-lokalen Nachbarn P_j größer ist. Weiter stellt das Prädikat $unique_x$ sicher, dass die temporäre Wahl eines Sendeplatzes nicht mit einem anderen Prozess kollidiert. Schließlich werden die beiden Prädikate min_x und $unique_x$ konjunkt vereint im Prädikat $valid_x$.

process P_i {
  localvar $turn_i$, $slot_i$;

  const $\mathcal{N}_i^\kappa = \{P_j \in \Pi \backslash P_j \mid dis_{min}(P_i, P_j) \leq \kappa\}$;
  const $\mathcal{N}_i^{\kappa\perp} = \{P_j \in \mathcal{N}_i^\kappa \mid slot_j = \perp\}$;
  const $\tilde{n} = max_{P_a \in \Pi} |\mathcal{N}_a^\kappa|$;

  macro $min_x \equiv \forall P_j \in \mathcal{N}_i^{\kappa\perp} : x \neq turn_j \wedge x < turn_j$;
  macro $unique_x \equiv \forall P_j \in \mathcal{N}_i^\kappa : x \neq turn_j$;
  macro $valid_x \equiv min_x \wedge unique_x$;

  $slot_i = \perp \wedge turn_i \leq \tilde{n} \wedge valid_{turn_i}$
    $\rightarrow slot_i := turn_i$;
  $slot_i \neq \perp \wedge turn_i \leq \tilde{n} \wedge \neg unique_{turn_i}$
    $\rightarrow slot_i := \perp$;
  $slot_i > \tilde{n} \vee turn_i > \tilde{n}$
    $\rightarrow turn_i := turn_i + 1 \pmod{\tilde{n} - 1}; slot_i := \perp;$
  $slot_i = \perp \wedge turn_i \leq \tilde{n} \wedge \neg valid_{turn_i}$
    $\rightarrow turn_i := turn_i + 1 \pmod{\tilde{n} - 1};$
}

Listing 2.1: Sub-Algorithmus $\mathbb{SSWMAC}$ auf Prozess P_i

Der erste Guard überprüft, ob der Prozess P_i einen Sendeplatz sucht, der temporäre Sendeplatz im gültigen Bereich liegt, sowie minimal und eindeutig ist. Wenn dem so ist, wird der temporäre ($turn_i$) als endgültiger Sendeplatz ($slot_i$) gewählt. Damit wäre die Aufgabe des Sub-Algorithmus auf P_i erfüllt. Doch es muss weiterhin sichergestellt werden, dass die Wahl des Sendeplatzes nicht in Zukunft mit dem eines anderen Teilnehmers kollidiert. So prüft der zweite Guard, ob Prozess P_i einen endgültigen Sendeplatz gesetzt hat ($slot_i \neq \perp$), der temporäre Sendeplatz im gültigen Bereich liegt, aber nicht eindeutig ist. In diesem Fall wird der endgültige Sendeplatz zurückgesetzt ($slot_i := \perp$). Sollten die Sendeplätze außerhalb des gültigen Bereichs liegen, stellt der dritte Guard sicher, dass der temporäre Sendeplatz wieder gültig wird durch eine Inkrementierung modulo $\tilde{n} - 1$. Der endgültige Sendeplatz wird wie auch im vorigen Guard zurückgesetzt. Den Suchlauf treibt im eigentlichen Sinne bloß der vierte Guard voran. Er überprüft ob noch kein endgültiger Sendeplatz gefunden wurde, der temporäre Sendeplatz gültig aber nicht eindeutig und minimal ist. Als Konsequenz wird der temporäre Sendeplatz inkrementiert modulo $\tilde{n} - 1$.

2.2 Fehlertolerante Systeme

Dieser Abschnitt befasst sich mit fehlertoleranten Systemen. Zunächst werden diese Systeme definiert (2.2.1), danach anhand eines Beispiels illustriert (2.2.2) und anschließend wird auf die Abläufe zur Sicherstellung der Fehlertoleranz eingegangen.

2.2.1 Definition

Verteilte Systeme, die eine gewisse Robustheit gegenüber Fehlern der Anwendungsausführung aufweisen, werden als fehlertolerante Systeme bezeichnet. Mit Fehlern sind in diesem Kontext flüchtige Fehler (*transient faults*) im Anwendungsspeicher gemeint, die stets nur lokale Variablen betreffen und den Programmcode intakt lassen. Es existieren mehrere Abstufungen der Fehlerresistenz. Erfüllt ein fehlertolerantes System die Eigenschaft der Lebendigkeit (*liveness property*), so ist garantiert, dass es zu einem Zeitpunkt in einen fehlerfreien Zustand wechselt und damit korrekte Berechnungen stattfinden. Ist sogar ausgeschlossen, dass das System sich je in einem fehlerhaften Zustand befindet, ist die Eigenschaft der Sicherheit (*safety*) erfüllt. Ein sicheres System würde terminieren, bevor es in einen fehlerhaften Zustand wechseln würde, während ein lebendiges System zu einem Zeitpunkt nach dem Fehlerfall wieder die korrekte Programmausführung aufnehmen würde.

	sicher	nicht sicher
lebendig	maskierend	nicht maskierend
nicht lebendig	fehlersicher	intolerant

Abbildung 2.3: Klassifikation fehlertoleranter Systeme durch die Eigenschaften der Lebendigkeit und Sicherheit nach [Gär99]

Durch die Eigenschaften der Lebendigkeit und Sicherheit werden fehlertolerante Systeme klassifiziert [Gär99]. Wie auch in der Matrix in Abbildung 2.3 zu sehen ist, nennen sich lebendige Systeme **maskierend** (*masking*) und **nicht maskierend** (*non masking*), sofern sie sicher bzw. nicht sicher sind. Mit Maskierung ist hierbei die Verdeckung von Fehlern während der Programmausführung gemeint. Während nicht maskierende Systeme also nach Fehlern gesunden und die korrekte Ausführung wieder aufnehmen, ist bei maskierenden Systemen sogar zugesichert, dass sich das System zu jeder Zeit in einem sicheren Zustand befindet. Anders verhält es sich bei den sicheren nicht lebendigen Systemen, genannt **fehlersicheren** (*failsafe*) Systemen. Obgleich diese Systeme sich stets in einem guten Zustand befinden, fehlt ihnen die Lebendigkeit. In einem Fehlerfall terminieren sie, um zu verhindern, in einen fehlerhaften Zustand zu wechseln. Sollte ein System weder sicher noch lebendig sein, so ist es kein fehlertolerantes sondern ein intolerantes System.
Zur Realisierung der Fehlertoleranz wird dabei auf Redundanz gesetzt [Jal94]. Als redundant werden Systemkomponenten verstanden, welche nicht gebraucht würden, würde das System nicht fehlertolerant sein. Ohne diese Komponenten würde das System in Abwesenheit von Fehlern korrekt funktionieren. Redundanz wird in drei Kategorien unterteilt:

Physische Redundanz Unter *physischer Redundanz* (*hardware redundancy*) werden zusätzliche Hardware-Komponenten verstanden, welche die Fehlertoleranz unterstützen.

Logische Redundanz Die *logische Redundanz* (*software redundancy*) fasst zusätzlich Logik, welche der Fehlertoleranz zu Gute kommen, zusammen. Simple logische Redundanz ist zum Beispiel die wiederholte Durchführung einer Operation im Fehlerfall.

Zeitliche Redundanz *Zeitliche Redundanz* (*time redundancy*) bezeichnet zusätzliche Zeitfenster, die im System vorgesehen sind um Fehlertoleranz zu unterstützen. Diese Zeit kann für Fehlerbehebungsmaßnahmen (*error recovery*) genutzt werden.

2.2.2 Beispiel Lichtsignalanlage

Am Beispiel einer Lichtsignalanlage im Straßenverkehr sollen die Definitionen nun verdeutlicht werden. Die Aufgabe des Systems besteht in der Koordination von Fahrzeugen bei der Wegequerung an Kreuzungssituationen. Jede an der Kreuzung anliegende Straße verfügt über einen Lichtsignalgeber mit drei Farben, wobei für diese Betrachtung nur die Farbe rot und grün von Bedeutung sein soll, da die verbleibende Farbe orange lediglich die Richtung der Transition beim Farbwechsel indiziert.

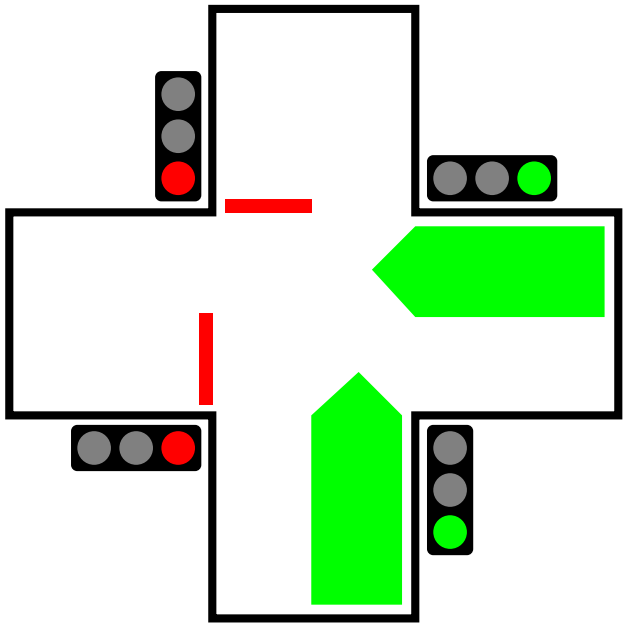


Abbildung 2.4: Fehlerhafter Zustand einer Lichtsignalanlage

Wird den Fahrzeugen einer Zufahrtsstraße grünes Licht gegeben, so dürfen diese die Kreuzung queren, bei roter Farbe hingegen muss vor der Kreuzung gewartet werden. Nehmen wir nun an, es handle sich um ein maskierendes verteiltes System, dann würde die Schaltung nie versagen und stets korrekt in ihrer Spezifikation arbeiten. Wäre das System hingegen nicht maskierend, so könnte es vorkommen, dass das System einige Zeit außerhalb der Spezifikation arbeitet, also zum Beispiel zwei Richtungen grün gegeben werden (siehe Abbildung 2.4). Nach einer gewissen Zeit würde das System zwar wieder korrekt arbeiten, doch vorher kann das System Fehlentscheidungen treffen. Da Fehler für diese Anlage nicht akzeptabel sind und gewisse externe Faktoren wie die Stromversorgung nicht immer garantiert werden kann, würde die Lichtsignalanlage in der Praxis als fehlersicheres System implementiert werden. Wenn zum Beispiel ein Lichtsignalgeber auf grün schalten möchte, während noch einer weiteren Seite grünes Licht gegeben wird (Abbildung 2.4), würde sich die Anlage ausschalten um eine fehlerhafte Anzeige zu vermeiden. Im Straßenverkehr ist diese Situation sogar dahingehend geregelt, als dass beim Ausfall der Lichtsignalanlage die Verkehrszeichen an der Kreuzung Geltung erlangen und die Vorfahrt regeln.

2.2.3 Phasen der Fehlertoleranz

Im weiteren wird auf die vier Phasen der Fehlertoleranz nach [Jal94] eingegangen.
Die erste Phase ist die Fehlererkennung (*error detection*), in der ein fehlerhafter Zustand diagnostiziert wird. Es folgt das Erfassen des Schadenausmaßes (*damage confinement*). Beide Phasen zusammen sind vonnöten, um richtig auf den Fehler zu reagieren. In der Fehlerbehebungsphase (*error recovery*) wird das System wieder in einen korrekten Zustand gebracht. Abschließend wird sichergestellt, dass die Kompo-

nente, welche den Fehler verursacht hat, künftig gemieden wird um eine weitere korrekte Ausführung des Systems zu gewährleisten (*fault treatment and continued system service*).

2.3 Selbststabilisierende Algorithmen

Tritt innerhalb eines selbststabilisierenden Systems ein flüchtiger Fehler auf, so arbeitet es nach einer finiten Anzahl von Schritten wieder korrekt. Mit dieser Definition stellte Dijkstra 1973 das Konzept der Selbststabilisierung [Dij74] vor. Erst später wurde das Konzept als Eigenschaft der Fehlertoleranz klassifiziert [Sch93]. Selbststabilisierende verteilte Algorithmen sind also nach [Gär99] nicht maskierende fehlertolerante Algorithmen.
Wir betrachten das verteilte System als endlichen Automat und teilen die Menge aller Zustände Σ nun in zwei Bereiche auf. So sind alle Zustände $\sigma_x \in \Sigma$, welche das Prädikat $\mathcal{P}$ erfüllen, **legale** Zustände. Das Prädikat $\mathcal{P}$ beschreibt alle Zustände innerhalb der Systemspezifikation. Die verbleibenden Zustände sind jeweils außerhalb der Spezifikation, also fehlerhafte Zustände.
Befindet sich ein selbststabilisierendes System nicht in einem legalen Zustand, ist garantiert, dass es nach einer endlichen Anzahl von Schritten wieder entsprechend in einen legalen Zustand wechselt. Für jeden selbststabilisierenden Algorithmus ist dieses Verhalten gezeigt. Die Beweisführung dieses Strebens zu einem legalen Zustand geschieht durch eine Bewertungsfunktion.

2.3.1 Prädikat $\mathcal{P}$

Das Prädikat $\mathcal{P}_A$ (*safety predicate*) stellt fest, ob sich ein selbststabilisierendes System unter dem Algorithmus A in einem legalen Zustand befindet, also innerhalb der Spezifikation. Sollte dies nicht der Fall sein, so befindet sich das System in einem fehlerhaften Zustand. Die Eigenschaft der Selbststabilisierung garantiert nun, dass das System sich nach endlich vielen Schritten wieder in einem legalen Zustand befindet, also das Prädikat $\mathcal{P}_A$ erfüllt:

$$< \sigma_1, \sigma_2, \cdots, \sigma_n > \text{ mit } \sigma_n \models \mathcal{P}_A$$

Das Prädikat wird lokal von den Prozessen überprüft und eventuell Maßnahmen getroffen um die Konvergenz wiederherzustellen. Wenn man das Prädikat für jeden Prozess betrachtet, kann darüber hinaus festgestellt werden, ob sich das gesamte System in einem korrekten Zustand befindet:

$$\wedge_{P_x \in \Pi} \mathcal{P}_A(P_x)$$

Für den Algorithmus $\mathbb{SSWMAC}$ beispielsweise ist das Prädikat für Prozess P_i wie folgt definiert:

$$\mathcal{P}_{\mathbb{SSWMAC}}(P_i) \equiv (turn_i = slot_i) \wedge (\forall j \in \mathcal{N}_i^{\kappa} : slot_j \neq slot_i)$$

Das Prädikat stellt sicher, dass der temporäre mit dem permanenten Sendeplatz übereinstimmt und kollisionsfrei ist. Obwohl selbststabilisierende Systeme nie terminieren, kann mit dem Prädikat $\mathcal{P}_A$ überprüft werden ob der Algorithmus seine Aufgabe im Moment erfüllt hat.

2.3.2 Bewertungsfunktion

Die Eigenschaft der Selbststabilisierung eines verteilten Algorithmus wird über eine Bewertungsfunktion nachgewiesen [DOT06]. Diese zeigt, dass der Algorithmus von einem fehlerhaften Zustand nach endlich vielen Schritten wieder in einen legalen Zustand eintritt. Das Streben des Algorithmus in den Bereich der

legalen Zustände wird analog der Analysis als Konvergenz bezeichnet. Auch die Beweisführung erfolgt ähnlich, sodass eine strenge Monotonie der Zustandsübergänge nachgewiesen werden muss.

Um dieses Verhalten zu zeigen, muss eine Ordnungsrelation definiert sein. Für eine Ordnungsrelation wiederum müssen die Zustände vergleichbar sein. Daher werden die fehlerhaften Zustände nach verschiedenen Kriterien mittels einer Funktion bewertet ($\Delta : \Sigma \mapsto \Theta$) und somit jedem Zustand ein Wert zugewiesen. Anschließend untersucht man die Differenz eines Zustands mit seinem Folgezustand. Sollte hier für jeden fehlerhaften Zustand und seinem Folgezustand eine fallende Tendenz vorliegen ($\Delta(\sigma_a) > \Delta(\sigma_b) \forall \sigma_a, \sigma_b \in \Sigma$ mit σ_a folgt direkt σ_b), so ist eine streng fallende Monotonie und somit auch die Konvergenz gezeigt.

Für jeden selbststabilisierenden Algorithmus existiert eine solche Bewertungsfunktion. Da die Auswahl der Folgezustände durch den Scheduler des verteilten Algorithmus bestimmt wird, garantiert dieser direkt die Eigenschaft der Selbststabilisierung. Umgekehrt lässt sich feststellen, dass der Scheduler seine Entscheidungen im Einklang mit der Bewertungsfunktion fällt. Diese Eigenschaft ist der Transformation zur Komposition mehrerer Algorithmen von zentraler Bedeutung ($\rightarrow$ 3).

Betrachten wir nun konkret die Bewertungsfunktion für den Algorithmus $\mathbb{SSWMAC}$ nach [Dha11]:

$$\Delta_{WMAC}(\sigma_x) := \langle inconsistent_x, conflict_x, n - correct_x, n - valid_x, seek_x, distance_x \rangle$$

Unter dem Scheduler $\mathbb{D}_{WMAC}$ konvergiert der Algorithmus, sodass sich der Vektor der Bewertungsfunktion stetig vermindert. Die erste Dimension ist $inconsistent_x$, welche die Anzahl der inkonsistenten Prozesse zählt:

$$inconsistent_x \equiv |\{P_i \in \mathcal{N}_i^\kappa \mid slot_i \neq \perp \wedge slot_i \neq turn_i\}|$$

Danach werden die in Konflikt stehenden Prozesse betrachtet, welche jeweils einen gleichen Sendeplatz beanspruchen:

$$conflict_x \equiv |\{P_a, P_b \in \mathcal{N}_i^\kappa \mid P_a \neq P_b \wedge slot_a = slot_b \neq \perp\}|$$

Anschließend folgt invertiert die Anzahl der Prozesse im legalen Zustand:

$$correct_x \equiv |\{P_i \in \mathcal{N}_i^\kappa \mid \forall P_j \in \mathcal{N}_i^\kappa \backslash P_i : slot_i \neq slot_j\}|$$

Hiernach werden Prozesse invertiert gezählt, welche sich in einem korrekten Suchzustand befinden:

$$valid_x \equiv |\{P_i \in \mathcal{N}_i^\kappa \mid slot_i = \perp \ \wedge \forall P_j \in \mathcal{N}_i^\kappa \backslash P_i : slot_i \neq slot_j \wedge (slot_j \neq \perp \vee turn_j > turn_i)\}|$$

Als nächstes Merkmal zur Konvergenz wird die Anzahl der suchenden Prozesse herangezogen:

$$seek_x \equiv |\{P_i \in \mathcal{N}_i^\kappa \mid slot_i = \perp \ \wedge \exists P_j \in \mathcal{N}_i^\kappa \backslash P_i : slot_i = slot_j \vee (slot_j = \perp \ \wedge turn_j \leq turn_i)\}|$$

Abschließend findet der Abstand zum nächsten freien temporären Sendeplatz Beachtung. Die Menge aller freien temporären Sendeplätze sei zunächst:

$$free := \{i \in \mathbb{N} | 0 \leq i < \mathcal{N}_k \wedge \forall turn_j \in \mathcal{N}_i^\kappa : turn_j \neq i\}$$
$$seeking_x \equiv slot_x = \perp \wedge turn_x \leq \mathcal{N}_k \wedge \ \exists P_j \in \mathcal{N}_i^\kappa \backslash P_x : turn_x = turn_j$$

Die Differenz zum nächsten minimalen Sendeplatz lautet dann:

$$distance_x \equiv \begin{cases} min_{i \in free}(i - turn_x \pmod{\tilde{n}}) & \text{wenn } seeking_x \\ 0 & \text{sonst} \end{cases}$$

2.3.3 Beispiel: balance-unbalance protocol

Betrachten wir im folgenden nun den simplen selbststabilisierenden Algorithmus *balance-unbalance protocol* nach [DIM93], welcher gegenseitigen Ausschluss in einem verteilten System mit zwei Prozessoren (siehe Abbildung 2.1) gewährleisten soll. Die Kommunikationsregister können in diesem System lediglich die Werte 0 und 1 annehmen. P_1 sei der ausgewogene (*balanced*) Prozess, welcher nur dann seinen kritischen Abschnitt ausführt, wenn die Register $r_{1,2}$ und $r_{2,1}$ den gleichen Wert beinhalten. Entsprechend sei P_2 unausgewogen (*unbalanced*) und würde bei Ungleichheit der Register seinen kritischen Abschnitt durchlaufen. Nachdem einer der Prozessoren eine Berechnung durchgeführt hat, würde er den Wert seines Registers invertieren.
Es ist leicht zu sehen, dass diese Vorgehensweise stets gegenseitigen Ausschluss garantiert. Durch den eingeschränkten Wertebereich der Register existieren insgesamt nur vier Möglichkeiten der Wertebelegung jener Register: $(0, 0)$, $(0, 1)$, $(1, 0)$ und $(1, 1)$. Für jede dieser Belegung gilt die Eigenschaft der Gleichheit oder Ungleichheit, sodass nur genau ein Prozess zum Zuge kommt, da seine Ausführungsbedingung wahr ist. Außerdem wird durch das anschließende Invertieren des ausgehenden Kommunikationsregisters des Prozesses das vorige (Un)gleichgewicht umgekehrt und damit garantiert, dass der andere Prozess eine Berechnung durchführen darf.

3 Transformation selbststabilisierender Algorithmen

Für den Entwurf von komplexen verteilten Systemen werden kleinere verteilte Algorithmen kombiniert. Jeder Algorithmus löst ein Teilproblem und in ihrer Gesamtheit können die Algorithmen ein größeres Problem lösen. Für fehlertolerante, insbesondere selbststabilisierende Systeme muss jedoch sichergestellt werden, dass nicht nur jeder Teilalgorithmus sondern auch das Gesamtsystem die gewünschten Eigenschaften der Fehlertoleranz erfüllen. Im folgenden wird nach [DT10] beschrieben, wie die Teilalgorithmen zunächst transformiert werden, damit sie unabhängig vom Scheduler konvergieren. Danach wird aufgezeigt, wie sich die transformierten Algorithmen in einer Kompositionen zusammenführen lassen.

3.1 Scheduler-Unabhängigkeit

Für jeden selbststabilisierenden Algorithmus ist durch seine Bewertungsfunktion die Selbststabilisierung bewiesen. Der Scheduler ist hierbei eine direkte Abhängigkeit der Konvergenz. Bei der Komposition mehrerer selbststabilisierender Algorithmen tritt nun das Problem auf, dass die Algorithmen nur ihren eigenen Scheduler konvergieren, nicht jedoch unter einem anderen Scheduler. Somit können die Algorithmen nicht einfach alternierend kombiniert und der gewünschte Scheduler von nur einem Algorithmus gewählt werden. Das Gesamtsystem würde nicht mehr konvergieren.

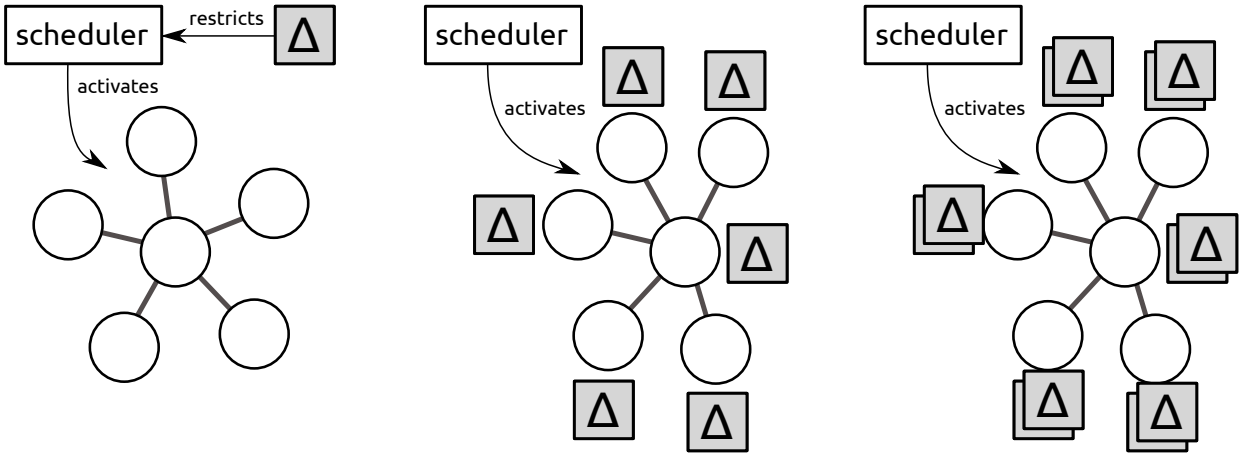


Abbildung 3.1: Zustände der Transformation und Komposition selbststabilisierender Algorithmen

Dieses Dilemma kann umgangen werden, wenn die Algorithmen so modifiziert werden können, dass sie unter demselben Scheduler konvergieren. Für diese Transformation muss also die Wirkungsweise des Schedulers erhalten bleiben, ohne dass dieser selbst im Einsatz wäre. In 2.3.2 haben wir bereits festgestellt, dass der Scheduler seine Entscheidungen stets im Einklang zur Bewertungsfunktion fällt. Genau diese Eigenschaft kann nun ausgenutzt werden, indem jeder Prozess zunächst mithilfe der Bewertungsfunktion sein Handeln überprüft (*progress monitor*) [DT10]. Dieser Transformationsschritt kann in Abbildung 3.1 als Übergang vom System links zum System in der Mitte nachvollzogen werden. Das Sicherstellen des Konvergenzstrebens wird somit von der globalen Position des Scheduler in die lokale Ebene des Sub-Algorithmus jedes Prozesses verlagert. Mehrere transformierte Algorithmen können danach als Komposition verwendet werden wie in Abbildung 3.1 rechts zu sehen ist. Die Konvergenz der einzelnen Algorithmen wird lokal

über die Bewertungsfunktion sichergestellt und das gesamte System konvergiert unter jedem beschränkt fairen Scheduler.

3.2 Verifikation mittels Bewertungsfunktion

Zur Erhaltung der Eigenschaft der Selbststabilisierung muss jeder Sub-Algorithmus eines Prozess selbst sicherstellen, dass seine Entscheidung zur Konvergenz beiträgt. Dieses geschieht durch Betrachtung der Bewertung des aktuellen Zustandes und aller aktivierten Folgezustände (2.3.2). Falls ein Folgezustand niedriger bewertet wird als der aktuelle Zustand, so wird er ausgewählt.
Damit die Bewertungsfunktion jedoch einen Wert berechnen kann, benötigt sie Kenntnis über den Zustand des gesamten Systems und nicht nur des lokalen Prozesses. Der Zustand des Gesamtsystems lässt sich nur ermitteln, wenn die Prozesse untereinander systematisch ihren Zustand kommunizieren. Allerdings kann der Zustand des verteilten Systems sich bereits verändert haben, wenn die Information den Prozessor erreicht. Um fehlerhafte Kenntnis des Zustand des Gesamtsystems auszuschließen, benötigen wir gegenseitigen Ausschluss.[DT10] In den folgenden Abschnitten wird auf Hilfsalgorithmen für den gegenseitigen Ausschluss eingegangen (3.2.1).
Die Überprüfung der sinkenden Bewertungsfunktion wird in der Bedingung jedes Guards hinzugefügt:

$$\grave{\mathcal{G}}_{ix} :: \mathcal{G}_{ix} \wedge (\mathcal{P}_A \vee \Delta_A(\sigma_i) > \Delta_A(\sigma_{i+1}))$$

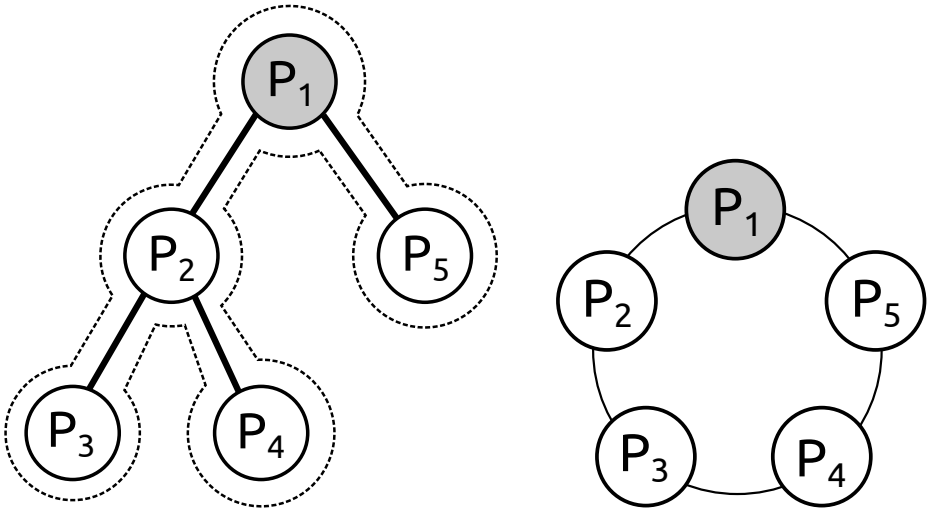
Es ist hierbei zu beachten, dass eine Verifikation des Konvergenzstrebens nur genau dann geschieht, wenn noch keine Konvergenz vorliegt ($\neg\mathcal{P}_A$). Sonst kann beliebig fortgefahren werden, da sich das System bereits in einem legalen Zustand befindet.

3.2.1 Gegenseitiger Ausschluss

Zur Realisierung des gegenseitigen Ausschlusses wird nun ein selbststabilisierender Algorithmus benötigt. Hierfür existieren mehrere Algorithmen für lokalen und gegenseitigen Ausschluss. Die Algorithmen für den globalen gegenseitigen Ausschluss erfordern ihrerseits eine Ring-Topologie. Eine solche Ring-Topologie kann durch Verwendung eines weiteren selbststabilisierenden Algorithmus generiert werden. Konkret wird dabei eine aufspannende Baumstruktur gebildet, welche dann über den Eulerpfad als virtueller Ring betrachtet wird.
Algorithmen für den lokalen gegenseitigen Ausschluss benötigen hingegen keine Baumstruktur. Sie nutzen die bereits vorhandene Topologie und realisieren darauf mit Uhren lokalen gegenseitigen Ausschluss. Die Nutzung des lokalen gegenseitigen Ausschlusses ist besonders für κ-lokale Algorithmen interessant, da sie sichere Nebenläufigkeit ermöglichen und damit möglicherweise auch die Systemleistung steigern.
Der gegenseitige Ausschluss transportiert nicht nur die Information, welcher Prozess privilegiert ist, sondern auch den Status des gesamten verteilten System. Mit diesem Schnappschuss ist der privilegierte Prozess in der Lage seine Bewertungsfunktion zur Überprüfung des Konvergenzstrebens zu verwenden. Beim globalen gegenseitigen Ausschluss ist der Systemstatus an das Privileg gebunden, im lokalen gegenseitigen Ausschluss findet eine dezentrale Verteilung des Schnappschusses durch alle Prozesse statt. Zur weiteren Vereinfachung wird jedoch im folgenden nicht explizit Schnappschuss eingegangen.
Nachfolgend werden zwei Algorithmen für den gegenseitigen Ausschluss nach [Dol00] und [BP08] beschrieben. Des Weiteren wird der selbststabilisierende Algorithmus für die Erzeugung einer Baumstruktur nach [AKY97] erläutert.

Globaler gegenseitiger Ausschluss

Betrachten wir zunächst den selbststabilisierenden Algorithmus zum globalen gegenseitigen Ausschluss nach [Dol00]. Wie bereits erwähnt, fordert dieser Algorithmus, dass das verteilte System ringförmig strukturiert ist. Vereinfacht dargestellt gibt es einen Zustand der Privilegierung, im folgenden **Token** genannt, welcher nur ein Prozess des verteilten Systems innehaben kann. Wird ein Prozess durch den Scheduler ausgewählt, so prüft er, ob er im Besitz des Tokens ist und fährt nur dann fort, wenn der Prozess es besitzt. Das Token wird nach Ausführung des privilegierten Zustands weitergereicht an den nächsten Prozess im Ring.



(a) Gestrichelte Eulertour des Tokens (b) Virtueller Ring der Eulertour

Abbildung 3.2: Route des Tokens zur Privilegierung eines Prozesses

Die simple Darstellung weicht insofern von der Realität ab, als dass der Algorithmus auf einer Baum-Topologie basiert, die dann ringförmig durchlaufen wird. In Abbildung 3.2 ist links beispielhaft ein verteiltes System in einer Baum-Struktur gezeigt, wie es durch einen Algorithmus für der aufspannenden Baum generiert wird. Der Wurzelprozess ist hierbei grau markiert. Gestrichelt gezeichnet ist hierbei die Eulertour des Tokens. Es ist hierbei besonders zu beachten, dass Token von einem Prozess, der bereits innerhalb einer Runde bereits privilegiert war, lediglich weitergereicht wird. Die tatsächliche Privilegierungsreihenfolge entspricht daher dem virtuellen Ring in der rechten Grafik von Abbildung 3.2.
Da eine Baumstruktur vorliegt, gibt es zwei Typen von Prozessen, den Wurzelknoten und Blätter. Somit wird auch der Privilegierungszustand für den Wurzelknoten der Baumstruktur anders definiert als für die normalen Prozesse. Wir gehen davon aus, dass die Nachbarn eines Prozesses P_i der festen Anordnung λ_i folgen, welche bis zu einer Veränderung der Topologie an dem Prozess besteht.
Wie Listing 3.1 entnommen werden kann, hat jeder Prozess P_i ein Kommunikationsregister $token_i \in \mathbb{N}$. Das Prädikat $havetoken_x$ ist eine Disjunktion und unterscheidet zwischen den Prozesstypen. Der Wurzelprozess P_1 darf seinen kritischen Abschnitt genau dann betreten, sofern der letzte Nachbar P_x nach Anordnung λ_1 im Kommunikationsregister $token_x$ den gleichen Wert vorhält wie $token_1$.
Normale Prozesse P_i hingegen sind privilegiert, wenn ihr Register $token_i$ nicht mit dem Register $token_{parent_i}$ ihres Vaterknotens übereinstimmt. Falls der Prozess P_x tatsächlich privilegiert ist, führt es den Anwendungsalgorithmus aus, welcher durch den gegenseitigen Ausschluss geschützt ist. Anschließend reicht es das Privileg weiter, indem es den eigenen Wert von $token_x$ auf den Wert des Vorgängers setzt. Der Vorgänger ist für den Wurzelknoten der letzte Kindknoten und für die normalen Prozesse der Vaterknoten.

```
process P_x
{
    localvar token_x;
    const λ ≡ ⟨parent_x, n_1, ··· , n_m⟩ mit {n_1, ··· , n_m} = {P_i ∈ N_x \ parent_x | parent_i = id_x};

    macro havetoken_x ≡ (token_x ≠ token_{parent_x} ∧ parent_x ≠ id_x ≠ root_x)
                        ∨(token_x = token_ρ ∧ parent_x = root_x = id_x)


    if (havetoken_x)
    {
        ...

        if (parent_x = root_x = id_x)
            token_x := (token_{λ_|λ|} + 1) (mod 4n − 5);
        else
            token_x := token_{parent_x};
    }
}
```

Listing 3.1: Sub-Algorithmus für den globalen gegenseitigen Ausschluss an Prozess P_x

Das Kommunikationsregister $token_x$ ist speziell und wird nicht an jeden Nachbarprozess mit gleichem Inhalt versandt. Es ermöglicht durch getrennte Behandlung jedes Nachbarprozesses das permanente Weiterreichen des Tokens entlang der Eulertour wie sie beispielhaft auch in Abbildung 3.2 zu sehen ist. Wie man der Logik für die Zirkulation des Tokens in Listing 3.2 entnehmen kann, geschieht das Weiterreichen durch Betrachtung der Anordnung λ, in welcher jeder Prozess die Nachbarprozesse anordnet, die in der Baumstruktur Vater oder Kind von Prozess P_x sind. Jeder Prozess erhält somit das Token des Vorgängers in der Anordnung. Ein Spezialfall ist hierbei, wenn der Prozess das erste Kind oder der eigene Prozess der Wurzelknoten mit nur einem Kind ist. In diesem Fall wird dem anderen Prozess das eigene Token gesendet.

```
∀P_i ∈ N_x :
    if λ_2 = P_i ∨ (λ_1 = P_i ∧ |λ| = 1)
        token_{r_{x,i}} := token_x;
    else
        token_{r_{x,i}} := token_ℓ mit j = ℓ + 1 (mod |λ|) und λ_j = P_i;
```

Listing 3.2: Logik zur Zirkulation des Privilegs

Da das Token mit jedem Schritt nur einen Prozess weiterwandern kann, kommt es bei der Rücktour des Tokens zu Schritten, in welchen kein Prozess privilegiert ist. Wenn zum Beispiel Prozess P_4 aus Abbildung 3.2 nach seiner Privilegierung das Token weiterreicht, so gibt er es zunächst an Prozess P_2 weiter. Danach wird es von Prozess P_2 an Prozess P_1 weitergegeben und erst danach an Prozess P_5. Nachdem Prozess P_4 privilegiert war, und bevor das Token bei Prozess P_5 eintrifft, ist kein Prozess privilegiert.
Diese Rechenschritte, die nur zur Weiterleitung des Tokens dienen und keine direkte Privilegierung zur Folge haben, kommen durch die Abbildung der Baum- auf die Ringstruktur zustande. Selbst wenn die Systemtopologie ringförmig ist, würde eine Baumstruktur gebildet werden. Die Eulertour über die Struktur würde $n - 1$ Rechenschritte für die Rückleitung des Tokens benötigen.

Lokaler gegenseitiger Ausschluss

Während der globale gegenseitige Ausschluss die korrekte Funktionsweise jedes Algorithmusses durch einen korrekten globalen Systemzustand absichert, benötigt nicht jeder Algorithmus Informationen über den Zustand aller Prozesse.
So ist es bei κ-lokalen Algorithmen nur nötig, einen gegenseitigen Ausschluss innerhalb der κ-lokalen Nachbarschaft sicherzustellen. Abbildung 3.3 zeigt beispielsweise, wie in einem verteilten System ein κ-lokaler gegenseitiger Ausschluss mit $\kappa = 1$ die Prozesse P_1, P_{14}, P_{16}, P_{17}, P_{19}, P_{21}, P_{22} und P_{24} privilegiert sind. Im Gegensatz zum globalen gegenseitigen Ausschluss können hier mehrere Prozesse ihren κ-lokalen Algorithmus ausführen.

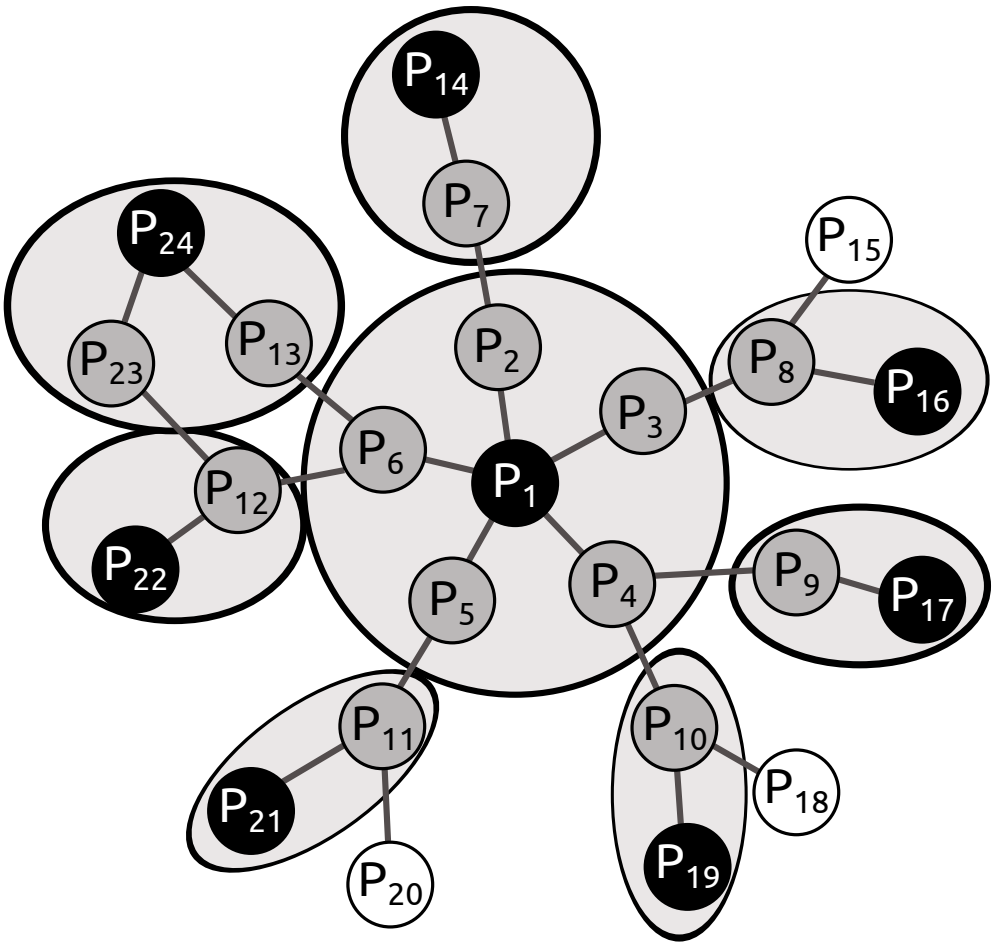


Abbildung 3.3: Zustand eines verteilten Systems unter κ-lokalen gegenseitigen Ausschluss

Im folgenden soll die Vorgehensweise des Algorithmus [BP08] nach [Dha11], welche in Listing 3.3 zu sehen ist, erläutert werden.
Jeder Prozess P_i verfügt hierbei über die Kommunikationsregister $clock_{i1}$ und $clock_{i2}$ sowie res_{i1} und res_{i2}. Während die Synchronisation über das Register $clock_{i1}$ geschieht, wird der gegenseitige Ausschluss durch $clock_{i2}$ geregelt. Die Prozesse betrachten hierbei ihre Nachbarn und wählen den Prozess mit dem niedrigsten Variablenwert $clock_{i2}$ und speichern diesen entsprechend in res_{i2}. Wird ein Prozess von allen Nachbarn vorgeschlagen, so darf er den kritischen Abschnitt betreten.
Konkret existieren mehrere Makros, welche als Guards verwendet werden. Das Prädikat $normalstep_{ix}$ prüft ob die Variable $clock_x$ des Prozesses P_i positiv ist und gleichauf oder um Eins hinter den Variablen $clock_j$ Nachbarn P_j liegt. Es folgt das Makro $locallycorrect_{ix}$, welches ebenfalls die Variable $clock_x$

des Prozesses P_i auf positiven Wert prüft und darüber hinaus sicherstellt, dass alle Variablen $clock_j$ des Nachbarn P_j den gleichen Wert beinhalten oder maximal um Eins positiv oder negativ abweichen. Mit dem Prädikat $reset_{ix}$ wird ermittelt, ob die Variable $clock_x$ des Prozesses P_i positiv ist und weiterhin nicht das Prädikat $locallycorrect_{ix}$ erfüllt, also eine größere Abweichung als Eins aufweist. Das Makro $convergestep_{ix}$ ist wahr, wenn der Wert der Variable $clock_x$ des Prozesses P_i negativ ist und die Variablen $clock_j$ der Nachbarn P_j ebenfalls negativ und sogar größer oder gleich mit $clock_x$ sind.

```
process P_i
{
    localvar clock_i1, clock_i2;
    localvar res_i1, res_i2;

    macro normalstep_ix ≡ 0 ≤ clock_ix < K_x ∧ (∀P_j ∈ N_i : clock_ix = clock_jx
                                                ∨clock_jx = clock_ix + 1 (mod K_x));
    macro locallycorrect_ix ≡ 0 ≤ clock_ix < K_x ∧ (∀P_j ∈ N_i : (clock_ix = clock_jx
                                                ∨clock_jx = clock_ix + 1 (mod K_x)
                                                ∨clock_ix = clock_jx + 1 (mod K_x))
                                                ∧0 ≤ clock_jx < K_x);
    macro reset_ix ≡ ¬locallycorrect_ix ∧ ¬(α_x ≤ clock_ix ≤ 0);
    macro convergestep_ix ≡ α_x ≤ clock_ix < 0
                                ∧(∀P_j ∈ N_i : α_x ≤ clock_jx ≤ 0 ∧ clock_ix ≤ clock_jx);

    nextstep_i
        → if clock_i1 = κ (mod κ + 1) {
              if normalstep_i2 ∧ ⟨clock_i2, P_i⟩ = res_i2 {
                  ...
                  clock_i2 := clock_i2 + 1 (mod K_2);
              }
              res_i1 := res_i2 := ⟨clock_i2, P_i⟩;
          } else {
              res_i1 := res_i2;
              res_i2 := ⟨clock_j2, P_j⟩ mit P_j hat niedrigste clock_j2 und ID;
          }
          clock_i1 := clock_i1 + 1 (mod K_1);
    ∀x ∈ {1, 2} : convergestep_ix
        → clock_ix := clock_ix + 1 (mod K_x);
    ∀x ∈ {1, 2} : reset_ix
        → clock_ix := −α_x;
}
```

Listing 3.3: Subalgorithmus für den κ-lokalen gegenseitigen Ausschluss nach [Dha11]

Der erste Guard prüft, ob der Prozess P_i das Prädikat $nextstep_i$ erfüllt. Sollte dies der Fall sein, so wird weiterhin geprüft, ob die Variable $clock_{i1}$ gleich $\kappa \pmod{\kappa + 1}$ ist. Wenn dann auch noch das Prädikat $normalstep_{ix}$ wahr ist und die Variable res_{i2} auf den eigenen Prozess zeigt, hat der Prozess P_i das Privileg und darf den Anwendungsalgorithmus ausführen. Anschließend inkrementiert er die Variable $clock_{i2}$ modulo K_2. Unabhängig davon, ob der Anwendungsalgorithmus ausgeführt wurde, werden die Auswahlvektoren auf den eigenen Prozess P_i gezeigt. Falls in dieser Runde noch keine Ausführung erfolgen darf ($clock_{i1} \neq \kappa \pmod{\kappa + 1}$), wird der nächste zu privilegierende Prozess innerhalb der Nachbarschaft gesucht. In diesem Auswahlverfahren werden alle Prozesse P_j mit niedriger $clock_{j1}$ gesucht und davon der

Prozess P_j mit kleinster Identifikationsnummer gewählt. Der Vorschlag wird entsprechend in res_{i2} gespeichert. Am Ende des ersten Guards wird abschließend $clock_{i1}$ zur Synchronisation inkrementiert modulo K_1.
Es folgen für jede $clock_i x$ des Prozesses P_i noch zwei weitere Guards, die jeweils die $clock_i x$ zurücksetzen ($reset_{ix}$) und anschließend wieder in den gültigen Wertebereich versetzen ($convergestep_{ix}$).

3.2.2 Aufspannender Baum

Für den globalen gegenseitigen Ausschluss wird eine Baumstruktur benötigt. Diese wird durch den selbststabilisierenden Algorithmus zur Erzeugung eines aufspannenden Baums nach [AKY97] geschaffen. Vereinfacht gesagt baut jeder Prozess mit seinen Nachbarn eine Baumstruktur auf. Dabei sucht hierbei jeder Prozess in seiner Nachbarschaft einen Prozess mit einer höheren Identifikationsnummer (nachfolgend ID genannt) und schließt sich dieser Baumstruktur an. Um sich als Blatt im Baum einzufügen, stellt der Prozess eine Beitrittsanfrage (*join request*), welche dann über die Astknoten weitergegeben wird an den Wurzelknoten (*root*). Dieser wiederum antwortet mit einer Beitrittserlaubnis (*grant response*). Nach einer bestimmten Zeit, sind alle Baumstrukturen vereinigt und es ist ein großer aufspannender Baum entstanden.

Detaillierte Beschreibung

Im folgenden wird der selbststabilisierende Algorithmus zur Generierung eines aufspannenden Baums in einem verteilten System mit eindeutig benannten Prozessen, wie er in Listing 3.4 zu sehen ist, ausführlich nach [AKY97] beschrieben.
Jeder Prozess P_x beherbergt mehrere Kommunikationsregister um den Algorithmus ausführen zu können, welche in drei Gruppen klassifiziert werden.

- Standardvariablen, welche durch tiefer liegende Algorithmen gegeben sind: id_x und $\mathcal{N}_x$

  id_x beinhaltet den eindeutigen Prozessnamen

  $\mathcal{N}_x$ ist eine Liste der Namen aller Nachbar-Prozesse

- Variablen, welche den Status in der Baumstruktur beschreiben: $root_x$, $parent_x$ und $distance_x$

  $root_x$ enthält den Namen des Prozesses, welcher der Wurzelknoten der Baumstruktur ist, in der sich der Prozess P_x befindet

  $parent_x$ zeigt auf den Prozess, welcher Vaterknoten des Prozesses P_x ist

  $distance_x$ speichert die Anzahl der Knotenverbindungen, also den Abstand, zum Wurzelknoten

- Statusregister, die den Zustand von Beitrittsanfragen und -erlaubnissen anzeigen: $request_x$, $from_x$, to_x und $direction_x$

  $request_x$ zeigt den Knoten an, der um Beitritt bittet

  $from_x$ enthält den Namen des Prozess, von welchem diese Nachricht weitergeleitet wurde

  to_x gibt an, durch welchen Nachbarprozess von P_x die Nachricht weitergereicht werden soll

  $direction_x$ kategorisiert die Nachricht: Bei einem Wert von Ask handelt es sich um eine Beitrittsanfrage von $request_x$, andernfalls bei Grant um eine Beitrittserlaubnis

Die Prozesse des verteilten Systems überprüfen anhand mehrerer Prädikate den Zustand der Baumstruktur und treffen dann Entscheidungen über das weitere Vorgehen. Dabei wird das Ziel im Auge behalten, dass jeder Prozess das Prädikat st erfüllt. Ein einzelner Prozess kann anhand des Prädikats überprüfen ob er sich in einer korrekten Baumstruktur befindet. Gilt das Prädikat für alle Prozesse, so befindet sich das gesamte System in einer korrekten Baumstruktur.

Das Prädikat prüft zunächst die Konsistenz des Prozesses in der Baumstruktur. Entweder der Prozess P_x ist Wurzelknoten ($root_x = id_x$) und hat sich daher auch selbst als Vaterknoten ($parent_x = id_x$) und eine Distanz von Null zur Wurzel ($distance_x = 0$) oder ist Blattknoten und erfüllt die Bedingungen, dass der Wurzelknoten eine größere Kennung als P_x ($root_x > id_x$) hat, der Vaterknoten in der Nachbarschaft ist ($parent_x \in \mathcal{N}_x$) und den gleichen Wurzelknoten hat ($root_x = root_{parent_x}$) sowie entsprechend eine um Eins verminderte Distanz zur Wurzel hat ($distance_x = distance_{parent_x} + 1$). Es ist hierbei zu beachten, dass das Prädikat st_x stets nur eine Aussage der Korrektheit über den Prozess P_x macht und nicht über seine Nachbarn, obgleich es für diese Überprüfung Register der Nachbarn abfragt. So wäre es denkbar, dass der Prozess - wenn er Wurzelknoten ist - überprüft, ob die Nachbarprozesse seine Kinder sind. Dies geschieht aber ausdrücklich nicht, denn das Prädikat gibt lediglich die lokale Konsistenz wieder.

process P_x
{
  localvar $root_x$, $parent_x$, $distance_x$;
  localvar to_x, $request_x$, $from_x$, to_x, $direction_x$;

  const id_x;
  const $\mathcal{N}_x$;

  macro $frst_x$;
  macro st_x;
  macro $rqst_x$;
  macro $rqst'_x$;

  $\neg frst_x$
    $\rightarrow root_x := parent_x := id_x;\ distance_x = 0;$
  $frst_x \wedge [\exists u \in \mathcal{N}_x | (root_u = max_{i \in \mathcal{N}_x} root_i) > root_x = id_x] \wedge \neg AlreadyAsking_x$
    $\rightarrow request_x := from_x = id_x; to_x = id_u; direction_x = \mathsf{Ask};$
  $st_x \wedge \neg rqst'_x$
    $\rightarrow request_x := from_x := to_x := direction_x := \perp;$
  $st_x \wedge rqst'_x \wedge \neg rqst_x \wedge (\exists w \in \mathcal{N}_x | direction_w = \mathsf{Ask} \wedge to_w = id_x$
    $\wedge request_w = id_w = root_w = from_w \wedge from_{parent_x} \neq id_x)$
    $\rightarrow request_x = from_x = id_w; to_x = parent_x; direction_x = \mathsf{Ask};$
  $st_x \wedge rqst'_x \wedge \neg rqst_x \wedge (\exists w \in \mathcal{N}_x | direction_w = \mathsf{Ask} \wedge to_w = id_x$
    $\wedge \perp \neq request_w \neq id_w \wedge parent_w = id_x \wedge from_{parent_x} \neq id_x)$
    $\rightarrow request_x := request_w; from_x := id_w; to_x := parent_x; direction_x = \mathsf{Ask};$
  $st_x \wedge rqst_x \wedge (P_x$ ist Wurzelknoten $) \wedge direction_x = \mathsf{Ask}$
    $\rightarrow direction_x = \mathsf{Grant};$
  $st_x \wedge rqst_x \wedge to_x = parent_x \wedge direction_{parent_x} = \mathsf{Grant} \wedge direction_x = \mathsf{Ask}$
    $\wedge request_{parent_x} = request_x \wedge from_{parent_x} = id_x$
    $\rightarrow direction_x = \mathsf{Grant};$
  $frst_x \wedge \neg st_x \wedge direction_x = \mathsf{Ask} \wedge (\exists u \in \mathcal{N}_x | request_u = request_x = from_x = id_x = root_x$
    $\wedge from_u = id_x \wedge direction_u = \mathsf{Grant} \wedge to_x = id_u \wedge root_u > root_x)$
    $\rightarrow parent_x := id_u; distance_x := distance_u + 1; root_x := root_u;$
      $request_x := from_x := to_x := direction_x := \perp;$
}

Listing 3.4: Sub-Algorithmus zur Erzeugung einer aufspannenden Baumstruktur auf Prozess P_x

Das Prädikat st_x ist wie folgt definiert:

$$
\begin{aligned}
st_x \equiv [& \underbrace{(root_x = id_x \wedge parent_x = id_x \wedge distance_x = 0)}_{P_x \text{ ist Wurzelknoten}} \\
& \vee \underbrace{(root_x > id_x \wedge parent_x \in \mathcal{N}_x \wedge root_x = root_{parent_x} \wedge distance_x = distance_{parent_x} + 1)}_{P_x \text{ ist korrekter Blattknoten eines Baums}}] \\
\wedge \quad & \underbrace{root_x \geq max_{i \in \mathcal{N}_x} root_i}_{\text{Der Wurzelknoten ist maximal}}
\end{aligned}
$$

Sollte das Prädikat st_x nicht erfüllt sein, so liegt eine lokale Inkonsistenz vor. Der Algorithmus prüft daher mit weiteren Prädikaten, mit welchen weiteren Schritten Konvergenz hergestellt werden kann. Im weiteren wird daher mit dem Prädikat $frst_x$ nun nur geprüft, ob sich ein Prozess P_x in einem gültigen Umfeld einer Baumstruktur (Wald) befindet, sodass die Konsistenzüberprüfung des Wurzelknotens entfällt:

$$
\begin{aligned}
frst_x \equiv & (root_x = id_x \wedge parent_x = id_x \wedge distance_x = 0) \\
& \vee (root_x > id_x \wedge parent_x \in \mathcal{N}_x \wedge root_x = root_{parent_x} \wedge distance_x = distance_{parent_x} + 1)
\end{aligned}
$$

Ist $frst_x$ nicht erfüllt, so befindet sich P_x nicht in einer konsistenten Baumstruktur. Dies wird behoben, indem sich P_x selbst als Wurzelknoten setzt und somit $frst_x$ erfüllen wird (Guard 1). Im Falle, dass $frst_x$ gilt, aber nicht st_x, gilt es, der benachbarten Baumstruktur mit größerem Wurzelknoten beizutreten (Guard 2). Damit der Prozess nicht kontinuierlich neue Beitrittsanfragen schickt, sondern auf die Antwort der ersten Anfrage wartet, existiert das Prädikat $AlreadyAsking_x$:

$$
\begin{aligned}
AlreadyAsking_x \equiv & [\exists i \in \mathcal{N}_x | (root_i = max_{j \in \mathcal{N}_x} root_j) > root_x] \\
& \wedge request_x = from_x = id_x \\
& \wedge to_x = id_i \\
& \wedge direction_x = \mathsf{Ask}
\end{aligned}
$$

In Guard 2 wird daher mit $\neg AlreadyAsking_x$ sichergestellt, dass nicht schon auf eine Antwort von einer früheren Anfrage gewartet wird. Mit diesem Vorgehen wird schließlich lokale Konsistenz erfüllt und es gilt das Prädikat st_x. Nun fängt der Prozess an, Beitrittsanfragen und -erlaubnisse der Nachbarprozesse weiterzuleiten (Guards 4-7). Doch auch bei der Nachrichtenübermittlung muss auf Korrektheit geachtet werden. Prädikat $rqst_x$ stellt hierbei die Konsistenz der Übertragung in P_x und den Nachbarprozessen sicher und ist wie folgt definiert:

$$
\begin{aligned}
rqst_x \equiv & (\exists i \in \mathcal{N}_x \wedge parent_i = id_i \neq id_x \\
& \wedge request_i = from_i = request_x = from_x = id_i = root_i \\
& \wedge to_i = id_x \wedge direction_i = \mathsf{Ask} \wedge to_x = parent_x) \vee \\
& (\exists j \in \mathcal{N}_x \wedge parent_j = id_x \\
& \wedge \perp \neq request_j = request_x \neq id_j \wedge from_x = id_j \\
& \wedge to_j = id_x \wedge direction_j = \mathsf{Ask} \wedge to_x = parent_x)
\end{aligned}
$$

Sollte das Prädikat nicht erfüllt sein, aber st_x gelten, so setzt P_x die entsprechenden Variablen zurück

(Guard 3). Weiterhin prüft das Prädikat $rqst'_x$ ob der Prozess P_x das Prädikat $rqst_x$ erfüllt oder aber sich die Variablen im zurückgesetzten Zustand (*reset state*), welcher mit $\perp$ definiert ist, befinden:

$$rqst'_x \equiv rqst_x \vee request_x = to_x = from_x = direction_x = \perp$$

Da der Algorithmus selbststabilisierend ist, wird nach einer finiten Anzahl von Schritten Konvergenz erreicht. Mit der aufgespannten Baumstruktur kann dann der globale gegenseitigen Ausschluss realisiert werden (→ 3.2.1).

3.3 Verschärfen der Ausführungsbedingungen

Während die genannten Modifikationen die Konvergenz unter einem dauerhaft fairen Scheduler des Gesamtsystems gewährleisten, ist die Konvergenz unter einem schwach fairen Scheduler nicht gesichert. Einzelne aktivierte Ausführungsblöcke können ignoriert werden und somit aushungern (*starvation*). Um dieses Verhalten zu verhindern, werden die Guards des Sub-Algorithmus zusätzlich verschärft, um vollen Determinismus zu erzielen.
Hierzu gibt es mehrere Möglichkeiten. So kann anhand des Systemzustands durch einen Algorithmus eine Entscheidung generiert werden. Einfacher ist es jedoch, unabhängig vom Systemzustand eine deterministische Auswahl zu treffen. Nach [DT10] prüft man zunächst alle Bedingungen und stellt somit sicher, dass nur maximal ein Guard aktiviert ist, welcher dann ausgeführt wird. Die modifizierte erste Bedingung $\grave{\mathcal{G}}_{i1}$ des ursprünglichen Guards $\mathcal{G}_{i1}$ von Prozess P_i würde daher lauten:

$$\grave{\mathcal{G}}_{i1} :: \mathcal{G}_{i1} \wedge \neg\mathcal{G}_{i2} \wedge \cdots \wedge \neg\mathcal{G}_{in}$$

Die gesamte Transformation beinhaltet darüber hinaus die bereits genannte Verifikation anhand der Bewertungsfunktion (3.2) um die Konvergenz sicherzustellen. Es wird in der Bedingung also weiter konjunkt verknüpft überprüft, ob der globale Zustand nach der Ausführung (σ_{a+1}) der Zuweisungsfunktion act_{i1} niedriger durch die Bewertungsfunktion Δ_A des Algorithmus A bewertet wird als der aktuelle Zustand (σ_a). Wichtig ist hierbei, dass diese Prüfung nur erfolgt, wenn das Prädikat $\mathcal{P}_A$ nicht gilt und das System bereits sich bereits in einem legalen Zustand befindet. Der um diese Bedingungen erweiterte Guard $\grave{\mathcal{G}}_{i1}$ ist demnach:

$$\grave{\mathcal{G}}_{i1} :: (\mathcal{P}_A \vee \Delta_A(\sigma_a) > \Delta_A(\sigma_{a+1})) \wedge \mathcal{G}_{i1} \wedge \neg\mathcal{G}_{i2} \wedge \cdots \wedge \neg\mathcal{G}_{in} \rightarrow act_{i1} \text{ mit } \sigma_a \rightsquigarrow_{act_{i1}} \sigma_{a+1}$$

3.4 Komposition

Die transformierten selbststabilisierenden Algorithmen konvergieren unter jedem beschränkt fairen Scheduler. Sie können daher kombiniert werden bei gleichzeitiger Erhaltung der Eigenschaft der Selbststabilisierung des Gesamtsystems. Sollten sie nicht voneinander abhängig sein, können sie einfach alternierend kombiniert werden. Wenn nur eine einfache Abhängigkeit vorliegt, wie bei Hilfsalgorithmen für den globalen gegenseitigen Ausschluss und dem aufspannenden Baum, können diese auch einfach abwechselnd ausgeführt werden.
Für den Fall, dass die Beziehung der Algorithmen komplexer ist, können zusätzlich die Konvergenzprädikate der Algorithmen verwendet werden. So kann ein Algorithmus explizit darauf warten, dass seine Abhängigkeiten alle korrekt arbeiten, bevor er selbst die Arbeit aufnimmt. Des Weiteren kann somit wechselseitige Abhängigkeiten bis zu einem bestimmten Grad aufgelöst werden. Dafür müssen die Algorithmen in mehrere Ebenen aufgetrennt werden, die dann durch gegenseitige Überprüfung des Konvergenzprädikats sicherstellen, dass die zugehörige Ebene des anderen Algorithmus ebenfalls korrekt arbeitet. Im Endeffekt

wird die gegenseitige Abhängigkeit von zwei Algorithmen damit auf eine Abhängigkeitskette von Komponenten beider Systeme betrachtet und dann aufgelöst.

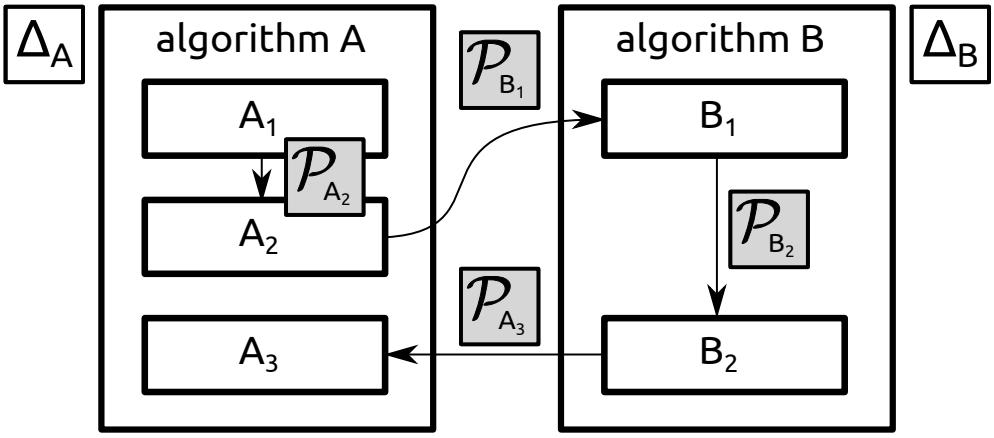


Abbildung 3.4: Komposition mit komplexer Abhängigkeit von Systemkomponenten

In Abbildung 3.4 sind beispielsweise zwei voneinander abhängige Algorithmen A und B zu sehen. Bei genauer Betrachtung stellt sich jedoch heraus, dass nicht die gesamten System voneinander abhängen sondern nur einzelne Systemkomponenten. So hängt die Komponente B_2 von der Komponente A_3 ab. Die gegebene Abhängigkeitskette lässt sich auflösen, da keine zirkuläre Abhängigkeit vorliegt. Entsprechend kann nun mit Konvergenzprädikaten der Systemkomponenten geprüft werden, ob sich die Systemkomponenten bereits in einem legalen Zustand befinden. Erst dann darf die abhängige Komponente fortfahren. Der Abschnitt A_1 von Algorithmus A ist durch die Abhängigkeitskette von allen anderen Komponenten abhängig. Um sicherzustellen, dass die gesamten Abhängigkeiten korrekt arbeiten, braucht A_1 lediglich alle Konvergenzprädikate der Komponenten entlang der Abhängigkeitskette abfragen.

4 Simulationsumgebung zur Transformation selbststabilisierender Algorithmen

Im folgenden wird auf die in dieser Abschlussarbeit entwickelte Simulationsumgebung zur Analyse des Konvergenzverhaltens von transformierten selbststabilisierenden Algorithmen eingegangen. Zunächst werden hierbei die Anforderungen gestellt und danach entsprechend die verwendeten Technologien erläutert. Weiter wird das Konzept sowie die Architektur beschrieben. Es folgen danach detaillierte Beschreibungen wichtiger Komponenten wie der GCL-Implementierung und der Benutzerschnittstelle.

4.1 Zielsetzung

Es soll eine Simulationsumgebung entwickelt werden, welche das Konvergenzverhalten von transformierten selbststabilisierenden Algorithmen unter wechselnden Umgebungen untersucht. Aufgrund der Tatsache, dass die meisten selbststabilisierenden Algorithmen eigene nicht vereinbare Anforderungen an den Scheduler stellen, werden diese in einen Algorithmus transformiert, der unabhängig vom Scheduler konvergiert ($\rightarrow$ 3). Die Wahl der Algorithmen zur Transformation, sowie diverse Eingabeparameter haben dabei Einfluss auf das Verhalten eines verteilten Systems. Von besonderem Interesse ist hierbei die benötigte Zeit zum Erreichen eines legalen Zustands und damit umgekehrt der Zeitraum außerhalb der Spezifikation. Da es erstrebenswert ist, das System stets innerhalb der Spezifikation zu betreiben, wird eine schnelle Konvergenz angestrebt.
Diese Arbeit soll eine Möglichkeit zur Untersuchung der Transformation selbststabilisierender Algorithmen in Bezug auf ihr Konvergenzverhalten anbieten. Verschiedene Hilfsalgorithmen der Transformationen und Konfigurationen können ausgewählt werden und Simulationsläufe mit diesen durchgeführt werden. So können Aussagen über die Eignung bestimmter Hilfsalgorithmen für bestimmte Umstände getroffen werden. Im Optimalfall können damit in Zukunft für jedes Szenario die richtigen Algorithmen zur Transformation gewählt werden, um den Zeitraum der Konvergenz so gut wie möglich zu minimieren. Neben dieser Klassifizierung können auch durch Analyse des Verhaltens bestehender Algorithmen darüber hinaus neue Ideen gewonnen und erprobt werden.
Um diesen Anforderungen gerecht zu werden, muss die Simulationsumgebung zukunftssicher sein. Dies soll durch leichte Wartbarkeit und hohe Modularität realisiert werden, sodass neue Verfahren und Ergänzungen am Simulator gemacht werden können. Zugleich sollte die Bedienung möglichst leicht sein und neue Algorithmen möglichst neu eingegeben werden können. Es ist hierbei wünschenswert, Algorithmen in einem Format mit möglich geringer Abweichung von der Repräsentation in den Veröffentlichungen eingeben zu können. Bei Fehlern im eingegebenen Algorithmus sollen verständliche Fehlermeldungen ausgegeben werden, welche direkt auf das Problem hinweisen und somit eine einfache Korrektur ermöglichen.
Für die statistisch Analyse wird lediglich eine Konsolenanwendung benötigt, die Resultate zu den Eingabedaten liefert. Obgleich die Bedienung selbsterklärend sein sollte, soll eine umfangreiche Hilfe und Korrekturvorschläge bei Fehleingaben die Nutzung des Simulators erleichtern.
Weiterhin soll die Anwendung neben der nüchternen statischen Analyse auch in der Lage sein, das Konvergenzverhalten von verteilten System graphisch zu repräsentieren und auch Möglichkeit zum Eingriff bieten. Die grafische Benutzeroberfläche sollte skalierbar sein und sich somit auch für die Präsentation im großen Maßstab eignen. Auf diese Weise kann weiteres Verständnis für das Verhalten von Algorithmen geschaffen werden oder aber bei der Fehlersuche geholfen werden.
Nicht zuletzt sollte die Interoperabilität beachtet werden und die Simulationsumgebung auf möglichst vielen Plattformen funktionieren, sodass Nutzer zum Beispiel nicht ein bestimmtes Betriebssystem nutzen müssen um die Anwendung zu betreiben.

4.2 Software-Architektur

Nachfolgend wird die Architektur der Simulationsumgebung vorgestellt, welche in dieser Abschlussarbeit entwickelt wurde. Während zuerst auf die Wahl der Projektabhängigkeiten eingegangen wird (→ 4.2.1), folgt ein kompletter Überblick auf die Software (→ 4.2.2). Im Anschluss wird gesondert auf besonders wichtige Module, wie die Sprachimplementierung eingegangen (→ 4.3).

4.2.1 Technologien

Für die Implementierung der Simulationsumgebung ist die Wahl der Programmiersprache von vorrangiger Bedeutung. Die Lehrsprache Java der Universität Oldenburg ist ungeeignet und hätte den Entwicklungsaufwand erheblich in die Höhe getrieben. Stattdessen wurde die junge Skriptsprache Ruby auserkoren, da sie eine sehr schnelle Entwicklung (*rapid prototyping*) ermöglicht. Des Weiteren zeichnet sich Ruby durch ein hohes Maß an Flexibilität und Modularität aus.
Zur grafischen Ausgabe werden die Bindings zum Toolkit Qt sowie die Browserengine WebKit und auch die Graphensoftware Graphviz eingesetzt. Im Verbund ermöglichen die Komponenten eine skalierbare Ansicht von Systemtopologien.

Ruby

Ruby ist eine dynamische Skriptsprache, welche viele Eigenschaften einer ganzen Reihe anderer Sprachen wie Perl, Smalltalk, Eiffel, Ada und Lisp vereint. Sie verfolgt einen objektorientierten Ansatz, kann in Teilen aber auch funktional und imperativ programmiert werden. Während Ruby nach der ersten Veröffentlichung durch Yukihiro Matsumoto im Jahre 1995 nur wenig Anwendung fand, gelang der Durchbruch 2006 mit der Webanwendungsumgebung Ruby on Rails, welche von dort an standardmäßig im Betriebssystem Mac OS X mitgeliefert wird. Seitdem ist Ruby eine feste Konstante in der Webentwicklung und findet zunehmend auch außerhalb Verwendung, was die steigende Zahl der Bibliotheksportierungen belegt. [Rub03]

Abbildung 4.1: Ruby-Logo

Skriptsprachen im Allgemeinen und Ruby im Speziellen neigen durch schwache Typisierung und automatische Speicherverwaltung zu einer höheren Produktivität pro Code-Zeile. In Programmen zu verteilten System sind zudem häufig Listenoperationen auszuführen, welche in Ruby bereits durch syntaktische Features wie iterativen Block-Callbacks vereinfacht wurden.
Es darf jedoch nicht unerwähnt bleiben, dass Skriptsprachen langsamer sind als ihre Kontrahenten. Dieses trifft auch besonders auf die junge Skriptsprache Ruby zu. Während bei Java seit Jahrzehnten die Ausführungszeit optimiert wurde, sind diese Bemühungen bei Ruby erst seit einigen Jahren zu beobachten. So wurde ab Version 1.9.1 die Referenzimplementierung gewechselt auf eine performantere Alternative und es existieren inzwischen auch diverse Ruby-Interpreter wie JRuby, Rubinius, IronRuby, MacRuby und HotRuby, welche unter anderem durch den Einsatz von Just-in-time-Kompilierung erheblich bessere Performance zeigen als die Referenzimplementierung. Besonders hervorzuheben ist hierbei Rubinius, eine Bytecode-VM für die Übersetzungsumgebung LLVM. Sie übersetzt Ruby in den Bytecode der von Apple maßgeblich mitentwickelten LLVM-Umgebung und profitiert von den Optimiertungsverfahren des schon lange bestehenden Projekts. [Rub] Bereits jetzt in der frühen Entwicklungsphase schneidet Rubinius in rechenintensiven Benchmarks besser ab als die Referenzimplementierung [Rub10]. Es zu erwarten, dass sich dieser Trend in Zukunft fortsetzen wird, sodass die Performance bald konkurrenzfähig sein wird.
In Anbetracht der Anforderungen wurde daher nach reiflicher Überlegung Ruby als Sprache für Implementierung gewählt. Die Modularität und Code-Produktivität wurden als wichtiger erachtet als Performance.

Infolgedessen kann mit dieser Arbeit ein größerer Funktionsumfang abgedeckt werden als sonst möglich gewesen wäre.

Graphviz

Graphviz ist eine freie Software zur Visualisierung von Graphen. Während es bei der Ausgabe eine Vielzahl von Strukturen, nach welcher Graphen dargestellt werden sollen, unterstützt, erfolgt die Eingabe in einer simplen textuellen Beschreibung des Graphs. Als grafische Ausgabeformate stehen alle gängingen Typen von Raster- und Vektorgrafiken zu Verfügung. Insbesondere ist es auch möglich, skalierbare Vektorgrafiken (SVG) zu generieren. Da SVG ein XML-Namespace ist und XML menschenlesbar, lassen sich SVG-Dateien leicht per Hand modifizieren, verstehen und komprimiert auch platzsparend speichern. Obgleich mehrere Bindings für Ruby existieren, wurde im Rahmen der Simulationsumgebung auf die Verwendung einer solchen verzichtet, da lediglich ein Bruchteil der Funktionalität verwendet werden würde. Stattdessen werden die Kommandozeilenprogramme von Graphviz verwendet und eine Abhängigkeit auf ein Ruby-Binding verhindert.

Qt

Qt ist eine plattformunabhängige Anwendungsumgebung, welche unter allen gängigen verfügbar ist. Durch die Abstraktion ist möglich, Programme mit Qt zu schreiben, welche dann unter diversen Betriebssystemen laufen. Besonders prominent wird es für die grafische Ausgabe verwendet, obgleich es mit deutlich mehr Funktionalität aufwartet. Im Gegensatz zu anderen prominenten grafischen Anwedungsumgebungen zeichnet sich Qt durch Konsistenz und insgesamt gutes Design aus. Obwohl Qt alleine in der Lage wäre, SVG-Grafiken zu anzuzeigen, wurde im Rahmen dieser Arbeit die Entscheidung getroffen, dafür zusätzlich die Webbrowser-Engine WebKit einzusetzen.
WebKit ist eine Bibliothek zur Verarbeitung, Anzeige und Interaktion von Webseiten und findet in Browsern wie Safari und Chromium Anwendung. Sie ermöglicht es, durch den Einsatz von JavaScript leicht Manipulationen an angezeigten Inhalten vorzunehmen. Da in der grafischen Oberfläche Interaktion möglich sein soll und dafür die Manipulation von Inhalten nötig ist, wird WebKit eingesetzt. Ein weiterer Vorteil dieser Lösung ist, dass Qt in der Lage ist komplette grafische Anwendungsinhalte als Vektorgrafiken abzuspeichern, welches eine besonders hohe Qualität von Anwendungabbildungen zu Dokumentationszwecken zusichert.
Konkret geschieht die Anzeige über das Ruby-Binding QtRuby in Verbindung mit qtwebkit.

TreeTop

TreeTop ist eine Ruby-Bibliothek zur Erzeugung von Syntaxbäumen. Die Erzeugung der Baumstruktur geschieht hierbei unter Beachtung einer speziellen Parser-Grammatik (*parsing expression grammar*, PEG). Während das Übersetzen von einer Quellsprache zu einer Zielsprache normalerweise die Implementierung eines eigenen Parsers und Lexers erfordert, nimmt die Bibliothek einem diese Aufgabe ab, da es aus der Grammatik selbstständig eine Parser-Anwendung in Ruby erzeugt. Dadurch dass bereits in der PEG-Eingabe bestimmten Mustern Klassen zugewiesen werden können, enthält der Syntaxbaum schon semantische Informationen.

4.2.2 Architektur

Abbildung 4.2 zeigt die konzeptuelle Zerlegung der Simulationsumgebung. Bereits hier kann entnommen werden, dass auf starke Modularisierung Wert gelegt wurde. Im folgenden soll die auf die Module eingegangen und ihre Verwendung erläutert werden.
Um überhaupt ein verteiltes System zu simulieren werden Information über die Struktur des Systems benötigt, die Topologie. Ein Modul des Simulator soll die Prozesse mit den korrekten Topologie-Informationen versorgen. Jeder einzelne Prozess verfügt durch die Transformation durch mehrere Ebenen von Hilfsalgorithmen.

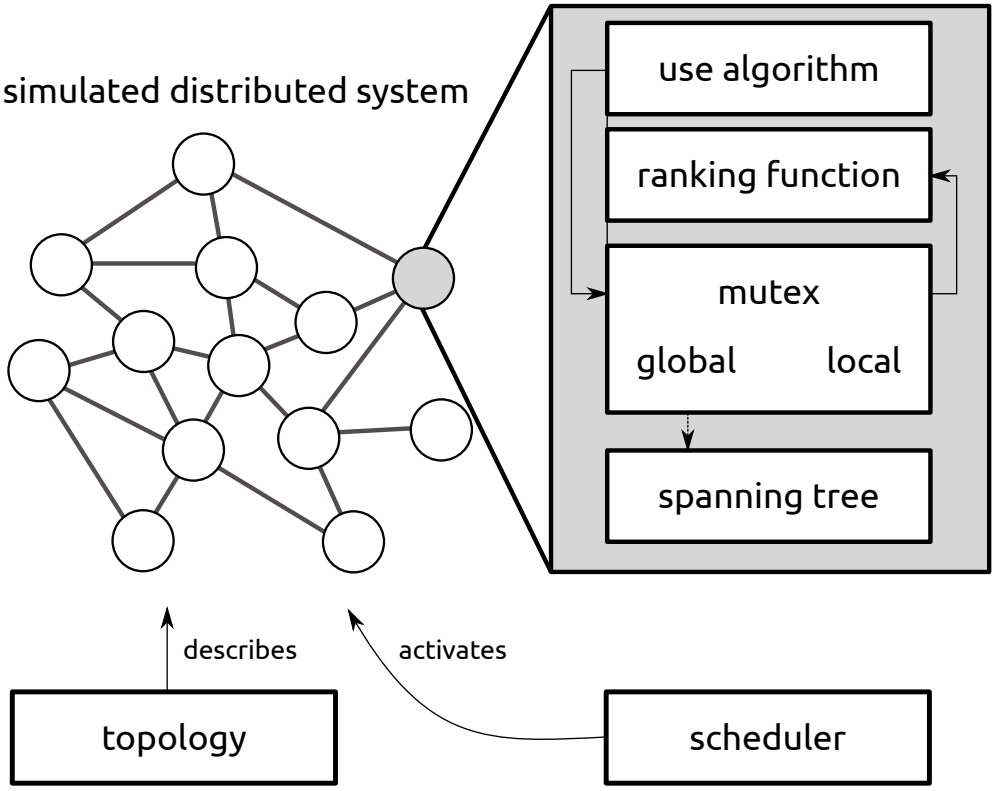


Abbildung 4.2: Konzeptueller Aufbau der Simulationsumgebung

Auf höchster Ebene befindet sich hierbei der Anwendungsalgorithmus. Dieser darf nur genau dann selbst Berechnungen durchführen wenn er dafür die Erlaubnis von der Ebene, welche den gegenseitigen Ausschluss regelt, erhält. Der gegenseitige Ausschluss stellt dann anhand des Systemzustands und der Bewertungsfunktion fest, ob eine solche Erlaubnis gegeben werden kann. Sollte globaler gegenseitiger Ausschluss verwendet werden, existiert noch eine weitere Ebene zur Erzeugung eines aufspannenden Baums. Die Baum-Struktur ist nämlich Ausgangsvorraussetzung für den Algorithmus (3.2.1) und wird daher zwingend benötigt.
Nachdem die Topologie-Informationen geladen und die Prozesse alle richtig initialisiert wurden, übernimmt die Ablaufsteuerung den Betrieb. Es wählt hierbei anhand einer vom Benutzer vorgegebenen Auswahlstrategie pro Runde Prozesse aus und aktiviert diese. Damit das verteilte System selbststabilisierend ist, muss der Scheduler durch die Transformation des Anwendungsalgorithmusses lediglich beschränkt fair sein.

4.2.3 Implementierung

Die Architektur folgt dem vorgestellten Konzept. So wird mit der `Tree`-Klasse des `Topology`-Moduls die eingegebene Topologie geladen. Die Syntax zur Beschreibung ist sehr simpel: Jede Zeile ist eine durch Leerzeichen getrennte Liste von benachbarten Prozessen. Eine Deklaration aller Prozesse ist nicht notwendig. Anschließend wird aus der im Speicher nachgebildeten Baumstruktur für die grafische Oberfläche mit Graphviz ein visuelle Darstellung für das verteilte System generiert.

Für die Simulation der Prozesse im verteilten System werden die Threads verwendet, die von Ruby angeboten werden. Bis Ruby-Version 1.9.1 finden hierbei Userland-Threads, auch *green threads* genannt, Anwendung. Danach wechselte das Ruby-Projekt ihren Referenzinterpreter und fortan werden die Threads vom Betriebssystem verwendet. Es ist jedoch zu beachten, dass auch hier keine echte Parallelität entsteht, da Ruby wie auch Python durch einen globalen gegenseitigen Ausschluss, dem *global interpreter lock* (GIL), sicherstellt, dass nur ein Prozess zur Zeit eine Aktion durchführen kann. [Sch07]

Anstatt direkt Ruby-Threads zu verwenden, findet die modifizierte Klasse `Task` Anwendung, welche den Betrieb durch einen Scheduler ermöglicht. Jedem Prozess wird ein Anwendungsalgorithmus zugewiesen, welcher den Klassentyp `App` hat. Die Klasse setzt sich dabei aus dem eigentlichen Anwendungsalgorithmus sowie den Hilfsalgorithmen für den gegenseitigen Ausschluss und gegebenenfalls für den aufspannenden Baum zusammen. Da der Anwendungsalgorithmus nicht in Ruby sondern einer GCL-Sprachimplementierung geschrieben ist, wird er zunächst mit `Parser`-Modul übersetzt. Das Modul generiert mit der TreeTop-Bibliothek zunächst einen Syntaxbaum. Dieser wird anschließend bereinigt und auf die Semantik reduziert. Anschließend wird die Logik in Ruby übersetzt und dann im Simulator geladen.

Zur Kommunikation stehen jedem Prozess Methoden des Moduls `Communication` von der Simulationsumgebung bereit. Intern sind die Kommunikationsregister durch eine globale Datenstruktur nachgebildet, deren Lese- und Schreibzugriffe über das Monitorkonzept geschützt sind (*threadsafe*).

Die Synchronisation der Rechenschritte übernimmt das `Scheduling`-Modul. Das Modul weckt nach der Ablaufsteuerungsstrategie (*scheduling strategy*) die Threads auf, welche sich nach ihrem Rechenschritt wieder abmelden und dann schlafen legen. Erst wenn sich alle Threads abgemeldet haben, beginnt der nächste Rechenschritt.

Für die grafische Anzeige wird das Modul `UserInterface` benötigt. Dieses stellt mittels Qt und WebKit die Simulation grafisch dar. Um die Ablaufsteuerung interaktiv zu beeinflussen, existieren in der Benutzeroberfläche Buttons, welche entsprechend eine Bedingungsvariable (*condition variable*) verändern, sodass der Scheduler wartet oder fortfährt.

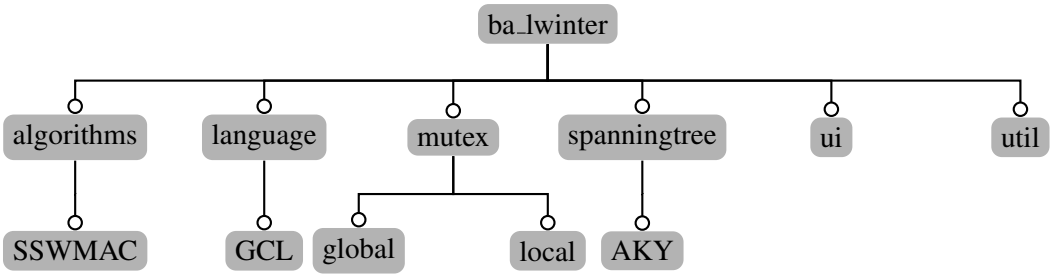


Abbildung 4.3: Verzeichnis-Struktur des Projekts mit Inhalten (vereinfacht)

In Abbildung 4.3 ist die Verzeichnisstruktur des Projekts zu sehen. Das Unterverzeichnis `algorithms` enthält die Anwendungsalgorithmen in der GCL-Sprachimplementierung jeweils mit der Dateiendung `gcl`. Für jeden Anwendungsalgorithmus liegt zusätzlich noch die jeweilige Bewertungsfunktion mit Prädikat $\mathcal{P}$ in einem Rubyskript bei.

Komponenten der Sprachimplementierung befinden sich im Unterverzeichnis `language`. Zur Übersetzung

nach Ruby befindet sich dort die `Algorithm`-Klasse sowie semantische Übersetzungslogik in der Datei `gcl2rb.rb`.

Das Verzeichnis `mutex` beinhaltet die Algorithmen für den gegenseitigen Ausschluss. Jeder dieser Algorithmen implementiert die statische Methode `needSpanningTree?` um dem System anzuzeigen ob ein weiterer Hilfsalgorithmus zur Generierung eines aufspannenden Baums benötigt wird. Entsprechende Algorithmen finden sich im Unterverzeichnis `spanningtree`.

Das Aussehen der Benutzeroberfläche wird durch die XML-Dateien im Verzeichnis `ui` bestimmt. Sie enthalten XML-Elemente, welche durch die `UserInterface`-Klasse modifiziert und mit der Simulationsumgebung verknüpft werden.

Weitere Hilfsroutinen wie die Abstraktion der Visualisierung mit Graphviz befinden sich dann im Verzeichnis `util`.

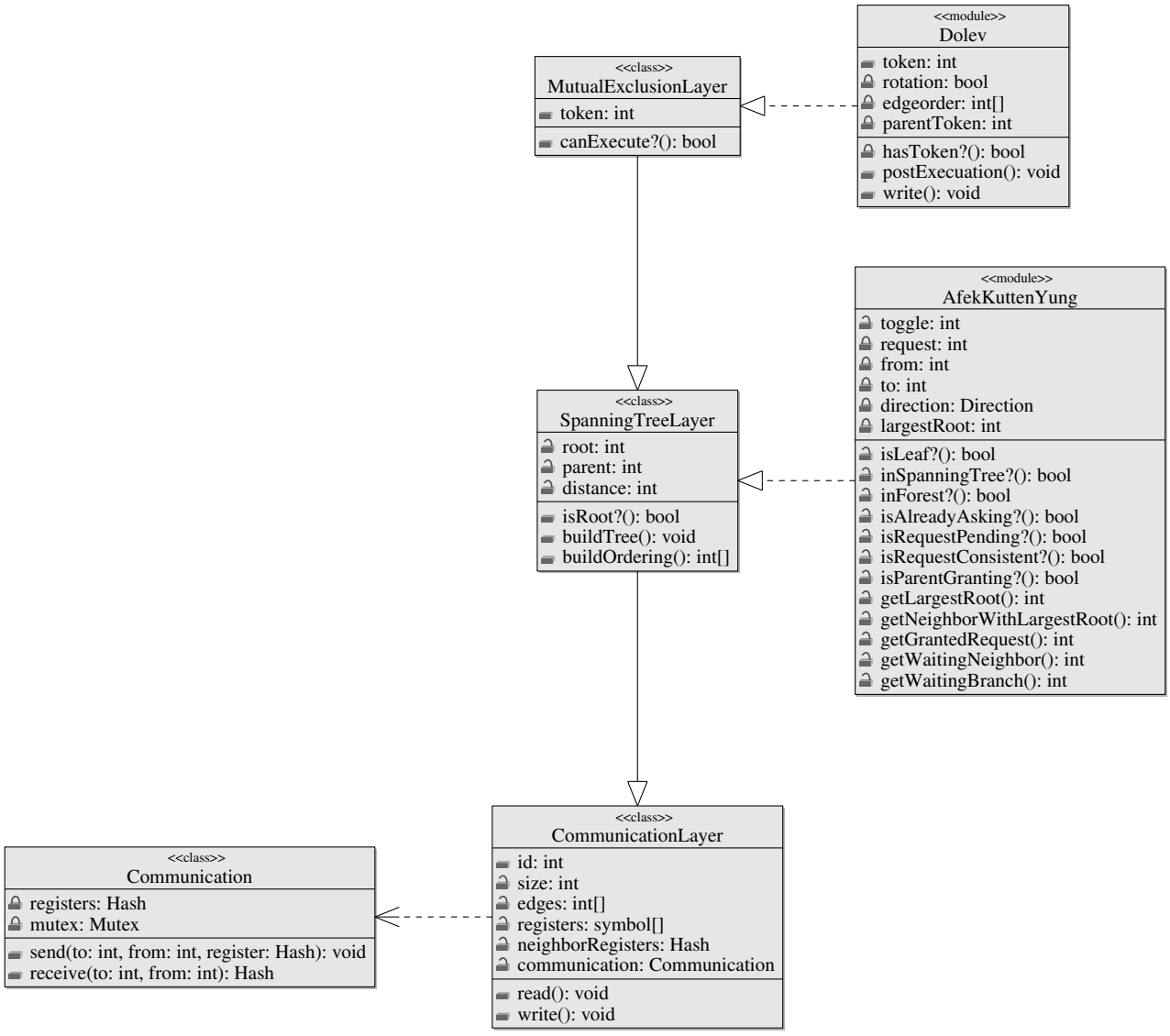


Abbildung 4.4: Konkrete Vererbung für den globalen gegenseitigen Ausschluss

Abbildung 4.4 zeigt die tatsächliche Vererbung der Klassen und Module für den globalen gegenseitigen Ausschluss. Im Gegensatz zum lokalen gegenseitigen Ausschluss wird hier zusätzlich der aufspannen-

de Baum benötigt. Ganz unten ist der `CommunicationLayer` zu sehen, welcher jedem Prozess seine Topologie-Informationen anzeigt und darüber hinaus die Methoden zur Prozesskommunikation enthält.

4.3 Sprache des Anwendungsalgorithmus

In diesem Abschnitt wird auf die Eingabesprache der Anwendungsalgorithmen, die in dieser Abschlussarbeit zusätzlich entwickelt wurde, eingegangen. Dabei werden zunächst die Anforderungen aufgestellt bevor auf die Syntax und Semantik erläutert wird. Abschließend folgen Details zur Implementierung.

4.3.1 Motivation

Die Eingabesprache für Anwendungsalgorithmen sollte der GCL-Darstellung folgen, wie sie in 2.1.1 definiert wurde, um eine einfache Übernahme in die Simulationsumgebung zu vereinfachen. Des Weiteren basiert der Transformationsansatz ($\rightarrow$ 3) auch auf GCL, sodass eine ähnliche interne Repräsentation vorteilhaft ist.
Es existieren mehrere zu GCL ähnliche Sprachimplementierungen, welche vorwiegend aus dem Feld der Programmverifikation stammen. Diese haben allerdings den Nachteil, nicht alle formalen Sprachkonstrukte mitzubringen, mit welcher die Algorithmen ursprünglich beschreiben wurde. Primär mangelt es den Sprachen an Quantoren, sodass bei großen Systemtopologien entsprechend lange Konjunktionen und Disjunktionen geschrieben werden müssen.
Im Rahmen dieser Bachelorarbeit wurde daher eine neue GCL-ähnliche *Domain Specific Language* (DSL) entwickelt, die den Anforderungen besser als bestehende Sprachimplementierungen gerecht werden. Die DSL ist der formalen GCL-Beschreibung vieler Algorithmen sehr ähnlich und beinhaltet auch Quantoren. Um weitere Entwicklung und Erweiterung zu erleichtern, ist der Übersetzungsvorgang in mehrere Module gekapselt. So ist es möglich aus der semantischen Baumstruktur auch andere Zielsprachen als Ruby zu generieren.

4.3.2 Struktur

Die Sprachimplementierung hält sich weitestgehend an die initiale Spezifikation von Dijkstra [Dij75]. Da GCL lediglich zur Beschreibung von Algorithmen konzipiert wurde, enthält es Sonderzeichen wie Quantoren und damit nicht ohne Weiteres einzugeben sind. Infolgedessen werden solche Sonderzeichen in dieser Sprachimplementierung entweder durch ähnliche Zeichen oder textuelle Beschreibungen nachgebildet.
Wie auch in der Definition (2.1.1) beschrieben ist ein GCL-Algorithmus in zwei Bereiche unterteilt. Abbildung 4.5 zeigt hierbei den Kopfbereich mit der Definition der lokalen Variablen, Konstanten und Makros detailliert. Wie in der Abbildung zu sehen ist, fängt jeder deklarative Unterbereich mit einem entsprechenden Schlüsselwort an. Diesem folgt eine beliebige Anzahl von Definitionen, die jeweils mit einem Komma voneinander getrennt sind. Abschließend wird der Unterbereich mit einem Semikolon geschlossen.
Lokale Variablen können beliebig mit Zeichen aus dem Alphabet benannt werden; der Anfangsbuchstabe muss jedoch klein sein. Konstanten hingegen müssen komplett groß geschrieben werden. Beiden können beliebige Ausdrücke zugewiesen werden.

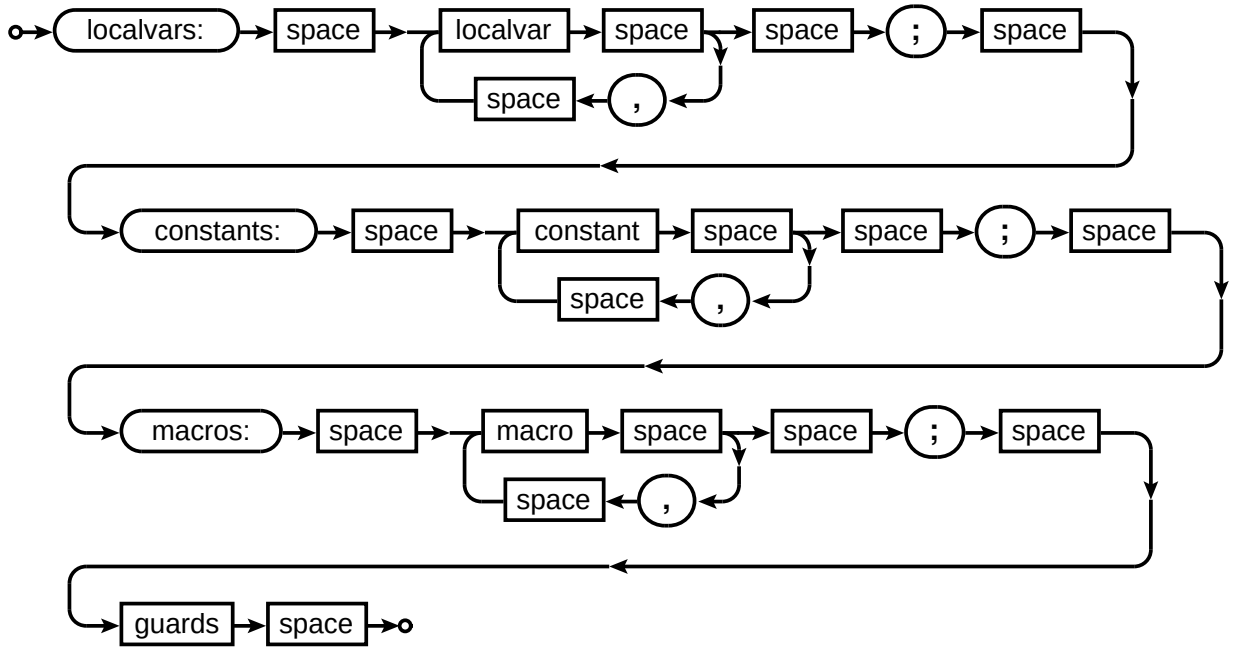


Abbildung 4.5: Aufbau eines Algorithmus in GCL

Um auf Informationen vom System zuzugreifen zu können, müssen Systemvariablen verwendet werden. Diese werden von der Simulationsumgebung bereitgestellt und fangen mit einem Unterstrich an.
Den Variablen und Konstanten folgen Makros. Makros sind Ausdrücke, die stets einen Wahrheitswert zurückgeben, also Prädikate. Diese Semantik wird auch bei der Übersetzung überprüft und ein Ausdruck, welcher nicht bool'sch ist, mit einer Fehlermeldung quittiert.

```
rule guard
   l:label? cond:(bool) space '->' space exec:(proc) ';' <GuardNode>
end
```

Listing 4.1: PEG-Regel für Guards in der Sprache TreeTop

Schließlich folgen die Guards, welche die Anwendungslogik darstellen. Listing 4.1 zeigt einen Ausschnitt aus Grammatik für die GCL, der die Regel für das Vorkommen eines Guards enthält. Nach einer optionalen

Beschriftung folgt in jedem Guard die Bedingung, welche die nachfolgende Ausführungsfunktion schützt. Die Bedingung muss zwingend ein bool'scher Ausdruck und wird auch semantisch nach diesem Kriterium geprüft. Jeder Guard endet mit einem Semikolon.
Als angewandtes Beispiel folgt nun der bekannte Algorithmus $\mathbb{SSWMAC}$ aus 2.1, welcher in Listing 4.2 in der GCL-Sprachimplementierung dargestellt ist. Die Abweichungen zur Originalbeschreibung sind dabei minimal. Lediglich die Definitionen von der κ-lokale Nachbarschaft $\mathcal{N}_i^\kappa$ und dem Maximum $\tilde{n}$ entfallen, da diese von der Simulationsumgebung als Systemvariablen bereitgestellt werden. Weiterhin ist das Makro min_x etwas verändert um auf eine explizite Definition von $\mathcal{N}_i^{\kappa\perp}$ zu verzichten.

```
localvars:
    turn = 0,
    slot = -1;

constants:
   MAX = _nk;

macros:
    min(x)       { forall j in _khops: j.slot != -1 || j.turn > x },
    unique(x)    { forall j in _khops: j.turn != x },
    valid(x)     { min(x) && unique(x) };

<a> :: slot == -1 && turn <= MAX && valid(turn)
   -> { slot = turn };
<b> :: slot != -1 && turn <= MAX && !unique(turn)
   -> { slot = -1 };
<c> :: slot > MAX || turn > MAX
   -> { turn = (turn + 1) % (MAX - 1); slot = -1 };
<d> :: slot == -1 && turn <= MAX && !valid(turn)
   -> { turn = (turn + 1) % (MAX - 1) };
```

Listing 4.2: Sub-Algorithmus $\mathbb{SSWMAC}$

Im Makrobereich des Listings sind Quantoren zu sehen. Dort ist ein Variablenbezeichner aus einer bestimmten Menge zu sehen, welcher im Bereich nach dem Doppelpunkt gültig ist und die Elemente der Menge repräsentieren. Die Sprachimplementierung verfügt hierbei auch über die Technik der Variablenüberschattung (*variable shadowing*). So verfügt ein Iterator eines Quantors im Wirkungsbereich (*scope*) über die Gültigkeit einer lokalen Variable und verdeckt gleichlautende Variablen.
In der Ausführungsfunktion der Guards sind Zuweisungsoperationen zu sehen. Diese können beliebig komplex verschachtelt und geklammert werden. Es stehen sämtliche arithmetischen Operatoren sowie Modulo-Rechnung zur Verfügung. Als logische Verküpfungen können die Konjunktion und Disjunktion jeweils in C-Notation verwendet werden. Der einstellige Negationsjunktor steht zur Invertierung von Wahrheitswerten durch ein vorangestelltes Ausrufezeichen zur Verfügung.

4.3.3 Implementierung

Die Verarbeitung der in GCL geschriebenen Algorithmen übernimmt die Klasse `Algorithm` des Moduls `parser` der Simulationsumgebung. In der Abbildung 4.6 ist die Klasse umrandet zu sehen. Gestrichelt ist jeweils ein Zustand vom Algorithmus, welcher übersetzt wird. Der Übersetzungsvorgang selbst in modular in mehrere Phasen eingeteilt. Zunächst wird die Eingabe mit dem Parser, welcher von TreeTop aus der Parser-Grammatik (*parser expression grammar*, PEG) erzeugt wurde, auf syntaktische Korrektheit geprüft und dabei ein Syntaxbaum erstellt. Dieser Syntaxbaum wird nun auf die semantisch wichtigen Elemente reduziert. In diesem Zuge werden sämtliche Leerzeichen oder andere Einrückungselemente aus dem Syntaxbaum entfernt. Es folgt eine Plausiblitätsprüfung, welche semantische Fehler wie doppelte Variablendeklarationen aufspürt und eine formatierte Fehlermeldung generiert. Danach übernimmt ein weiteres Modul die Übersetzung in die Zielsprache. Für die weitere Verwendung innerhalb der Simulationsumgebung ist dies Ruby, daher wird mit dem Modul `grl2rb` entsprechend Ruby-Quelltext generiert.

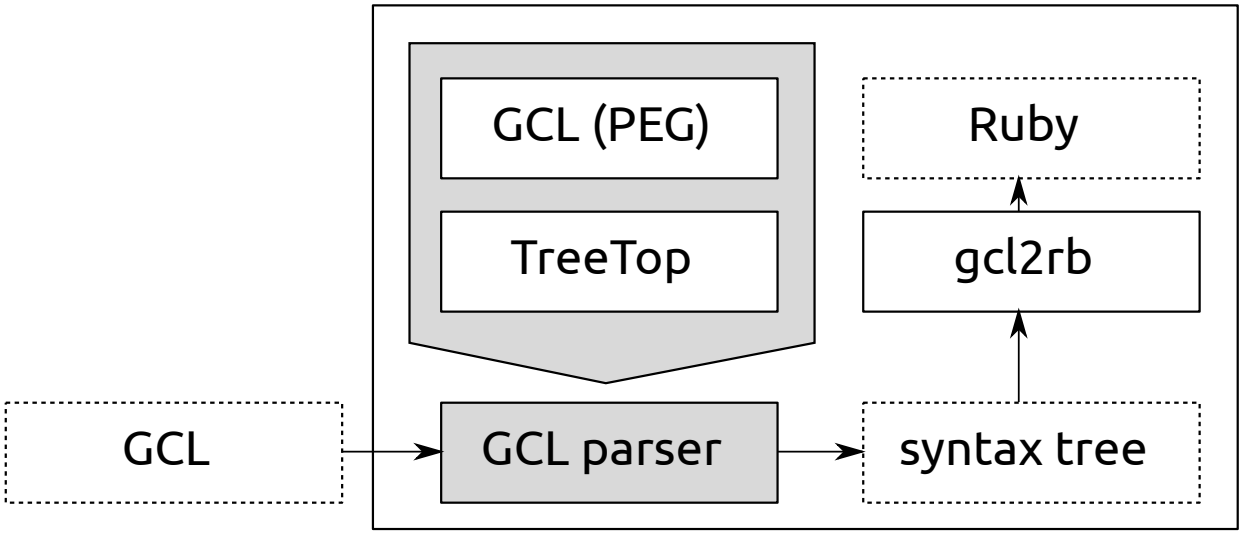


Abbildung 4.6: Komponenten der Sprachübersetzung

Das generierte Ruby-Skript nutzt die `Guard`-Klasse zum Definieren von Ausführungsbedingung. Alle Guards werden in einer Hashmap der entsprechenden Zuweisungsfunktion zugeordnet. Während der Simulation iteriert dann die `run`-Methode der Klasse `UseAlgorithm` die Hashmap und überprüft ob der Guard aktiv ist. Als weiterer Transformationsschritt wird das Konvergenzprädikat $\mathcal{P}$ vom Anwendungsalgorithmus überprüft (3.2). Sollte das Prädikat wahr sein, so wird die Zuweisungsfunktion einfach ausgeführt. Andernfalls wird ein Schnappschuss vom System gemacht, die Zuweisungsfunktion ausgeführt und dann die Zustände mit der Bewertungsfunktion verglichen. Bei einer Verminderung des Wertes der Bewertungsfunktion findet keine weitere Aktion statt; bei Gleichheit oder Erhöhung wird das System auf den Zustand vor der Ausführung der Zuweisungsfunktion zurückgesetzt. Der Determinismus bei der Auswahl der Guards (3.3) ist dadurch gegeben, dass die Guards stets der Reihenfolge nach iteriert werden und immer der erste aktive Guard ausgewählt wird.
Neben der Übersetzung speichert die `Algorithm`-Klasse des Weiteren die Namen der lokalen Variablen um diese Information in der grafischen Oberfläche zu nutzen.

4.4 Benutzerschnittstelle

Die Benutzerschnittstelle ist für eine Simulationsumgebung von entscheidender Bedeutung, da der Nutzer durch Verwendung dieser Schnittstelle neue Erkenntnisse gewinnen soll. Sowohl die Konfiguration und Bedienung als auch die Ausgabe sollte hierbei verständlich sein und den Nutzer soweit wie möglich unterstützen. Um den unterschiedlichen Verwendungszwecken gerecht zu werden, ist die Simulationsumgebung in zwei Benutzerschnittstellen aufgegliedert. Während die grafische Oberfläche eher zur interaktiven Exploration und zu Präsentationszwecken Verwendung findet, ist die Verwendung auf der Kommandozeile für automatisierte Benchmarks und Analysen anzuraten.

4.4.1 Kommandozeile

Die Simulationsumgebung verwendet die von Ruby bereitgestellte `optparse`-Bibliothek zum Parsen von Parametern. Diese vereinfacht das Auslesen der Argumente deutlich, unterstützt auch lange Parameternamen und generiert darüber hinaus eine vollständige Hilfsausgabe, die alle Parameter anzeigt.

```
Usage: simulator.rb [-g] [-Z] -t TOPOLOGY
         -m MUTEX-ALGORITHM [-s SPANNING-TREE-ALGORITHM]
         -u USE-ALGORITHM -k K -p SCHEDULING-POLICY
        [-n N] [-x] [-L LOG-LOCATION]

    -g, --gui                        Toggle interactive graphical
                                     ↪ user interface
    -t, --topology TOPOLOGY-FILE     File describing the toplogy of
                                     ↪ the distributed system
    -m, --mutex MUTEX-ALGORITHM      Mutual Exclusion algorithm
    -s SPANNING-TREE-ALGORITHM       Spanning Tree algorithm
        --spaning-tree
    -k N                             Parameter k for k-local algorithms
    -n, --repeat-count N             Number of repetition for
                                     ↪ simulation run
    -u USE-ALGORITHM                 Use Algorithm to be transformed
        --use-algorithm              ↪ and used
    -p, --scheduling-policy POLICY   Scheduling policy for process
                                     ↪ scheduling
    -x, --exit-when-safe             Quit simulation when system is
                                     ↪ convergent
    -h, --help                       Show this help
    -L, --log-location LOGDIR        Directory to place to logfile
    -Z, --start-paused               In GUI mode this options will set
                                     ↪ the simulation to pause
```

Listing 4.3: Übersicht über die Eingabeparameter

Listing 4.3 zeigt die Eingabeparameter der Simulationsumgebung. Mittels dieser lassen sich die Simulationen ausführlich konfigurieren. Im folgenden sollen die Parameter beschrieben werden. Die Datei mit der Topologiebeschreibung wird über den Parameter `-t` angegeben und bestimmt die Verbindungen der Prozessoren des verteilten Systems untereinander. Über den Parameter `-u` übergibt man Anwendungsalgorithmus, der transformiert und simuliert werden soll. Um das κ von einem κ-lokalen Algorithmus zu wählen, wird die der Parameter `-k` benötigt. Zur Simulation wird ein Hilfsalgorithmus für den gegenseitigen Ausschluss benötigt, welcher mit `-m` gesetzt wird. Benötigt der Algorithmus für den gegenseitigen

Ausschluss zusätzlich eine aufspannende Baumstruktur so kann ein entsprechender Algorithmus mit den Parameter `-s` bestimmt werden. Weiterhin wichtig für die Simulation ist die Auswahl der Ablaufsteuerung. Mit `-p` kann daher die Auswahlstrategie des Schedulers festgelegt werden.
Es stehen weiterhin eine Reihe optionaler Parameter bereit. Wird der Parameter `-x` angegeben, beendet sich die Simulationsumgebung, wenn Konvergenz vorliegt. So bestimmt der Parameter `-n`, wie oft die Simulation wiederholt werden soll. Die Schritte zur Konvergenz werden standardmäßig auf die Konsole geschrieben. Um dieses Verhalten zu ändern, kann mit dem Parameter `-L` ein Verzeichnis bestimmt werden, wo die Protokolldateien mit den Konvergenzzeiten gespeichert werden sollen. Schließlich besteht auch die Möglichkeit der grafischen Anzeige. Mit dem Parameter `-g` wird die grafische Benutzeroberfläche aktiviert. Wird zusätzlich noch Parameter `-Z` angegeben, startet die Oberfläche im pausierten Zustand, sodass der Betrachter keinen Rechenschritt der Simulation übersieht.
Für die statistische Auswertung von Protokolldateien kann das beiliegende Ruby-Skript `Analyser.rb` verwendet werden. Ruft man es mit Pfad der Protokolldatei als ersten Parameter auf, wertet es diese Datei aus und gibt entsprechend den Mittelwert und die Standardabweichung sowie Minimum und Maximum des Datenbestands an.

Eingabeunterstützung

Bei Fehleingaben unterstützt das Programm das Finden der richtigen Eingabe. So listet es für die Hilfsalgorithmen für den gegenseitigen Ausschluss, den aufspannenden Baum und den Anwendungsalgorithmen sowie den Auswahlstrategien des Schedulers jeweils alle verfügbaren Optionen auf.

```
$ ruby1.8 simulator.rb -m test
The mutex algorithm 'test' could not be found
Try one of the following algorithms instead:

name                    need spanning tree?
------------------------------------------
MutualExclusion         yes
LocalMutualExclusion    no
```

Listing 4.4: Verbesserungsvorschläge bei Fehleingabe

Listing 4.4 zeigt beispielsweise die Ausgabe der Simulationsumgebung nach einer Fehleingabe. Die Ausgabe besteht dabei zunächst aus einer eindeutigen Beschreibung des Fehlers. Darauf werden alle möglichen korrekten Eingaben angezeigt. Bei dem angegebenen Beispiel handelt es sich um die Liste aller Algorithmen für den gegenseitigen Ausschluss. Diese umfasst des Weiteren noch die Information, ob der gelistete Algorithmus noch einen weiteren Algorithmus für eine aufspannende Baumstruktur benötigt.
Nicht nur die Eingabeparameter verfügen über eine Fehlerbehandlung sondern auch das Modul für Verarbeitung des Anwendungsalgorithmus. Bei syntaktischen Fehlern wirft der TreeTop-Parser eine Warnung und gibt entsprechend die erlaubten Token an der Fehlerstelle gemäß seiner Grammatik an. Das Modul zur Sprachverarbeitung führt auch einfache Prüfungen an der Semantik durch.

```
line 3, column 24 (byte 35): Variable "slot" defined more than once
↪ (SemanticError)

   turn = 0, slot = -1, slot = 9
                        ^^^^^^^^
```

Listing 4.5: Semantischer Fehler im Anwendungsalgorithmus

Listing 4.5 zeigt die Ausgabe der Simulationsumgebung bei einem semantischen Fehler im Anwendungsalgorithmus. Konkret wurde eine Variable zweimal deklariert. Nach Angabe der Fehlerposition erfolgt zudem die Ausgabe der Zeile mit Hervorhebung der Fehlers.

4.4.2 Graphisch

Zur Präsentation oder Lehrzwecken sind interaktive Demonstrationen besonders beliebt. Diesem Wunsch soll mit der grafischen Benutzeroberfläche Rechnung getragen werden. Die grafische Oberfläche zeigt standardmäßig die Topologie und die Variablen des Anwendungsalgorithmus in eckigen Klammern an. Ist der lokale gegenseitige Ausschluss aktiv, werden zusätzlich die beiden Uhren vom Hilfsalgorithmus in normaler Klammerung angezeigt. Wird der globale gegenseitige Ausschluss genutzt, wird stattdessen in normaler Klammerung das Token angezeigt. Des Weiteren stehen werden zusätzliche Buttons in der Benutzeroberfläche aktiviert, die es ermöglichen das System aus einer anderen Sichtweise zu betrachten. Zum ersten kann somit die dynamisch erzeugte Baumstruktur eingesehen und zum anderen der virtuelle Ring der Eulertour betrachtet werden. Letzteres ist erst dann möglich, wenn der die aufspannende Baumstruktur konsistent ist.

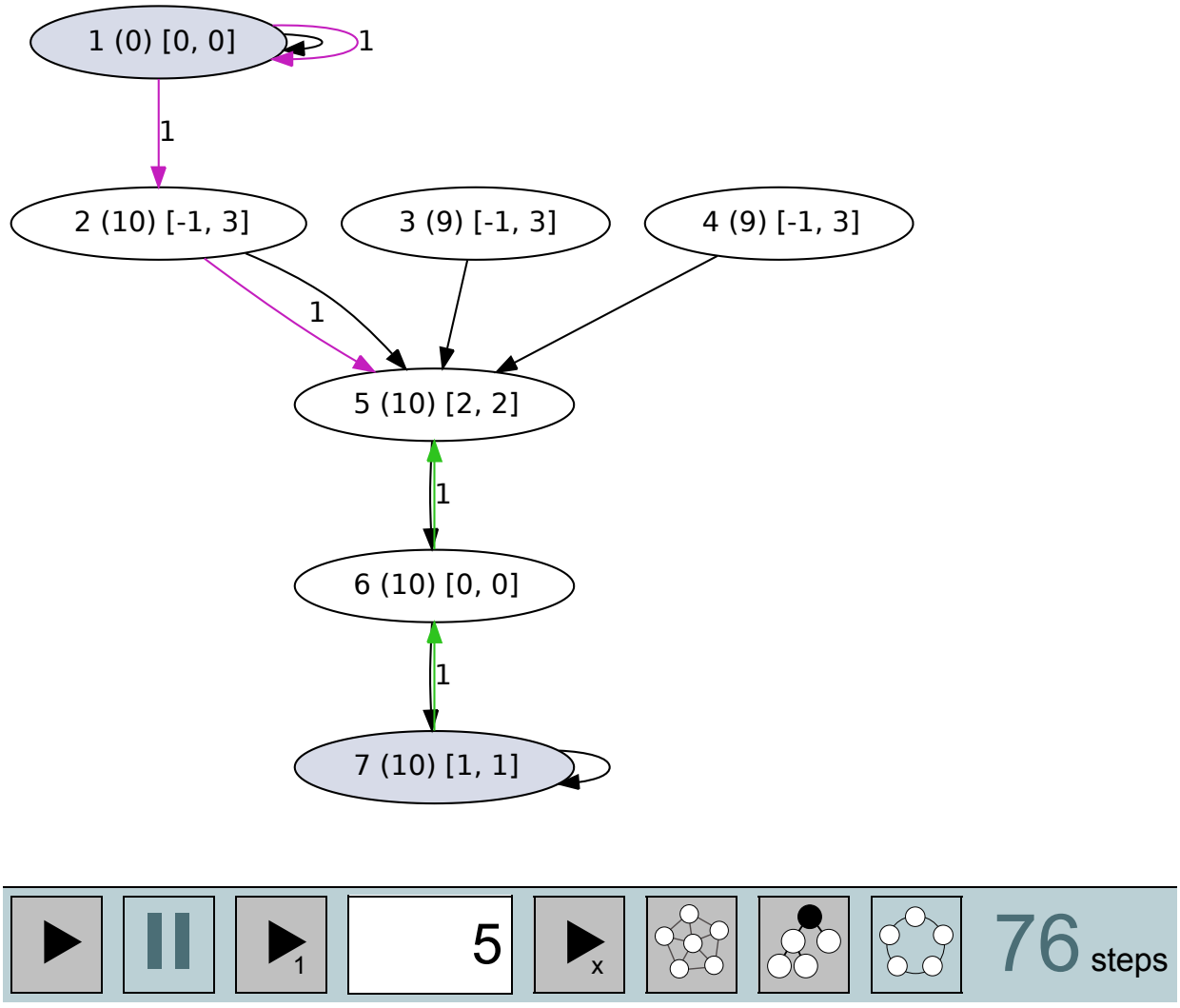


Abbildung 4.7: Screenshot der Simulationsumgebung

Abbildung 4.7 zeigt die Simulationsumgebung in Aktion. Im ausgewählten Betrachtungsmodus wird die Baumstruktur angezeigt, welche gerade über den Algorithmus zur Erzeugung einer Baumstruktur erzeugt wird. Konkret ist zu sehen, wie der Prozess P_1 sich in den Baum eingliedern möchte und entsprechend eine Beitrittsanfrage, die violett eingezeichnet ist, stellt. Die Anfrage ist bereits vom Wurzelknoten P_7 genehmigt worden, was anhand der Erlaubnisnachricht, die grün dargestellt ist, zu sehen ist. Der Wurzelprozess ist grau markiert und dadurch ersichtlich, dass er auf sich selbst als Vaterknoten zeigt.
In der unteren Leiste in der grafischen Benutzeroberfläche befinden sich neben der Buttons für den Anzeigemodus noch weitere Buttons auf der linken Seite. Mit diesen Knöpfen kann interaktiv in die Simulation eingegriffen werden. So kann mit den ersten beiden Buttons die Simulation angehalten oder das Fortfahren aktiviert werden. Wird der dritte Button gedrückt, wird lediglich ein Rechenschritt ausgeführt und die Simulation wieder angehalten. Möchte man den Zeitraum bis zur nächsten Pause flexibler gestalten, so kann über das Eingabefeld und den dazugehörigen Button eine beliebige Anzahl von Rechenschritten bestimmt werden. Diese werden dann ausgeführt bevor die Simulation wieder pausiert wird.
Die Anzahl der bereits getätigten Rechenschritte wird ganz rechts angezeigt. Wurde der Simulator mit den Optionen ausgeführt, dass er mehrere Simulationen durchführen soll und bei Konvergenz die aktuelle Simulation beendet, kann beobachtet werden, wie der Zähler regelmäßig zurückgesetzt und das System für eine neue Simulation zurückgesetzt wird.
Eine große Besonderheit ist die Skalierbarkeit der grafischen Oberfläche. So ist es möglich bei gedrückter `Ctrl`-Taste mit dem Scrollrad oder den Tasten Plus und Minus die gesamte Oberfläche zu skalieren. Hierbei treten keine Artefakte auf, da die komplette Oberfläche aus Vektorgrafiken und skalierbarem XHTML besteht. Durch diese Eigenschaft eignet sich die Simulationsumgebung besonders gut für Demonstrationszwecke.

5 Fallstudien der Simulation

Im folgenden soll der Zeitraum, den der verteilte Algorithmus $\mathbb{SSWMAC}$ auf verschiedenen Topologien zum Erreichen der Konvergenz benötigt, gemessen werden. Dazu wurde das verteilte System 1000 mal mit zufälligen Werten in den Anwendungsvariablen befüllt und dann die Rechenschritte bis zum Erreichen eines legalen Zustands gezählt. Diese Simulation wurde dann unter den Transformationen mit globalen gegenseitigen Ausschluss nach [DIM93] und lokalen gegenseitigem Ausschluss nach [BP08] ausgeführt. Der Algorithmus für den gegenseitigen Ausschluss lief dabei in Komposition mit dem Algorithmus zur Erzeugung einer Baumstruktur nach [AKY97].
Die Darstellung der Simulationsläufe besteht jeweils aus einer Ergebnistabelle und einer Visualisierung des Konvergenzverhaltens. In dem abgebildeten Graph sind dabei die Ausführungsschritte (*steps*) auf der Y-Achse zu sehen, während auf der X-Achse κ angegeben wird. Jeder Simulationslauf des lokalen und gegenseitigem Ausschluss unter einem κ wird durch eine blaue Linie visualisiert, welche Minimum und Maximum anzeigen. Die größeren Blöcke in der Mitte jeder Linie drücken den Mittelwert mit Standardabweichung aus.
Berechnet wurde die Standardabweichung als Wurzel der Varianz:

$$\sigma_X := \sqrt{Var(X)}$$

Die Varianz ist als Summe der Differenzen von den Werten x_i und dem Durchschnittswert $\bar{X}$ der Menge X dividiert durch die Kardinalität der Menge n ($= |X|$) definiert:

$$Var(X) := \sum_{i=0}^{n} \left(\frac{x_i - \bar{X}}{n} \right)$$

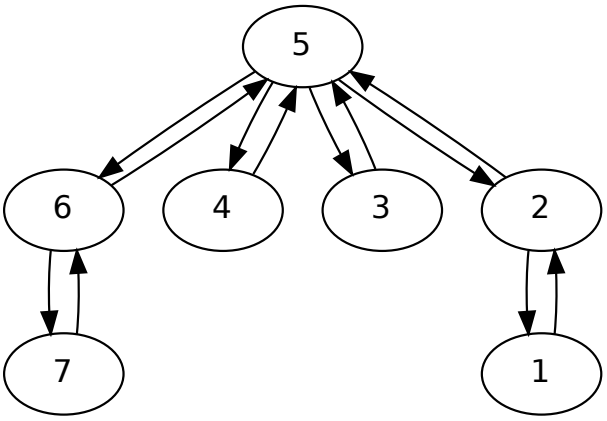
Der Durchschnittswert der Simulationsläufe wird mit ø, die Standardabweichung mit σ, Minimum und Maximum je mit min und max annotiert. Die Angabe der Werte folgt gerundet auf die zweite Dezimalstelle. Bei den Mini- und Maximalwerten erfolgt keine Rundung, da sie eine nicht gerundete Anzahl von Rechenschritten ausdrücken.

5.1 Erste Testreihe

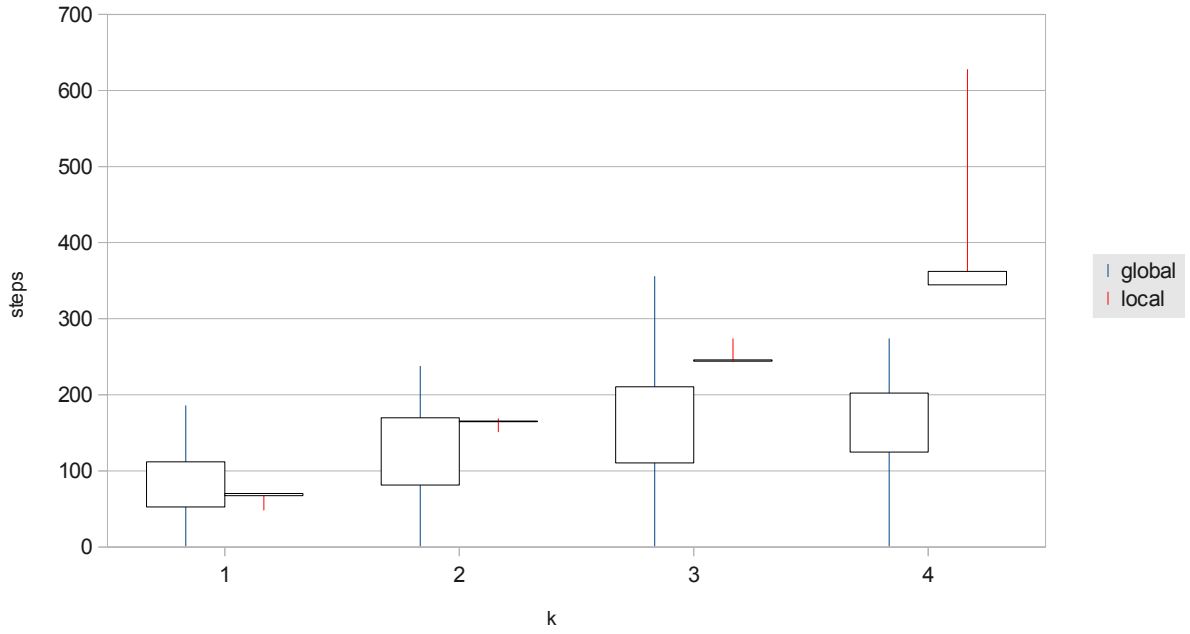
In dieser Testreihe soll das Konvergenzverhalten der Algorithmen für den gegenseitigen Ausschluss untersucht werden.
Abbildung 5.1 (b) zeigt die Topologie des zu untersuchenden verteilten Systems. Die Topologie ist baumförmig, dennoch wird beim globalen gegenseitigen Ausschluss eine neue Baumstruktur generiert (mit P_7 als Wurzelknoten). Bei $\kappa = 2$ umfasst die κ-lokale Nachbarschaft von P_5 bereits alle Prozesse, während die κ-lokale Nachbarschaft von P_1 und P_7 erst bei $\kappa = 4$ die maximale Größe erreicht.
In Abbildung 5.1 (a) und visualisiert in (c) sind nun die Ergebnisse der Simulation zu betrachten. Während die Laufzeit des lokalen gegenseitigen Ausschlusses linear mit κ wächst, bleibt die Laufzeit des gegenseitigen Ausschlusses in der Regel im gleichen Bereich. Lediglich bei $\kappa = 1$ konvergiert der gegenseitige Ausschluss schneller. Es fällt auf, dass der lokale gegenseitige Ausschluss ein sehr viel konstanteres Laufzeitverhalten hat und die Konvergenzzeit keine großen Abweichungen zeigt. Ausnahme ist hierbei $\kappa = 4$, wo es extreme Schwankungen nach oben gab.
Im folgenden sind die Ergebnisse der Simulationen zu interpretieren. Der globale gegenseitige Ausschluss benötigt zur Konvergenz eine konstante Zeit, die unabhängig von κ ist. Nur der darauf aufbauende κ-lokale Anwendungsalgorithmus benötigt bei größerem κ mehr Zeit. Somit ist beim globalen gegenseitigen Ausschluss nur ein moderater linearer Anstieg der Durchschnittswerte zu beobachten, der durch die längere Konvergenzzeit des Anwendungsalgorithmus bei höherem κ zustande kommt. Weiterhin ist zu beobachten, dass der globale gegenseitige Ausschluss eine enorme Schwankung der Konvergenzzeiten zeigt. Dieses Phänomen lässt sich durch den Hilfsalgorithmus des gegenseitigen Ausschlusses erklären. Dadurch, dass der Algorithmus auf der Baumstruktur arbeitet und der Algorithmus zur Erzeugung einer Baumstruktur bereits nach dem ersten Schritt alle Prozesse in eine gültige Baumstruktur versetzt, kann jeder Prozess schon auf einer beschränkten Baumstruktur arbeiten und somit auch Rechenschritte des Anwendungsalgorithmus ausführen. Somit kann eine Konvergenz bereits erfolgen noch bevor die Baumstruktur komplett aufgebaut wurde. Die Ausschläge nach oben zeigen allerdings auch, dass diese Vorgehensweise auch kontraproduktiv zum Konvergenzstreben sein kann.
Der lokale gegenseitige Ausschluss hingegen genehmigt keine Rechenschritte des Anwendungsalgorithmus bevor eigene Konvergenz vorliegt. Die Konvergenz des lokalen gegenseitige Ausschlusses wird über Synchronisation beider Uhren erreicht. Dazu werden alle Uhren auf α zurückgesetzt. Da α von κ multiplikativ anhängig ist, kann daher eine direkte lineare Abhängigkeit bei den Durchschnittswerten der Konvergenz beobachtet werden. Durch dieses Vorgehen wird stets eine minimale Zeit zur Konvergenz benötigt und die Schwankungen der Konvergenz des Systems sind minimal. Lediglich $\kappa =$ zeigt ein stark erhöhtes Maximum. Diese hohe Konvergenzzeit kann durch ungünstige Sendeplatzentscheidungen im Anwendungsalgorithmus zustande zu kommen. Anders als beim globalen gegenseitigen Ausschluss ist die Wartezeit zwischen der Ausführung zweier Prozesse konstant bei κ Rechenschritten. Sollte der Anwendungsalgorithmus also lange zur Konvergenz brauchen, vervielfacht sich die Zeit durch die Unterbrechungen des lokalen gegenseitigen Ausschlusses. Da bei $\kappa = 4$ jede κ-lokale Nachbarschaft das gesamte System umfasst, zeigt dieser Fall, dass der lokale gegenseitige Ausschluss auf ganzen Systemen nicht besonders gut skaliert.
Insgesamt zeigt der globale gegenseitige Ausschluss meistens eine bessere Leistung auf dieser Topologie, wenngleich hier mit größeren Schwankungen bei den Konvergenzzeiten gerechnet werden muss. Lediglich für $\kappa = 1$ zeigt der lokale gegenseitige Ausschluss eine überlegene Leistung durch eine niedrigere Konvergenzzeit und minimale Schwankung.

	κ			
	1	2	3	4
lokal				
ø	68,89	165,01	245,05	353,36
σ	1,53	0,51	0,93	8,81
min	48	151	243	353
max	70	169	274	628
global				
ø	82,19	125,62	160,57	163,53
σ	29,73	44,12	50,09	38,73
min	1	1	1	1
max	186	238	356	274

(a) Rechenschritte der Simulationsläufe



(b) Topologie des verteilten Systems



(c) Visualisierung der Konvergenzzeiten

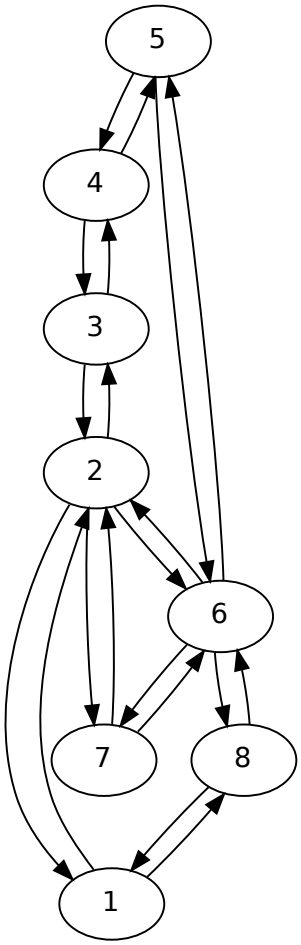
Abbildung 5.1: Erste Testreihe

5.2 Zweite Testreihe

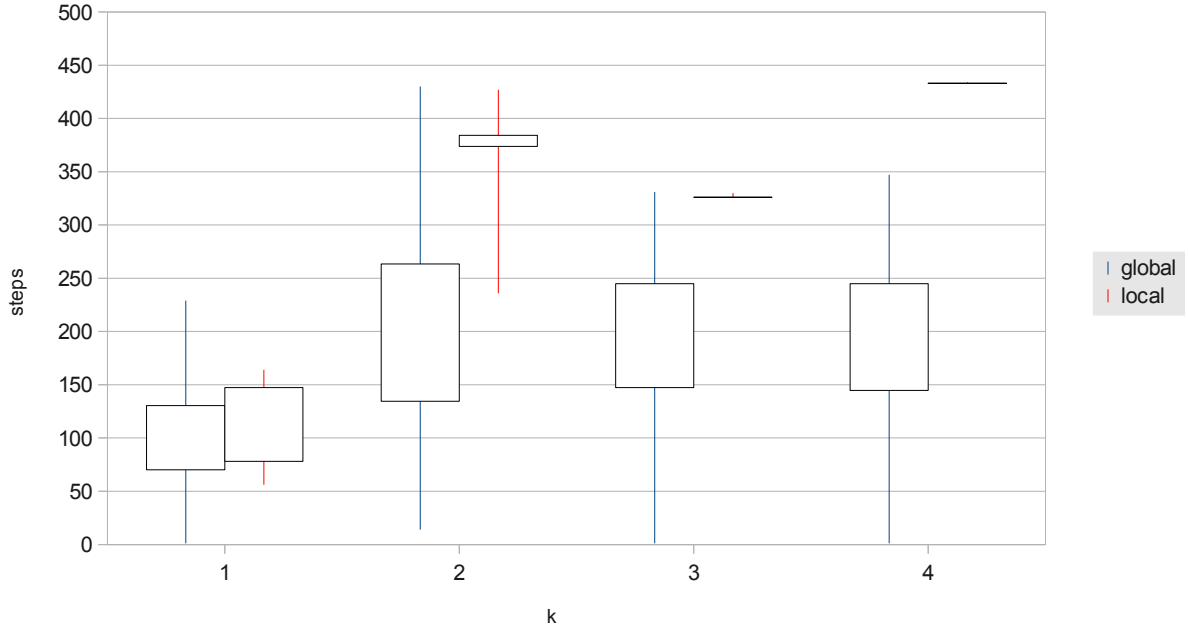
Das Konvergenzverhalten des gegenseitigen Ausschlusses soll in dieser Testreihe weiter untersucht werden. Dazu wurde ein verteiltes System mit höhere Komplexität wie in Abbildung 5.2 (b) gewählt. Die Topologie dieser Testreihe ist höher vermascht als die Topologie der vorigen Testreihe. Der Prozess P_6 umfasst mit seinen Nachbarn bereits ein Großteil des Systems. Bei $\kappa = 2$ besteht die κ-lokale Nachbarschaft von P_6 aus dem gesamten verteilten System. Die κ-lokale Nachbarschaft von jedem Prozess umfasst das Gesamtsystem bei $\kappa = 3$.
Im Gegensatz zur vorigen Topologie zeigt der lokale gegenseitige Ausschluss in dieser Konfiguration stets höhere Konvergenzzeiten als der gegenseitige Ausschluss. Unter $\kappa = 1$ ist in dieser Topologie zudem eine stärkere Schwankung als bei der vorigen Topologie unter lokalem gegenseitigen Ausschluss zu beobachten. Auffällig ist weiterhin, dass beide Ausschlussverfahren unter $\kappa = 2$ eine besonders schlechte Leistung zeigen. Der globale gegenseitige Ausschluss verzeichnet hier sein schlechtestes Ergebnis mit maximaler Durchschnittszeit und höchstem Maximum. Auch der lokale gegenseitige Ausschluss zeigt eine hohe Bandbreite an Ausreißern, die Abweichung ist aber eher gering. Während der globale gegenseitige Ausschluss unter größerem κ nahezu gleiche Konvergenzzeiten hat, wächst der Zeitraum zur Konvergenz unter lokalem gegenseitigen Ausschluss linear zu κ weiter.
Im folgenden sollen die Konvergenzzeiten interpretiert und mit der ersten Testreihe verglichen werden. Der globale gegenseitige Ausschluss zeigt mit Ausnahme von $\kappa = 1$ ein relativen konstanten Mittelwert der Konvergenzzeit. Es kann wieder angenommen werden, dass der Anwendungsalgorithmus bei $\kappa = 1$ konvergiert bevor die Hilfsalgorithmen konvergieren. Ab κ kann eine Konvergenz des Anwendungsalgorithmusses in der Regel erst unter konvergierenden Hilfsalgorithmen erfolgen. Da der κ-lokale Anwendungsalgorithmus danach aber relativ unabhängig von κ konvergiert, liegen die Konvergenzzeiten für den gegenseitigen Ausschluss im gleichen Bereich. Die Ausschläge für minimale und maximale Konvergenzzeiten bestehen weiterhin und sind durch eine frühzeitige Konvergenz unter inkonsistenten Hilfsalgorithmen zu erklären.
Der lokale gegenseitige Ausschluss zeigt unter dieser Topologie eine sehr schlechte Leistung. Lediglich unter $\kappa = 1$ kann er mit dem globalen gegenseitigen Ausschluss konkurrieren. Die lineare Proportionalität des lokalen gegenseitigen Ausschlusses hängt wie bereits in der ersten Testreihe mit dem Guards $convergestep$ und $reset$ des Algorithmusses zusammen. Stellt der verteilte Algorithmus Inkonsistenz fest, wird über ein von κ abhängigen Zeitraum erneut Synchronität hergestellt. Der Ausreißer bei $\kappa = 2$ zeigt allerdings, dass die Konvergenz unter ungünstigen Umständen sich auch sehr viel schlechter verhalten kann. Bemerkenswert ist weiterhin, dass die Konvergenzzeit für $\kappa = 4$ höher liegt als bei $\kappa = 3$ obwohl bereits dort die κ-lokale Nachbarschaft aller Prozesse maximal ist. Dieser Sachverhalt erklärt sich aus der direkten Abhängigkeit von $alpha$ zu κ.
Wie bereits in der ersten Testreihe dominiert der globale gegenseitige Ausschluss. Allerdings wurde eine Schwäche der Berechnung von α in der Implementierung des Algorithmusses vom lokalen gegenseitigen Ausschluss aufgedeckt. Statt von κ sollte α von der Größe der minimalen κ-lokale Nachbarschaft abhängen.

	κ			
	1	2	3	4
lokal				
ø	112,73	379,00	326,01	433,00
σ	34,62	5,23	0,13	0,03
min	56	236	326	433
max	164	427	330	434
global				
ø	100,26	198,98	196,04	194,72
σ	30,18	64,43	48,83	50,08
min	1	14	1	1
max	229	430	331	347

(a) Rechenschritte der Simulationsläufe



(b) Topologie des verteilten Systems



(c) Visualisierung der Konvergenzzeiten

Abbildung 5.2: Zweite Testreihe

6 Zusammenfassung und Ausblick

In dieser Abschlussarbeit wurde eine Simulationsumgebung zur Analyse des Konvergenz- und Laufzeitverhaltens von transformierten selbststabilisierenden verteilten Algorithmen entworfen. Für die Eingabe von Anwendungsalgorithmen wurde eine GCL-ähnliche Sprache implementiert. Des Weiteren können diverse Parameter für Simulationsläufe konfiguriert werden. Eine grafische Oberfläche ermöglicht das interaktive Erkunden und eignet sich besonders zu Präsentationszwecken. Damit eignet sich die Simulationsumgebung sehr gut zur Klassifizierung von Hilfsalgorithmen der Transformation und zur Beobachtung vom Konvergenzverhalten unter sich ändernden Umständen.

Schließlich wurden exemplarisch mehrere Simulationsläufe mit der Simulationsumgebung durchgeführt und vorgestellt. Neben Aussagen zum generischen Konvergenzverhalten der Hilfsalgorithmen konnten weitergehende Kenntnisse zur Optimierung dieser gewonnen werden. Es wurden Zusammenhänge zwischen der Topologie und Anwendungsvariablen zum Konvergenzzeitraum festgestellt.

Die Simulationsumgebung bietet darüber hinaus noch viele Möglichkeiten für die Zukunft. So kann durch Eingabe neuer Hilfsalgorithmen weitere Kompositionen von Hilfsalgorithmen erkundet werden. Mittelfristig kann durch Einsatz der Simulationsumgebung eine Klassifizierung aller Hilfsalgorithmen erfolgen, sodass für jedes Szenario eine Komposition gewählt wird, welche den Anforderungen wie geringer Konvergenzzeit gerecht wird. Weiterhin kann mit der Simulationsumgebung erforscht werden, welche Modifikationen an Algorithmen die Selbststabilisierung beeinflussen. Außerdem könnte eine Simulation flüchtiger Fehler in der interaktiven grafischen Oberfläche implementiert werden. Weitere Perspektiven zur Verifikation könnten durch eine weiteres Übersetzungsmodul von GCL zu einer Sprache der theoretischen Verifikation aufgetan werden. Langfristig stellt die Simulation und Klassifizierung der Hilfsalgorithmen zur Transformation und Komposition den ersten Schritt für dynamisch adaptive Algorithmen da, die sich unter wechselnden Systembedingungen während der Laufzeit anpassen um stets optimale Leistung zu erbringen.

Literaturverzeichnis

[AKY97] AFEK, Yehuda ; KUTTEN, Shay ; YUNG, Moti: The local detection paradigm and its applications to self-stabilization. In: *Theor. Comput. Sci.* 186 (1997), October, 199–229. http://dx.doi.org/10.1016/S0304-3975(96)00286-1. – DOI 10.1016/S0304–3975(96)00286–1. – ISSN 0304–3975

[BP08] BOULINIER, Christian ; PETIT, Franck: Self-stabilizing wavelets and ρ-hops coordination. In: *IPDPS*, 2008, S. 1–8

[Dha11] DHAMA, Abhishek: *A Compositional Framework for Designing Self-Stabilizing Algorithms*, Carl-von-Ossietzky University of Oldenburg, Diss., 2011. – Under Review

[Dij74] DIJKSTRA, Edsger W.: Self-stabilizing systems in spite of distributed control. In: *Commun. ACM* 17 (1974), November, 643–644. http://dx.doi.org/http://doi.acm.org/10.1145/361179.361202. – DOI http://doi.acm.org/10.1145/361179.361202. – ISSN 0001–0782

[Dij75] DIJKSTRA, Edsger W.: Guarded commands, non-determinacy and a calculus for the derivation of programs. In: *SIGPLAN Not.* 10 (1975), April, 2–2.13. http://dx.doi.org/http://doi.acm.org/10.1145/390016.808417. – DOI http://doi.acm.org/10.1145/390016.808417. – ISSN 0362–1340

[DIM93] DOLEV, Shlomi ; ISRAELI, Amos ; MORAN, Shlomo: Self-stabilization of dynamic systems assuming only read/write atomicity. In: *Distrib. Comput.* 7 (1993), November, 3–16. http://dx.doi.org/10.1007/BF02278851. – DOI 10.1007/BF02278851. – ISSN 0178–2770

[Dol00] DOLEV, Shlomi: *Self-stabilization*. Cambridge, MA, USA : MIT Press, 2000. – ISBN 0–262–04178–2

[DOT06] DHAMA, Abhishek ; OEHLERKING, Jens ; THEEL, Oliver: Verification of Orbitally Self-Stabilizing Distributed Algorithms Using Lyapunov Functions and Poincare Maps. In: *Proceedings of the 12th International Conference on Parallel and Distributed Systems - Volume 1*. IEEE Computer Society (ICPADS '06). – ISBN 0–7695–2612–8, 23–30

[DT10] DHAMA, Abhishek ; THEEL, Oliver: A tranformational approach for designing scheduler-oblivious self-stabilizing algorithms. In: *Proceedings of the 12th international conference on Stabilization, safety, and security of distributed systems*. Springer-Verlag (SSS'10). – ISBN 3–642–16022–0, 978–3–642–16022–6, 80–95

[Gär99] GÄRTNER, Felix C.: Fundamentals of fault-tolerant distributed computing in asynchronous environments. In: *ACM Comput. Surv.* 31 (1999), March, 1–26. http://dx.doi.org/http://doi.acm.org/10.1145/311531.311532. – DOI http://doi.acm.org/10.1145/311531.311532. – ISSN 0360–0300

[Jal94] JALOTE, Pankaj: *Fault tolerance in distributed systems*. Upper Saddle River, NJ, USA : Prentice-Hall, Inc., 1994. – ISBN 0–13–301367–7

[Rub] *Rubinius : Use Ruby™*

[Rub03] *About Ruby*. `http://www.ruby-lang.org/en/about/`. Version: 2003

[Rub10] *Rubinius vs. the benchmark from hell*. `https://rfc2616.wordpress.com/2010/10/16/rubinius-vs-the-benchmark-from-hell/`. Version: October 2010

[Sch93] SCHNEIDER, Marco: Self-stabilization. In: *ACM Comput. Surv.* 25 (1993), March, 45–67. `http://dx.doi.org/http://doi.acm.org/10.1145/151254.151256`. – DOI http://doi.acm.org/10.1145/151254.151256. – ISSN 0360–0300

[Sch07] SCHUSTER, Werner: *The Futures of Ruby Threading*. `http://www.infoq.com/news/2007/05/ruby-threading-futures`. Version: May 2007

[T10] *T-106.420 Concurrent Programming - Liveness, locks, critical sections*. `http://www.cs.hut.fi/Studies/T-106.420/Lectures/CP_locks.pdf`