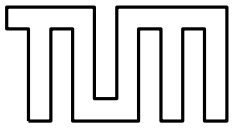


TECHNISCHE
UNIVERSITÄT
MÜNCHEN

INSTITUT FÜR INFORMATIK

Hierarchische Spezifikation und Korrektheitsbeweis eines parallelen Garbage Collectors

Diplomarbeit
von
Hans Stadtherr

Aufgabensteller/Betreuer:
Prof. Dr. W. Reisig

Abgabetermin:
15.5.91

Ich versichere, daß ich die vorliegende Diplomarbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Vorwort

Im Februar 1990 machte mich Taras Sajuk auf die objekt-orientierte Programmiersprache *Eiffel* aufmerksam, die Bertrand Meyer in seinem Buch "Object-oriented Software Construction" vorstellt. Abgesehen von der spielerischen Leichtigkeit, mit der er komplizierte Details aus der objekt-orientierten Software-Entwicklung präsentiert, faszinierte mich in [MEYE88] vor allem das Kapitel über die Speicherverwaltung in Eiffel. Meyer berichtet dort, daß das Laufzeit-System der aktuellen Implementierung von Eiffel über einen *parallelen* Garbage Collector verfügt. Ich war sofort begeistert, da ich mir damals nicht vorstellen konnte, wie ein solcher Algorithmus wohl aussehen würde.

Einige Wochen später wandte ich mich an Wolfgang Reisig, von dem ich wußte, daß er Diplomanden sucht, die Fallbeispiele erarbeiten, um eine neue Beweislogik für Petrinetze zu untersuchen. So entstand die Idee zur vorliegenden Arbeit: ein paralleler Garbage Collector sollte als Petrinetz modelliert und formal verifiziert werden.

Obwohl ich die volle Verantwortung für die (verbleibenden) Fehler in dieser Arbeit übernehme, so möchte ich mich trotzdem bei allen bedanken, die mir geholfen haben. Besonderen Dank schulde ich Rolf Walter, dessen Betreuung und Hilfe mir sehr willkommen war, sowie meiner Schwester Christine, die mich durch Literatur-Recherchen unterstützte. Wolfgang Reisig brachte mich durch seine Vorschläge immer wieder auf den richtigen Weg. Ekkart Kindler danke ich für zahlreiche Korrekturen und Anmerkungen zu einer früheren Version dieser Arbeit. Barbara Paech machte mich auf Probleme bei der Netzkomposition aufmerksam.

München, im Mai 1991 — Hans Stadtherr

Inhaltsverzeichnis

Einleitung **1**

TEIL 1 Einführung und Hintergrund

1 Parallele Garbage Collectoren: Arbeitsweise und Problematik **3**
1.1 Speicherverwaltung als Grundlage der EDV 3
1.2 Automatische Wiedergewinnung von Speicherbereichen 4
1.3 On-the-fly Garbage Collection 6
1.4 Das Konsistenzproblem in nebenläufigen Garbage Collectoren 7
1.5 Lokale Erkennung eines globalen Zustands 9
1.6 Garbage Collection “on-the-fly” als reaktives System 10

2 On-the-fly Garbage Collection in der Literatur **13**
2.1 G.L. Steele 13
2.2 E.W. Dijkstra et al. 14
2.3 D. Gries 19
2.4 H.T. Kung und S.W. Song 23
2.5 S. Ramesh und S.L. Mehndiratta 24
2.6 M. Ben-Ari 27
2.7 K.M. Chandy und J. Misra 28
2.8 Weiterführende Literatur 31

TEIL 2 Theoretische Grundlagen

3 Systemmodellierung mit Petrinetzen **35**
3.1 EN-Systeme 35
3.2 Abstrakte Datentypen 39
3.3 Multimengen-Spezifikationen 43
3.4 Netze mit strukturierten Marken 46
3.5 S-Invarianten und verwandte Netzeigenschaften 48
3.6 Abläufe und Semantik von beschrifteten Netzen 52

4 Systemanalyse und -Spezifikation durch eine temporale Logik **55**
4.1 Eine Sprache für Eigenschaften von beschrifteten Netzen 55
4.2 Unless 60
4.3 Ensures 64

4.4 Leads-to 66
4.5 Fair 71

5 Systemstrukturierung durch Komposition 73
5.1 Netzkomposition durch Vereinigung 73
5.2 Eigenschaften komponierter Netze 74
5.3 Das Substitutions-Axiom 76

TEIL 3 Ein paralleler Garbage Collector als Petrinetz

6 Problemanalyse 79
6.1 Aufgabenstellung 79
6.2 Abstraktionen 79
6.3 Beschreibung des Mutators 81
6.4 Beschreibung des Collectors 81

7 Implementierung 85
7.1 Ein sequentieller Collector 85
7.2 Ein zum Mutator nebenläufiger Collector 91
7.3 Analyse durch S-Invarianten 92

TEIL 4 Hierarchische Spezifikation und Korrektheitsbeweis

8 Beweisstruktur 95
8.1 Horizontale Beweisstruktur 95
8.2 Vertikale Beweisstruktur 97

9 Formale Spezifikation des dynamischen Verhaltens 99
9.1 Spezifikation des parallelen Garbage Collectors 99
9.2 Spezifikation des Gesamtsystems 100

10 Verfeinerung der Systemspezifikation 105
10.1 Nachweis der Lebendigkeit 105
10.2 Der Collector-Zyklus als Grundlage der Lebendigkeit 110
10.3 Nachweis der Sicherheit 117
10.4 Abstützung der Lebendigkeit durch elementare Eigenschaften 123

11 Aufteilung der Systemspezifikation 133
11.1 Spezifikation des Mutators 133
11.2 Terminale Spezifikation des Collectors 134
11.3 Abstützung des Systems auf die Komposition von Mutator und Collector 138

TEIL 5 Schluß und Anhang

12 Zusammenfassung **153**

Literaturverzeichnis **155**

Einleitung

Mit wachsender Komplexität der informationstechnischen Aufgaben, die durch Rechner bewältigt werden sollen, stellt sich zunehmend das Problem der Korrektheit von Algorithmen. Während es einerseits allgemein üblich ist, Programme jeder Größenordnung zu entwerfen und in Betrieb zu nehmen, wird andererseits über die Funktionstüchtigkeit solcher Software-Systeme eher pragmatisch argumentiert: je länger das Programm im Einsatz ist, desto mehr Fehler werden gefunden (und ausgebessert) und desto "korrekter" wird die Software. Diese Vorgehensweise ist äußerst risikoreich. Im Bereich verteilter (paralleler) Programme stellt sich die Situation analog, aber mit deutlich schwierigeren Problemen dar.

Um die Korrektheit eines Programms zu beweisen, ist es erforderlich, daß die Anforderungen, die an den Algorithmus zu stellen sind, geeignet formuliert werden, um eine Überprüfung zu ermöglichen. Es reicht daher nicht aus, die geforderten Eigenschaften umgangssprachlich festzuhalten. Benötigt wird eine formale Notation, die für Spezifikation und Verifikation eine homogene Logik bereitstellt. Auch die Implementierungssprache muß auf einem formalen Kalkül basieren und mit der Spezifikationssprache harmonieren. In dieser Arbeit wird am Beispiel eines parallelen Garbage Collectors aufgezeigt, daß eine formale Spezifikation und der Beweis der Korrektheit notwendige Bestandteile der Software-Entwicklung sind, ohne die ein paralleles oder verteiltes System nicht handhabbar ist.

> Ziel der vorliegenden Arbeit ist es, einen bekannten Algorithmus für einen parallelen Garbage Collector als Petrinetz zu modellieren und die Korrektheit der Modellierung gegenüber einer gegebenen Spezifikation formal nachzuweisen.

Als Implementierungssprache wurden Petrinetze ausgewählt, da sie ein anschauliches Werkzeug zur Modellierung verteilter Systeme sind. Darüberhinaus gibt es wertvolle Methoden zur Überprüfung von Netzeigenschaften (z.B. S-Invarianten). Das in dieser Arbeit benutzte Netzkalkül stammt aus [REIS90a]. Als Spezifikationssprache und Beweislogik wird eine Form der temporalen Logik benutzt, die von K.M. Chandy und J. Misra entwickelt wurde und mit der in [CHAN88] die Korrektheit von UNITY-Programmen bewiesen wird.

Vorbild meiner Modellierung ist der in [DIJK76] vorgestellte DLMSS-Algorithmus (benannt nach den Anfangsbuchstaben der Autoren), der Grundlage vieler weiterer Arbeiten auf diesem Gebiet wurde. Tatsächlich gibt es zum Thema "On-the-fly Garbage Collection" nur wenige Arbeiten, deren algorithmische Idee sich wesentlich von der in [DIJK76] unterscheidet. Zwei Gründe machen einen nebenläufigen Garbage Collector zu einem interessanten Kandidaten für theoretische Untersuchungen:

- Speicherbereinigung und -Wiedergewinnung sind grundlegend für die maschinelle Informationsverarbeitung. Eine Verbesserung der verwendeten Algorithmen wird daher in vielen Bereichen der Informatik Auswirkungen zeigen. Ein tiefergehendes Verständnis für die Problematik ist Voraussetzung für die Verbesserung der Algorithmen.
- Die möglichen Abläufe in einem System mit parallelem Garbage Collector sind vielschichtig und kompliziert. Mordechai Ben-Ari bemerkt dazu: “This algorithm (henceforth called the DLMSS algorithm) is one of the most difficult concurrent programs ever studied.” [BENA84, S. 333]. Ein solcher Algorithmus stellt daher eine besondere Herausforderung für die Erprobung einer neuen Methodik zur Modellierung und Verifikation paralleler Systeme dar.

Im ersten Teil der vorliegenden Arbeit werden Ziele und Probleme erörtert, die sich im Zusammenhang mit der nebenläufigen Wiedergewinnung von Speicherbereichen ergeben. Ein Exkurs in die geschichtliche Entwicklung der parallelen Garbage Collectoren und eine Übersicht über die bisherigen Veröffentlichungen zum Thema “On-the-fly Garbage Collection” runden den einführenden Teil ab.

Im zweiten Teil stelle ich die grundsätzlichen Konzepte vor, auf denen die Implementierung und der Beweis ihrer Korrektheit basieren. Dazu zählen ein Kalkül für Netze mit strukturierten Marken zur Systemmodellierung und eine temporale Logik zur Spezifikation und Verifikation von Verhaltenseigenschaften solcher Systeme (Netze). Wir betrachten insbesondere die Komposition von Teilnetzen und den Zusammenhang zwischen Eigenschaften der Komponenten und den Eigenschaften des Gesamtsystems.

Thema des dritten Teils dieser Arbeit ist die Modellierung eines parallelen Garbage Collectors als Petrinetz. Im vierten Teil wird diese formal verifiziert.

Der fünfte und letzte Teil faßt die Ergebnisse der Arbeit zusammen und versucht durch eine kritische Analyse der verwendeten Methoden einen Ausblick auf zukünftige Forschungsziele zu geben.

Der Schwerpunkt dieser Arbeit liegt im dritten und vierten Teil. Dort versuchen wir durch ein aussagekräftiges Fallbeispiel nachzuweisen, daß die Kombination von Petrinetzen mit temporaler Logik eine brauchbare und anschauliche Methode liefert, um parallele Systeme zu modellieren und ihre Korrektheit formal zu beweisen. Eine überzeugende Begründung für die Notwendigkeit, parallele Algorithmen formal zu überprüfen, können wir dem Schlußwort von [DIJK76] entnehmen, in dem Dijkstra et al. über die Schwierigkeiten berichten, mit denen sie beim Entwurf des DLMSS-Algorithmus zu kämpfen hatten:

> “It has been surprisingly hard to find the published solution and justification. It was only too easy to design what looked — sometimes even for weeks and to many people — like a perfectly valid solution, until the effort to prove it to be correct revealed a (sometimes deep) bug. (...) One firm conclusion, however, can be drawn: to believe that such solutions can be found without a very careful justification is optimism on the verge of foolishness.”

TEIL 1

Einführung und Hintergrund

Der erste Teil dieser Arbeit bettet die mehr formalen Darstellungen der restlichen Kapitel in einen anschaulichen Rahmen ein. Wir beginnen mit einer kurzen Einführung in die Speicherverwaltung und stellen dann die Funktionsprinzipien eines nebenläufigen Garbage Collectors vor. Unser Augenmerk legen wir dabei weniger auf die algorithmischen Details als auf die Darstellung des Charakters des zugrundeliegenden Verfahrens. Ein daran anschließender Überblick über die bisherigen Veröffentlichungen zum Thema “nebenläufige Speicherbereinigung” soll dem Leser Hintergrund-Informationen vermitteln.

1 Parallele Garbage Collectoren: Arbeitsweise und Problematik

1.1 Speicherverwaltung als Grundlage der EDV

Viele Aufgaben, die von Rechnern bearbeitet werden, erfordern, daß der Speicher, der zur Datenhaltung benutzt wird, erst zur Laufzeit des Programms dynamisch vom Betriebssystem angefordert wird. Meistens ist es nämlich unmöglich, bereits vor dem Programmstart festzustellen, welche Datenmengen zu bearbeiten sind. Wäre das Datenvolumen bei einem funktionalen Programm noch abschätzbar, z.B. durch Betrachtung der Eingabedaten, so kann darüber in Programmen, die mit dem Benutzer interagieren, a priori keine Aussage gemacht werden. Auch im Sinne einer wirtschaftlichen Nutzung des Speichers in Mehrbenutzeranlagen ist eine dynamische Verwaltung des Speicherplatzes wünschenswert. Die eigentliche Entstehung des Problems der Speicherverwaltung resultiert aber erst aus der Tatsache, daß während eines Programmlaufs der Speicherbedarf großen Schwankungen unterliegen kann. Dabei ergeben sich zwei Teilprobleme:

- Wie wird erkannt, ob ein Speicherblock noch benutzt wird, oder ob er bereits freigegeben werden kann?

- Wie werden die freigegebenen Speicherblöcke verwaltet, um sie möglichst vollständig wiederverwendbar zu machen?

Der zweite Punkt ist nicht Gegenstand meiner Arbeit und wird nur der Vollständigkeit halber erwähnt. In einigen LISP-Systemen wird dieses Problem durch Speicherblöcke konstanter Größe gelöst. Weil alle Blöcke gleich groß sind, entsteht keine Fragmentierung im Freispeicher und die Wiederverwendbarkeit wird gewährleistet. Der Nachteil ist, daß diese Methode den verwendbaren Datenstrukturen Beschränkungen auferlegt. Alle anderen Lösungen, die Speicherblöcke variabler Größe zulassen, arbeiten mit mehr oder weniger aufwendigen Heuristiken und unterschliedlichem Erfolg [TANE87].

Der erste Punkt wird in prozeduralen Programmiersprachen üblicherweise vernachlässigt, d.h. die Verantwortung über die Freigabe von Speicherblöcken wird dem Algorithmus — und damit dem Programmierer — übertragen. Daß diese Verfahrensweise nicht nur in großen oder verteilten Applikationen unüberschaubare Risiken birgt, ist unumstritten. In funktionalen und objekt-orientierten Sprachen ist es jedoch bereits seit langem üblich, die Erkennung unbenutzter Speicherblöcke automatisch durchführen zu lassen [STOY86, S.1; BUDD87, S. 144; GOLD89, S. 441].

1.2 Automatische Wiedergewinnung von Speicherbereichen

Eine Speicherzelle ist eine Einheit variabler oder konstanter Größe, die atomare Werte und/oder eine Menge von Zeigern auf andere Speicherzellen trägt. Sie befindet sich in einem von drei möglichen Zuständen:

- Sie wird in der Liste der freien Speicherzellen aufbewahrt.
- Sie ist eingebunden in die Datenstruktur eines Prozesses.
- Sie ist weder in der Liste der freien Speicherzellen noch ist sie eingebunden in die Datenstruktur eines Prozesses. Sie wird als *Müll* (engl. “Garbage”) bezeichnet.

Unter einer Speicherzelle verstehen wir also (in diesem Zusammenhang) einen Speicherblock, der dynamisch während der Laufzeit eines Programms vom Betriebssystem angefordert wird (z.B. durch die Prozedur *malloc*() in der Programmiersprache C). Der statische Speicherbereich eines Prozesses (z.B. der Stack) wird als spezielle Speicherzelle aufgefaßt, die als “Aufhängung” für die dynamisch angeforderten Speicherzellen dient und ständig zur Datenstruktur des Prozesses gehört. Wir bezeichnen sie auch als *Wurzel*.

Die atomaren Werte, die eine Speicherzelle tragen kann, sind ohne Bedeutung für das Zusammenspiel mit der automatischen Speicherwiedergewinnung. Wir konzentrieren uns deshalb im weiteren nur auf die Zeiger der Speicherzellen. Die Menge aller Zeiger der benutzten Zellen bezeichnen wir auch als Verzeigerung der Datenstruktur des Prozesses. Auf einem geeigneten Abstraktionsniveau stellen sich für uns die Vorgänge im Speicher folgendermaßen dar: Der Prozeß kann seine Datenstruktur verändern, indem er Zeiger aus einer Speicherzelle löscht oder neue hinzufügt. Das Ziel eines neuen Zeigers ist ent-

weder eine Speicherzelle der alten Datenstruktur oder eine, die soeben aus der Liste der freien Speicherzellen entnommen wurde. So kann der Prozeß neue Speicherzellen in seine Datenstruktur einbinden. Alle Speicherzellen einer Datenstruktur sind von der Wurzel aus durch eine Folge von Zeigern erreichbar. Wird ein Zeiger gelöscht, so kann die Speicherzelle, auf die er gerichtet war, unter Umständen zu Müll werden, nämlich dann, wenn die Zelle durch keinen Pfad mehr von der Wurzel aus erreichbar ist. Der Prozeß kann keine neuen Zeiger auf Speicherzellen richten, die bereits Müll sind. Um von einer gegebenen Speicherzelle aus zu einer ihrer Nachfolgerzellen zu gelangen, reicht eine atomare und primitive Operation aus. Um jedoch die Vorgängerzellen zu erreichen, muß die ganze Datenstruktur durchsucht werden. Das Problem der Wiedergewinnung von Speicher kann damit wie folgt definiert werden:

> Jede Speicherzelle, die Müll ist, soll nach endlicher Zeit automatisch erkannt und an die Liste der freien Speicherzellen angehängt werden.

Für die Durchführung dieser Aufgabe kommen im wesentlichen zwei verschiedene Verfahren in Frage [COHE81]:

- *reference counting*: Jedes Speicherobjekt[1] erhält einen Zähler, mit dessen Hilfe die Anzahl der Referenzen, die auf dieses Objekt zeigen, gespeichert wird. Jedesmal, wenn eine Referenz erzeugt oder gelöscht wird, muß der Zähler des Zielobjekts um eins erhöht bzw. erniedrigt werden. Ergibt sich ein Zähler zu null, kann das Objekt freigegeben werden. Dieses Verfahren ist inkrementell und verteilt den Zeitaufwand, der für die Wiedergewinnung von Speicher benötigt wird, gleichmäßig auf den Programmablauf. Ein Nachteil ist, daß unter Umständen Speicherzellen, die sich in zirkulären Datenstrukturen befinden, vom Collector nicht als Müll erkannt werden. Wir werden uns mit diesem Verfahren nicht weiter beschäftigen.

- *mark and sweep*: Nur solche Objekte, die von der Wurzel aus durch eine Folge von Zeigern erreichbar sind, gehören zur Datenstruktur des betreffenden Prozesses. Daraus ergibt sich das folgende Verfahren: Ausgehend von der Wurzel des Speichergeflechts[2], werden alle erreichbaren Objekte mit einer besonderen Markierung versehen. Zu diesem Zweck wird das Speichergeflecht *traversiert* und jedes von der Wurzel aus erreichbare Objekt einmal aufgesucht. Anschließend werden alle Objekte, die keine Markierung tragen, freigegeben, da sie nicht mehr zur Datenstruktur gehören.

Wird beim letzten Verfahren ein sequentieller Collector benutzt, dann muß das Programm gelegentlich angehalten werden (vor allem dann, wenn der Speicher knapp wird), um das Speichergeflecht zu traversieren. Anschließend werden die Markierungen wieder gelöscht und das Programm kann fortfahren. Ein solcher Algorithmus hat den Nachteil,

---

1) Wir benutzen den Begriff “Speicherobjekt” als Synonym für “Speicherzelle”.
2) Der Begriff “Speichergeflecht” steht für alle Zellen und alle Zeiger im Speicher, während der Begriff “Datenstruktur” nur den von der Wurzel aus erreichbaren Teil des Speichergeflechts bezeichnet.

daß Lücken im Programmablauf entstehen, deren Länge mit der Größe der Datenstruktur wächst. Das Verfahren ist somit ungeeignet für Applikationen, in denen Anforderungen an die Antwortzeiten des Programms gestellt werden.

1.3 On-the-fly Garbage Collection

Unter dem Begriff "On-the-fly Garbage Collector" verstehen wir einen Algorithmus, der die Wiedergewinnung unbenutzter Speicherzellen nach dem mark-and-sweep-Verfahren *parallel* zum Hauptprogramm ermöglicht. Der Garbage Collector ist dann ein eigener Prozeß und kann sogar auf einem eigenen Prozessor ablaufen. Die Vorteile dieses Verfahrens sind:

- Der Ablauf des Hauptprogramms muß nicht mehr unterbrochen werden. Die Reaktionszeit eines interaktiven Programms wird vorhersagbar.
- Wenn der Prozeß des nebenläufigen Garbage Collectors auf einen zweiten Prozessor ausgelagert wird, ergibt sich eine signifikante Steigerung der Arbeitsgeschwindigkeit (beider Prozesse) [HICK84].

Die Idee, die Speicherbereinigung einem zweiten Prozeß zu übertragen, der parallel zum eigentlichen Programm abläuft, stammt von M. Minsky (siehe [KNUT73], Seite 422 und 594). Der erste Algorithmus wurde aber erst 1975 von G.L. Steele vorgestellt und hatte den Nachteil, daß Semaphore benötigt wurden [STEE75]. Ein anderer Algorithmus, bei dem die beteiligten Prozesse weitgehend unabhängig voneinander arbeiten, wurde 1976 von Dijkstra et al. veröffentlicht [DIJK76]. In dem dort beschriebenen *DLMSS-Algorithmus* wird das System in die beiden Prozesse *Mutator* und *Collector* aufgeteilt, wobei der Collector auf der Datenstruktur des Mutators operieren darf. Während der Mutator — dem eigentlichen Programm folgend — die Verzeigerung seiner Datenstruktur manipuliert (d.h. Zeiger erzeugt oder Zeiger löscht), versucht der Collector durch das mark-and-sweep-Verfahren benutzte Speicherzellen zu markieren. Jede Zelle wird durch eine der drei Farben *schwarz*, *weiß* oder *grau* markiert. Der Collector befindet sich immer in einer von zwei Phasen:

- Zu Beginn der *Markierungsphase* (mark-phase) haben alle Objekte die Farbe weiß. Die Markierungsphase wird gestartet, indem der Collector die Wurzel grau markiert. Anschließend wird die Farbe grau entlang der Verzeigerung propagiert, d.h. jede graue Speicherzelle gibt die eigene Farbmarkierung an ihre Söhne weiter und färbt sich dann selbst schwarz. Ein Sohn, der bereits schwarz markiert ist, bleibt schwarz. Die Markierungsphase ist beendet, sobald es keine grauen Zellen mehr gibt. Dann sind alle erreichbaren Speicherobjekte schwarz und der Müll ist immer noch weiß.
- In der *Bereinigungsphase* (sweep-phase) werden alle weißen Speicherzellen freigegeben und die schwarzen Zellen werden wieder gebleicht, d.h. weiß gefärbt.

Um zu gewährleisten, daß der Collector auch wirklich nur *Müll* freigibt, muß der Mutator den Collector unterstützen: alle weißen Speicherzellen, zu denen er Zeiger errichtet, muß er selbständig *grau* markieren. Dieser Algorithmus stellt sicher, daß jede nicht mehr erreichbare Zelle nach spätestens einem Collector-Zyklus (mark-sweep) als solche erkannt wird. Müll, der während einer Markierungsphase entsteht, wird unter Umständen erst im nächsten Zyklus erkannt. Der Mehraufwand an Speicher beträgt 2 Bit je Speicherzelle und zwar zur Darstellung der Farbmarkierung.

Durch Einführung einer vierten Farbe (*off-white*) wird 1977 der DLMSS-Algorithmus wie folgt verbessert: bisher wurden alle freien Speicherzellen als erreichbar betrachtet und mußten deshalb in jeder Markierungsphase mitbehandelt werden. Kung und Song schlugen vor, die freien Zellen mit der Farbe off-white zu markieren und sie so aus der Traversierung auszuschließen [KUNG77a]. Zugleich werden in ihrem Algorithmus die grauen Speicherzellen in einer besonderen Liste gehalten. Diese Maßnahme beschleunigt zwar den Collector, fordert aber zusätzlichen Speicheraufwand bei der Organisation der Liste.

1984 schließlich stellt M. Ben-Ari eine Version vor, die mit nur zwei Farben auskommt: schwarz und weiß. Ausgehend von der Wurzel wird solange die Farbe schwarz entlang der Verzeigerung propagiert, bis kein neuer schwarzer Knoten[3] mehr entsteht. Erkannt wird diese Situation dadurch, daß jeweils zwischen zwei Propagierungsschleifen die Anzahl der schwarzen Knoten gezählt wird (eine Propagierungsschleife = alle Knoten werden einmal auf ihre Farbe getestet, wobei schwarze Knoten ihre Farbe an die Söhne weitergeben). Hat die Anzahl der schwarzen Knoten während einer Propagierungsschleife nicht zugenommen, dann ist die Markierungsphase beendet. Tatsächlich ist der Algorithmus von Ben-Ari weniger effizient als der DLMSS-Algorithmus, dafür läßt sich seine Korrektheit leichter beweisen [BENA84].

Eine ausführliche Diskussion dieser hier knapp gehaltenen Zusammenfassung der Entwicklung findet sich in Kapitel 2.

1.4 Das Konsistenzproblem in nebenläufigen Garbage Collectoren

Warum ist ein paralleler Garbage Collector aufwendiger als ein sequentieller? Um diese Frage zu beantworten, müssen wir die Aufgabe des Collectors analysieren. Für ihn stellt sich das Speichergeflecht als verteiltes System dar, da der Collector das Geflecht immer nur an einer Stelle beobachtet und sich die Datenstruktur zwischen seinen Beobachtungen verändern kann. Damit stellt sich ein Konsistenzproblem: Der Collector muß sich ein Bild vom globalen Zustand des Speichergraphen machen, seinen Blick kann er aber nur auf einen lokalen Ausschnitt richten. Während sein Blick über den Graphen schweift, fügt der Mutator neue Kanten ein, oder löscht alte (quasi hinter dem Rücken des Collec-

3) Wir abstrahieren von der Datenstruktur des Mutators und bezeichnen die Speicherzellen als Knoten und die Zeiger als Kanten. Die Begriffe *Knoten* und *Kanten* benutzen wir in Anlehnung an die Bestandteile eines gerichteten Graphen.

tors), was den bisherigen Kenntnisstand des Collectors veralten läßt. Der erkannte globale Zustand liefert nicht nur ein veraltetes, sondern sogar ein ungültiges Bild, d.h. ein im Ablauf nie existierender Zustand wird angenommen.

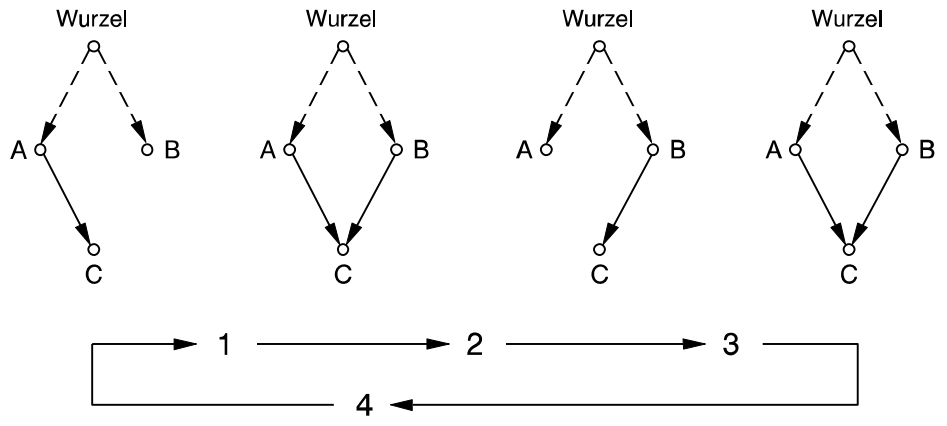


Abb. 1.1. Zum Konsistenzproblem

Abbildung 1.1 zeigt eine Sequenz von Situationen in der Datenstruktur des Mutators. Wir betrachten die drei Knoten A, B und C. Knoten A und B sind ständig durch feste Pfade von der Wurzel aus erreichbar, während der Knoten C nur durch eine Kante von A nach C erreicht werden kann. Der Mutator führt dann wiederholt die folgenden Operationen aus:

1. richte eine Kante von B nach C
2. lösche die Kante von A nach C
3. richte eine Kante von A nach C
4. lösche die Kante von B nach C

Der Collector, der immer nur einen Knoten auf einmal betrachten kann, wird feststellen, daß A und B erreichbar sind, wogegen er bei C getäuscht werden kann: während der Collector A beobachtet, ist C nur von B aus erreichbar (zwischen 2 und 3) und während er B beobachtet, ist C nur von A aus erreichbar (zwischen 4 und 1). Wir folgern daraus, daß es für einen nebenläufigen Collector unmöglich ist, ohne die Kooperation des Mutators auszukommen. Im DLMSS-Algorithmus besteht die Kooperation darin, daß der Mutator einen weißen Knoten, auf den er eine neue Kante richtet, grau markiert. In unserem Beispiel müßte also der Mutator bei 1 und bei 3 den Knoten C markieren, um dem Collector zu signalisieren, daß C von der Wurzel aus erreichbar ist. Durch diese Kooperation wird der Collector in die Lage versetzt, einen globalen Zustand im Graphen konsistent erfassen zu können. Gerade diese Kooperation und der damit verbundene Verwaltungsaufwand auf Seiten des Collectors beantwortet unsere eingangs gestellte Frage: ein neben-

läufiger Collector muß sich darum kümmern, daß das aktuelle Bild, das er sich vom Graphen macht, weitgehend konsistent ist, d.h. jeder Knoten, den der Collector für Müll hält, muß auch wirklich Müll sein. Andernfalls darf er dieses Bild nicht zur Grundlage der Entscheidung machen, einen Knoten als frei zu markieren[4]. In einem sequentiellen Collector entfällt dieser Aufwand.

Das Konsistenzproblem, wie wir es eben beschrieben haben, ist charakteristisch für verteilte Systeme, in denen es darum geht, globale Zustände durch lokale Betrachtungen zu erkennen. Eine weiterführende Beschreibung des Konsistenzproblems findet sich in [GOMM89] im Zusammenhang mit der Erkennung der verteilten Terminierung.

1.5 Lokale Erkennung eines globalen Zustands

Die Wiedergewinnung unbenutzter Speicherzellen stellt uns vor das Problem der lokalen Erkennung eines globalen Zustands, sofern Mutator und Collector nebenläufig arbeiten (wie eben erwähnt). Das Problem wird in einem sequentiellen Garbage Collector vollständig entschärft, da sich der globale Zustand des Speichergraphen während der Betrachtung nicht ändert. In einem parallelen Garbage Collector dagegen arbeitet der Prozeß, der den globalen Zustand verändert, gleichzeitig mit dem Prozeß, der einen bestimmten globalen Zustand erkennen soll.

Ein weiteres Merkmal von nebenläufigen Garbage Collectoren ist ihre Beziehung mit einem anderen Problem verteilter Systeme: der *Terminierungserkennung*. Auch hier richtet sich das Augenmerk auf die lokale Erkennung eines globalen Zustands, nämlich der Terminierung aller Prozesse im System. Tatsächlich gehört ein paralleler Garbage Collector mit seinem Graph-Markierer und der beschriebenen Kooperation mit dem Mutator, zu der Klasse von Systemen, die Gegenstand von Untersuchungen zur Terminierungserkennung sind. In einem parallelen Garbage Collector stellt sich das Problem der Terminierung bei der Markierungsphase: wie wird erkannt, daß es keine grauen Knoten mehr gibt?

G. Tel et al. zeigen in [TEL88] einen Zusammenhang auf zwischen parallelen Garbage Collectoren und der Erkennung der verteilten Terminierung. Sie transformieren bestehende Protokolle zur Erkennung der verteilten Terminierung in wenigen Schritten zu parallelen Garbage Collectoren. Das Transformationsschema lautet etwa folgendermaßen: Die Terminierungserkennung liefert einen Algorithmus zur Traversierung des Speichergraphen, indem sie auf Markierungsprozesse angewendet wird, wobei jeder Knoten des Graphen einen eigenen Prozeß darstellt. Empfängt ein weißer (d.h. unmarkierter) Knotenprozeß eine Aktivierungsnachricht, beginnt der Knoten, an alle Söhne Aktivierungsnachrichten zu verschicken. Der Knotenprozeß terminiert, nachdem er alle Nachrichten verschickt und sich selbst schwarz markiert hat. Um den Graph-Markierer zu starten, wird der Wurzel eine Aktivierungsnachricht geschickt. Die Traversierung des

4) Anstelle der Liste von freien Knoten können wir uns auch eine zusätzliche “Markierungsfarbe” vorstellen: jeder Knoten ist entweder frei (markiert) oder nicht.

Speichergraphen ist beendet, wenn alle Knotenprozesse terminiert sind und keine Aktivierungsnachrichten mehr unterwegs sind. Genau dieser Zustand wird vom Protokoll zur Erkennung der verteilten Terminierung erkannt. Alle Knoten, die danach noch weiß sind, können als frei markiert werden, da sie nicht mehr von der Wurzel aus erreichbar sind.

So kann in der Implementierung eines parallelen Garbage Collectors insofern von der normalen nicht-verteilten Datenstruktur eines gerichteten Graphen abstrahiert werden, als daß der Markierungsvorgang als Nachrichtenaustausch zwischen den Knoten verstanden werden kann und damit Speichergraph und Collector zu einem verteilten System werden.

1.6 Garbage Collection “on-the-fly” als reaktives System

In der klassischen funktionalen Informatik bestehen Programme aus einer Menge von Funktionen. Das Programm selbst ist ebenfalls eine Funktion. Es berechnet zu gegebenen Parametern nach endlicher Zeit ein Resultat und ist dann beendet. Viele der heutigen Probleme, die durch Rechner bewältigt werden, sind dagegen reaktiver Natur. Dabei steht nicht die Berechnung einer Funktion im Mittelpunkt, sondern die Bereitstellung von Diensten an Dienstnehmer [HARE85]. Zu dieser Klasse von Algorithmen zählen auch die parallelen Garbage Collectoren. Ihre “Dienstleistung” besteht darin, in dem Speichergeflecht des Mutators unerreichbare Knoten zu erkennen und als frei zu markieren. Wir sprechen in diesem Zusammenhang auch von zyklischen Systemen, da die Bereitschaft, Dienste anzubieten, nicht versiegen darf.

Reaktive Systeme unterscheiden sich von funktionalen Sytemen i.a. auch noch durch die Art und Weise, wie ihre Korrektheit nachgewiesen wird. Die Verifikation sequentieller funktionaler Algorithmen hat bereits eine lange Tradition. Man spricht von *partieller Korrektheit*, wenn der Nachweis gelingt, daß das Programm — sofern es terminiert — ein korrektes Ergebnis abliefert. *Totale Korrektheit* schließlich erfordert zusätzlich den Beweis, daß das Programm auch tatsächlich nach endlicher Zeit zum Stillstand kommt. Beide Begriffe lassen sich schlecht auf reaktive Systeme übertragen, da es weder ein Resultat gibt, noch wird ein Stillstand zu den wünschenswerten Eigenschaften eines reaktiven Systems gezählt. Statt dessen werden zwei adäquatere Bezeichnungen benutzt: Sicherheit und Lebendigkeit.

Sicherheitseigenschaften (savety properties) formulieren, daß bestimmte Ereignisse während des Ablaufs auf keinen Fall eintreten. *Lebendigkeitseigenschaften* (liveness properties) halten fest, welche Ereignisse ganz sicher irgendwann eintreten [OWIC82]. Der Zusammenhang zu den beiden oben genannten Begriffen stellt sich folgendermaßen dar: Partielle Korrektheit beinhaltet eine Sicherheitsanforderung in dem Sinne, daß, wenn die *Vorbedingung* (precondition) erfüllt wird, der Algorithmus niemals mit einer unerfüllten *Nachbedingung* (postcondition) anhält. Die Funktion und der Aufrufer der Funktion schließen gleichsam einen *Vertrag* ab, in dem sich der Aufrufer zur Erfüllung der Vorbedingung (z.B. Versorgung mit angemessenen Parametern) und die Funktion zur Erfüllung der Nachbedingung verpflichten (*programming by contract*: [MEYE88, S. 111-164]).

Totale Korrektheit beinhaltet zusätzlich eine Lebendigkeitseigenschaft, die ausdrückt, daß das Ereignis “*Funktion terminiert*” auch tatsächlich irgendwann eintritt. Die Sicherheits- und Lebendigkeitseigenschaften, die sich zu einem reaktiven System formulieren lassen, sind naturgemäß differenzierter und aufwendiger. Sicherheitseigenschaften sind z.B.:

- Keine *Deadlocks*: das System (oder ein Teil davon) nimmt unter keinen Umständen einen Zustand ein, in dem kein weiterer Fortschritt im Ablauf möglich ist.
- Gegenseitiger Ausschluß (*mutual exclusion*): zwei Prozesse befinden sich niemals zugleich in einem gemeinsamen kritischen Bereich.

Lebendigkeitseigenschaften können z.B. sein:

- Jede Anforderung an einen Dienst wird irgendwann bearbeitet.
- Eine Nachricht wird irgendwann beim Empfänger eintreffen.
- Ein Prozeß wird immer wieder seinen kritischen Bereich betreten.

Lassen sich auf der einen Seite solche Eigenschaften aus bestehenden Systemen ableiten, können auf der anderen Seite Sicherheits- und Lebendigkeitsanforderungen zur Spezifikation eingesetzt werden.

2 On-the-fly Garbage Collection in der Literatur

Bevor ich meine eigene Lösung des Problems erläutere, soll uns ein kurzer Streifzug durch die Literatur über die bisherigen Arbeiten zum Thema “nebenläufige Speicherbereinigung” informieren. Dieser Überblick ist weder vollständig, noch kann er die Leistungen, die den einzelnen Veröffentlichungen zugrundeliegen, ausreichend würdigen. Im Zweifels- oder Interessefall ist deshalb das Original zu konsultieren. Bei der Wiedergabe der einzelnen Arbeiten legen wir unser Augenmerk vor allem auf die verwendete Methodik und die algorithmische Idee.

2.1 G.L. Steele

In [STEE75] wird zum erstenmal über einen parallelen Garbage Collector berichtet. Der Autor stellt einen in der Sprache ECL (ähnlich Algol 68) implementierten Algorithmus vor, der es ermöglicht, in einem LISP-System mit mehreren Prozessoren und gemeinsamen Speicher, die Aufgabe der Listenverarbeitung und die der Speicherwiedergewinnung auf zwei Prozessoren zu verteilen. Darüberhinaus kann der Garbage Collector auf Wunsch auch für eine Kompaktifizierung des Heaps sorgen, d.h. die benutzten Speicherobjekte wandern an die untere Grenze der Halde und werden dort möglichst ohne Lücken gespeichert. Die Kompaktifizierung der Halde ist bei Benutzung von Speicherobjekten variabler Größe eine hilfreiche Maßnahme, die gewährleistet, daß der frei gewordene Speicherplatz wieder vollständig benutzt werden kann. Im Gegensatz zum später entwikkelten DLMSS-Algorithmus, macht Steele häufigen Gebrauch von Operationen auf Semaphore. Das bedeutet, daß das vorgestellte Programm — wie der Autor selbst erwähnt — nur auf speziell konstruierter Hardware effizient einsetzbar ist. Der Autor gibt in [STEE75] keine Hinweise auf die Korrektheit des vorgestellten Algorithmus.

2.1.1 Steele’s Algorithmus

Jedes Speicherobjekt erhält einen Markierungsbereich von 2 Bit, mit dessen Hilfe verschiedene Zustände signalisiert werden. Die Bedeutung der beiden Markierungen hängt von der aktuellen Arbeitsphase des Collectors ab und kann nicht direkt mit den Farbmarkierungen des DLMSS-Algorithmus verglichen werden. Der Collector führt zyklisch die folgenden Arbeiten aus:

1. **Mark phase**: Ausgehend von der Wurzel des Speichergeflechts werden alle erreichbaren Speicherzellen markiert. Zur Traversierung des Objektgraphen wird ein rekursives depth-first Verfahren benutzt, d.h. der Garbage Collector muß einen eigenen Stack anlegen und pflegen (entfällt beim DLMSS-Algorithmus). Auch bei diesem Algorithmus muß der Listenprozessor (entspricht dem Mutator) den Garbage Col-

lector in bestimmten Situationen unterstützen: wird eine neue Referenz erzeugt, dann muß der Mutator das Zielobjekt in Form einer Referenz auf den Stack des Garbage Collectors legen, um zu gewährleisten, daß das Objekt in der aktuellen Markierungsphase sicher als erreichbar gilt. Der gleiche Effekt wird in der Lösung von Dijkstra et al. erreicht, indem der Mutator die Markierung des betreffenden Objekts ändert.

2. **Relocate phase**: Die in Phase 1 als erreichbar erkannten Speicherzellen werden zu einem möglichst kompakten Block zusammengefaßt. Damit zur gleichen Zeit der Mutator arbeiten kann, wird im Falle einer Objektverschiebung an der alten Stelle des Speicherobjekts die neue Adresse der verschobenen Zelle hinterlegt. Der Listenprozessor erkennt diesen Zustand an einer besonderen Markierung der alten Stelle und führt in diesem Fall einen indirekten Objektzugriff durch.

3. **Update phase**: Die in Phase 2 entstandenen indirekten Verweise werden durch direkte Objektverweise ersetzt. Nach dieser Phase ist die Kompaktifizierung der Halde abgeschlossen.

4. **Reclaim phase**: Alle Objekte, die nicht in Phase 1 als erreichbar markiert wurden, können jetzt freigegeben werden. Darüberhinaus müßen alle Objektmarkierungen zurückgesetzt werden.

Die Abschnitte 2 und 3 implementieren die Kompaktifizierung des Heaps und sind optional.

2.1.2 Zusammenfassung

G.L. Steele legt seiner Arbeit einen sehr praxis-orientierten Ansatz zur Lösung paralleler Probleme zugrunde. Das führt dazu, daß er einerseits ein (anscheinend) sofort einsetzbares Programm präsentieren kann, andererseits aber die Frage der Korrektheit gänzlich mißachtet. Diese Verfahrensweise ist vor allem bei parallelen Algorithmen äußerst risikoreich und wird im Fall des parallelen Garbage Collectors nicht der Komplexität der möglichen Abläufe gerecht. Der Anspruch der praktischen Einsatzfähigkeit — den der Autor offensichtlich stellt — ist dagegen durch die Größe der verwendeten unteilbaren Operationen gefährdet, die eine effiziente Implementierung auf klassischen von-Neumann-Rechnern fragwürdig erscheinen lassen. Nichtsdestoweniger rechtfertigt der innovative Charakter der Arbeit eine besondere Würdigung von [STEE75] an dieser Stelle.

2.2 E.W. Dijkstra et al.

In [DIJK76] stellen fünf Autoren ein neues Verfahren für einen parallelen Garbage Collector vor und zwar den (später nach ihren Anfangsbuchstaben benannten) DLMSS-Algorithmus. Die Intention ihrer Arbeit war die Entwicklung einer Technik, die es erlaubt, den größten Teil des Arbeitsaufwands zur “Müllsammlung” einem zweiten Prozessor zu

übertragen, der seine Operationen parallel zu dem Prozessor ausführt, der für die eigentliche Programmabwicklung zuständig ist. Dabei soll eine gegenseitige Behinderung so weit wie möglich ausgeschlossen sein, d.h. der Synchronisationsaufwand und die Komplexität der verwendeten unteilbaren Operationen muß gering sein. Die Autoren schildern anschaulich die Schwierigkeiten, mit denen sie bei der Entwicklung eines "intuitiv korrekten" Algorithmus zu kämpfen hatten, insbesondere gab es zum Zeitpunkt der Veröffentlichung nach Meinung der Autoren noch keinen formalen Korrektheitsbeweis, der seinen Abdruck gerechtfertigt hätte. Nichtsdestoweniger veröffentlicht zum gleichen Zeitpunkt D. Gries einen Korrektheitsbeweis des DLMSS-Algoritmus [GRIE76]. Er benutzt dabei die kurz zuvor von S. Owicki in ihrer Dissertation veröffentlichte "axiomatische Beweistechnik für parallele Programme" [OWIC76]. Im folgenden beschreibe ich zuerst die Besonderheiten des DLMSS-Algorithmus, wie sie auch in [DIJK76] hervorgehoben werden; in Abschnitt 2.3 gehe ich dann kurz auf den Korrektheitsbeweis von D. Gries und die von ihm benutzte Beweistechnik ein.

2.2.1 Die Datenstruktur eines LISP-Systems

Das Problem der Garbage Collection stellte sich zur Zeit der Veröffentlichung von [DIJK76] vor allem in LISP-Systemen, da es kaum eine andere weit verbreitete Programmiersprache gab, in der die Speicherwiedergewinnung automatisch durchgeführt werden sollte. So beschränkten sich Dijkstra et al. darauf, ihren parallelen Garbage Collector in der Umgebung eines traditionellen "reinen" LISP-Systems zu sehen. Die Datenstruktur besteht somit aus einem gerichteten Graphen, in dem jeder Knoten genau zwei abgehende Kanten aufweist, nämlich eine Kante, die zum linken, und eine, die zum rechten Nachfolgeknoten führt. Alle Knoten, die durch einen Pfad von einem der besonders ausgezeichneten Wurzelknoten erreichbar sind (es gibt also mehr als nur eine Wurzel), gehören zur Datenstruktur des LISP-Systems, alle anderen Knoten sind Müll. Ausgehend von einem gegebenen Knoten kann der rechte oder linke Nachfolger durch eine atomare und primitive Operation erreicht werden; um die Vorgängerknoten zu erreichen, muß jedoch die ganze Datenstruktur durchsucht werden. Zur Homogenisierung des Modells tragen zwei Maßnahmen bei:

1. Die Kanten, die laut Programmvorschrift eigentlich nicht existieren (NIL-Zeiger), werden als Kanten zu einem speziellen "NIL-Knoten" aufgefasst, der als Wurzel gilt. Die beiden Kanten des NIL-Knotens zeigen selbst auf den NIL-Knoten.

2. Die freien Knoten werden in einer linearen Liste verkettet und an einen weiteren speziellen Wurzelknoten gehängt. Sie gehören damit zur Datenstruktur des LISP-Prozesses.

Als Folge dieser Maßnahmen gibt es nur noch eine einzige Sorte von Operationen, mit der die Datenstruktur manipuliert wird: eine der abgehenden Kanten eines erreichbaren Knoten wird auf einen ebenso erreichbaren Knoten gerichtet. Andere Operationen, wie z.B. das Löschen oder Hinzufügen von Kanten, sind nicht nötig.

Die Aufgabe des Garbage Collectors besteht nun darin, alle Knoten, die nicht mehr in der Datenstruktur enthalten sind, zu erkennen und an die Liste der freien Knoten anzuhängen. Die dazu notwendigen Abläufe sollten nach der Vorstellung der Autoren von [DIJK76] auf einen zweiten (physikalischen) Prozessor übertragen werden können, den sie "Collector" nannten. Der eigentliche LISP-Prozessor, dessen Abläufe durch das LISP Programm festgelegt werden, erhielt die Bezeichnung "Mutator". Beide Prozessoren haben einen gemeinsamen Arbeitsspeicher, in dem die Knoten und Kanten des gerichteten Graphen aufbewahrt werden.

2.2.2 Erste Überlegungen

Bereits nach den ersten Untersuchungen wurde klar, daß es nicht möglich sein würde, einen parallelen Collector zu entwerfen, bei dem der Collector auf eine Unterstützung durch den Mutator verzichten kann. Dieser Sachverhalt wurde in 1.4 als das Konsistenzproblem vorgestellt. [DIJK76] folgerten daraus, daß der Mutator die Knoten markieren muß, auf die er neue Kanten richtet.

Als Knotenmarkierung werden Farben eingeführt. Ausgehend von einem Zustand, in dem alle Knoten weiß sind, sollten am Ende der Markierungsphase alle erreichbaren Knoten schwarz sein, wodurch die noch weißen Knoten als Müll erkannt worden wären. Im Hinblick auf den Korrektheitsbeweis (der zum damaligen Zeitpunkt noch ohne die Begriffe Sicherheits- und Lebendigkeitseigenschaften auskommen mußte) wurden zwei Anforderungen formuliert: durch eine Monotonie-Klausel sollte sich die Terminierung der Markierungsphase beweisen lassen, während eine Invariante für ein korrektes Ergebnis sorgt. Im ersten Lösungsansatz wurde folgende Monotonieannahme gewählt:

> Während der Markierungsphase nimmt die Helligkeit jeder Knotenmarkierung monoton ab.

Für die Invariante mußte der Anfangs- und Endzustand des Markierungsvorgangs verallgemeinert werden, um eine gemeinsame Eigenschaft zu formulieren:

P1. Während des Markierungsvorgangs gibt es keine Kante, die von einem schwarzen Knoten auf einen weißen Knoten zeigt.

Aus P1 ergibt sich die Forderung, daß der Mutator nicht nur einfach neue Kanten erzeugen kann, sondern zusätzliche Operationen ausführen muß, die die Markierungen der beteiligten Knoten betreffen. Die Monotonie-Klausel besagt, daß der schwarze Knoten, von dem aus die neue Kante gelegt werden soll, seine Farbe beibehalten muß, woraus folgt, daß ein weißer Zielknoten seine Farbe ändern muß. Der Mutator kann diesen Zielknoten aber nicht einfach schwarz färben, da wieder P1 verletzt werden könnte, diesmal im Hinblick auf die vom Zielknoten abgehenden Kanten und deren Zielknoten. Aus diesem Grund wird die Farbe grau als zusätzliche Knotenmarkierung eingeführt. Die vom Mutator geforderte Unterstützung wird so formuliert:

A1. Erzeugt der Mutator eine neue Kante, so muß er den Zielknoten aktivieren, d.h. ein weißer Knoten wird grau, ein grauer Knoten bleibt grau und ein schwarzer Knoten bleibt scharz.

Die Wahl der Invarianten im ersten Ansatz führte zu einer Lösung, in der die unteilbaren Operationen komplexer waren, als ursprünglich erwünscht; sie wurde daher wieder verworfen. Insbesondere hätte A1 eine atomare Operation sein müssen, um P1 nicht zu verletzen. Die Grundidee des informellen Korrektheitsbeweises lag in einer Folgerung aus P1, die besagt, daß die Existenz eines noch weißen erreichbaren Knotens die Existenz eines grauen Knotens voraussetzt. Sobald alle grauen Knoten verschwunden sind, gibt es auch keine erreichbaren Knoten mehr, die noch weiß sind.

2.2.3 Über die Komplexität atomarer Operationen

Die Prozesse von Mutator und Collector werden als eine Folge von unteilbaren, d.h. atomaren Operationen aufgefaßt. E.W. Dijkstra et al. benutzen in diesem Zusammenhang den Begriff "*grain of interleaving*" (wörtlich etwa: Granularität des Abwechselns) und beziehen sich dabei auf die Auswirkung der atomaren Operationen der beteiligten Prozesse auf den Grad der Parallelität des Systems. In [DIJK76] werden verschiedene Lösungen für die parallele Wiedergewinnung von Speicher angegeben, die sich in der Komplexität der verwendeten unteilbaren Operationen unterscheiden. Die "coarse grained solution" benutzt Operationen, deren Größe die Komplexität von herkömmlichen Maschinenbefehlen übersteigt. Eine Implementierung dieses Algorithmus auf klassischen von-Neumann-Rechnern erfordert Semaphore, um die Unteilbarkeit der Operationen künstlich zu gewährleisten. Im entgültigen DLMSS-Algorithmus, der das Attribut "fine grain of interleaving" trägt, erreichen die verwendeten atomaren Operationen gerade die Größe von Maschinenbefehlen eines von-Neumann-Rechners. Die Unteilbarkeit der Operationen wird folglich bereits durch die Hardware gewährleistet, was eine allgemeine praktische Einsatzfähigkeit garantiert.

Die Größe der unteilbaren Operationen hat aber nicht nur Einfluß auf die Effizienz der Implementierung, sondern wirkt sich auch auf die Komplexität des Korrektheitsbeweises aus, da kleinere atomare Operationen die Abhängigkeiten der beteiligten Prozesse vergrößert. In einer Beweistechnik, die zuerst die Korrektheit der sequentiellen Prozesse zeigt und dann erst, daß sich die Prozesse nicht über das nötige Maß hinausgehend gegenseitig beeinflussen, wird die gewählte Größe der atomaren Operationen einen entscheidenden Einfluß auf die Komplexität des Korrektheitsbeweises haben (siehe S. 19: partielle Korrektheit des DLMSS-Algorithmus, bewiesen von D. Gries). Je größer nämlich die geforderten unteilbaren Prozeßabschnitte sind, desto kleiner und überschaubarer sind die möglichen unbeabsichtigten Interferenzen und umgekehrt.

2.2.4 Lösungen mit verschiedener Granularität der atomaren Operationen

Um die gegenseitige Behinderung von Mutator und Collector zu verringern, wurde P1 durch zwei weniger harte Invarianten ersetzt, aus denen sich die gleichen Folgerungen ableiten lassen:

P2. Während der Markierungsphase gibt es für jeden erreichbaren Knoten, der noch weiß ist, einen Pfad zu ihm, der bei einem grauen Knoten startet und nur über Knoten führt, die selbst weiß sind.

P3. Während der Markierungsphase darf höchstens die letzte vom Mutator erzeugte Kante von einem schwarzen zu einem weißen Knoten führen.

Die beiden Annahmen führten in [DIJK76] zuerst auf einen Algorithmus, der A1 durch A2 ersetzt:

A2. Erzeugt der Mutator ein Kante, so muß er zuvor den Zielknoten aus der vorhergehenden Operation A2 aktivieren.

Auch A2 muß eine atomare Operation sein. Der so begründete Algorithmus wurde in einer mehr oder weniger formalen Programmiersprache notiert und hatte eine komplexere atomare Operation: Der Collector muß in einem unteilbaren Schritt die beiden Nachfolgeknoten eines Knoten i aktivieren und anschließend i schwarz färben. Unter dieser Annahme ergibt sich eine relativ einfache Form für die informelle Begründung der Korrektheit. In einem zweiten Schritt zeigen die Autoren dann, daß auch diese Anforderung abgeschwächt werden kann, ohne die Invarianten P2 und P3 zu verletzen. In einem letzten Schritt schließlich wird auch A2 in zwei einzelne Mutatoroperationen aufgebrochen:

1. Ausgehend von einem erreichbaren Knoten wird eine Kante auf einen ebenfalls erreichbaren Knoten gerichtet.
2. Aktiviere den Zielknoten aus dem letzten Schritt.

Damit ergibt sich ein Algorithmus, der auf herkömmlichen Rechnern ohne Semaphor-Operationen implementiert werden kann.

2.2.5 Der DLMSS-Algorithmus

Der Collector führt zyklisch die folgenden Befehle aus [LAMP76], wobei V die Menge aller Knoten und $\emptyset$ die leere Menge darstellt (zu Beginn sind alle Knoten weiß):

```
(* Markierungsphase *)
shade all roots;
S := ∅;
while S ≠ V
        do      choose n ∈ V;
                S := S ∪ {n};
```

```
            if n is grey     then   shade each successor of n;
                                    color n black;
                                    S := ∅
            fi
      od;
(* Bereinigungsphase *)
for all n ∈ V
      do    if n is white    then   put n on the free list
                             else   color n white
            fi
      od
```

Die Markierungsphase wird nach endlicher Zeit terminieren, sofern nur die Auswahl von n zu Beginn der while-Schleife fair ist, d.h. jeder Knoten aus V wird irgendwann ausgewählt. Die Operation *shade* aktiviert einen Knoten, indem sie den Knoten grau färbt, falls er vorher weiß war. Knoten, die bereits grau oder schwarz sind, bleiben unverändert. Der Mutator führt zyklisch die folgenden Befehle aus, wobei W die Menge aller erreichbaren Knoten darstellt ($W \subseteq V$):

```
choose j, k ∈ W;
make k successor of j;
shade k
```

2.2.6 Zusammenfassung

In [DIJK76] wird der erste praktikable Algorithmus für einen nebenläufigen Garbage Collector präsentiert. Über die Korrektheit ihres DLMSS-Algorithmus argumentieren die Autoren nur informell. Um die Korrektheit der letzten Version ihres Collectors anhand des Programmtextes zu begründen, benötigen sie bereits ein gehöriges Maß an subtilen Argumenten. Offensichtlich spielt dabei die Größe der atomaren Operationen eine entscheidende Rolle, da kleinere atomare Operationen die möglichen Interferenzen zwischen dem Collector- und dem Mutatorprozeß vergrößern.

2.3 D. Gries

In [GRIE76] beschäftigt sich der Autor mit dem Beweis von Sicherheitseigenschaften des in [DIJK76] vorgestellten parallelen Garbage Collectors. Es folgt zuerst eine kurze Einführung in die von D. Gries benutzte Beweistechnik, anschließend werde ich die Idee aufzeigen, die zum Beweis der partiellen Korrektheit des DLMSS-Algorithmus führt.

2.3.1 Beweistechnik

Die in [GRIE76] angewandte Beweistechnik stammt von S. Owicki und basiert auf der in [HOAR69] vorgestellten axiomatischen Beweistechnik für sequentielle Programme. Diese von C.A.R. Hoare entwickelte Technik ist dazu geeignet, die partielle Korrektheit von sequentiellen Algorithmen anhand des Programmtextes zu zeigen. Eine Klausel

$$\{P\}\ \ S\ \ \{Q\}$$

in der S eine Folge von Anweisungen und P, Q logische Ausdrücke über Variablen sind, bedeutet, daß nach der Terminierung von S die Nachbedingung Q gilt, sofern vor der Ausführung von S die Vorbedingung P galt. Ob S tatsächlich terminiert, bleibt offen. Um die Vor- und Nachbedingungen eines Programmstücks aus seinen Einzelanweisungen herleiten zu können, gibt Hoare eine Reihe von Regeln an:

null $\quad \{P\}\ \ \textbf{skip}\ \ \{P\}$

assignment $\quad \{P_g^x\}\ \ x := g\ \ \{P\}$

alternation
$$\frac{\{P \wedge B\}\ \ S_1\ \ \{Q\},\quad \{P \wedge \neg B\}\ \ S_2\ \ \{Q\}}{\{P\}\ \ \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2\ \ \{Q\}}$$

iteration
$$\frac{\{P \wedge B\}\ \ S\ \ \{P\}}{\{P\}\ \ \textbf{while } B \textbf{ do } S\ \ \{P \wedge \neg B\}}$$

composition
$$\frac{\{P_0\}\ \ S_1\ \ \{P_1\},\quad \{P_1\}\ \ S_2\ \ \{P_2\}, ...,\ \{P_{n-1}\}\ \ S_n\ \ \{P_n\}}{\{P_0\}\quad \textbf{begin } S_1; \ldots ; S_n \textbf{ end}\ \ \{P_n\}}$$

consequence
$$\frac{\{P_1\}\ \ S\ \ \{Q_1\},\quad P \Rightarrow P_1,\quad Q_1 \Rightarrow Q}{\{P\}\ \ S\ \ \{Q\}}$$

Der Term P_g^x in der assignment-Regel repräsentiert den Term P, in dem alle freien Vorkommen von x durch g ersetzt wurden. Einige der Regeln sind Interferenzregeln der Form

$$\frac{a_1, a_2, \ldots, a_n}{b}$$

Sie bedeuten, daß sich aus der Gültigkeit von $a_1, \ldots, a_n$ die Gültigkeit von b folgern läßt. Eine Klausel $\{P\}\ S\ \{Q\}$ impliziert, daß es einen Beweis für diese Sicherheitseigenschaft gibt, unter Verwendung der oben aufgeführten Regeln. Zu Veranschaulichung wendet Hoare seine Technik auf ein Programm zur Division an:

```
r := x;
q := 0;
while y ≤ r do begin
        r := r - y;
        q := q + 1
```

end

Das Programm findet den Quotienten q und den Rest r, die bei der Division von x/y durch wiederholte Subtraktion entstehen. Um zu zeigen, daß r und q bei der Terminierung der Schleife die gewünschten Werte enthalten, muß gezeigt werden, daß die Gleichung

$$x = y \cdot q + r$$

eine Invariante der Schleife ist, d.h. die Gleichung gilt vor dem Eintritt in die Schleife und wird durch den Rumpf nicht zerstört. Mit der Regel *iteration* (s.o.) kann dann gefolgert werden, daß nach der Terminierung der Schleife die Gleichung

$$x = y \cdot q + r \ \wedge \ y > r \geq 0$$

gilt. Die zugehörige Beweisskizze sieht folgendermaßen aus:

$\{x \geq 0 \ \wedge \ y > 0\}$
$r := x;$
$\{x = r \ \wedge \ x \geq 0 \ \wedge \ y > 0\}$
$q := 0;$
$\{q = 0 \ \wedge \ x = r \ \wedge \ x \geq 0 \ \wedge \ y > 0\}$
$\{x = 0 + r \ \wedge \ r \geq 0 \ \wedge \ q = 0\}$
$\{x = y \cdot q + r \ \wedge \ r \geq 0\}$
while $y \leq r$ **do begin**
$\{x = y \cdot q + r \ \wedge \ y \leq r \ \wedge \ r \geq 0\}$
$r := r - y;$
$\{x = y \cdot (q + 1) + r \ \wedge \ r \geq 0\}$
$q := q + 1$
$\{x = y \cdot q + r \ \wedge \ r \geq 0\}$
end
$\{x = y \cdot q + r \ \wedge \ y > r \geq 0\}$

Dabei bedeuten zwei direkt hintereinander stehende Ausdrücke $\{P_1\}$ $\{P_2\}$, daß P_2 aus P_1 folgt, d.h. $P_1 \Rightarrow P_2$.

Um auch die partielle Korrektheit von parallelen Programmen zeigen zu können, erweiterte S. Owicki die vorgestellte Technik um Regeln zur Behandlung von Parallel-Operatoren [OWIC76]:

await
$$\frac{\{P \wedge B\} \ S \ \{Q\}}{\{P\} \ \textbf{await } B \textbf{ then } S \ \{Q\}}$$

parallelism
$$\frac{\{P_1\} \ S_1 \ \{Q_1\}, \ \{P_2\} \ S_2 \ \{Q_2\} \ \ \textit{interferenzfrei}}{\{P_1 \wedge P_2\} \ \textbf{cobegin } S_1 \parallel S_2 \textbf{ coend} \ \{Q_1 \wedge Q_2\}}$$

Dabei liefert *await* eine Semantik für kritische Bereiche mit unteilbaren Operationen, d.h. ein Prozeß wartet, bis B gültig ist und führt dann S als unteilbare Operation aus. Die Interferenzfreiheit zweier Beweise $\{P_1\}$ S_1 $\{Q_1\}$ und $\{P_2\}$ S_2 $\{Q_2\}$ ist gegeben, wenn

keine der Zuweisungen und *awaits* von S_1 die im Beweis von S_2 benutzten Vorbedingungen und die Nachbedingung Q_2 verletzt und umgekehrt. Durch die Interferenzfreiheit wird sichergestellt, daß die Einzelbeweise auch dann gültig sind, wenn die beiden Anweisungsfolgen parallel ausgeführt werden. S. Owicki führt darüberhinaus die Notation von Hilfsvariablen ein, um logische Ausdrücke mit "Gedächtnis" formulieren zu können. Diese Variablen dürfen keinen Einfluß auf den Ablauf des Programms haben.

2.3.2 Partielle Korrektheit des DLMSS-Algorithmus

Der Beweis von D. Gries kann hier nicht vollständig wiedergegeben werden; der interessierte Leser wird auf [GRIE76] verwiesen. Aufgrund der benutzten Beweistechnik gliedert sich der Beweis der partiellen Korrektheit des DLMSS-Algorithmus in zwei Abschnitte. Im ersten Teil wird die Korrektheit des Collectors, des Mutators und des Hauptprogramms (= Parallelkombination von Mutator und Collector) gezeigt. Der zweite Teil beschäftigt sich damit, die Interferenzfreiheit der einzelnen Beweise zu zeigen.

In der Beweisidee von D. Gries zur Herleitung der partiellen Korrektheit sind während der Markierungsphase u.a. die folgenden beiden Annahmen invariant:

P2. Alle Wurzelknoten sind mindestens grau, und für jeden weißen erreichbaren Knoten gibt es einen sog. Propagierungspfad, der bei ihm endet und bei einem grauen Knoten startet. Der Propagierungspfad führt nur über weiße Knoten.

P3. Keine Kante kann von einem schwarzen Knoten auf einen weißen Knoten gerichtet sein, es sei denn, daß der Mutator sie gerade erzeugt hat und als nächsten Schritt den weißen Zielknoten grau färbt.

P2 ist die entscheidende Eigenschaft, die garantiert, daß im Falle der Terminierung der Markierungsphase keine grauen Knoten mehr existieren und alle erreichbaren Knoten schwarz sind. Allerdings erfordert der Beweis von P2 den Beweis der stärkeren Annahme P2 $\wedge$ P3. P3 ermöglicht dem Mutator, ohne unteilbare Operationen auszukommen, die größer als normale Maschinenbefehle sind. In einer früheren Version der Beweisskizze aus [DIJK76] wurde P3' gefordert:

P3'. Keine Kante kann von einem schwarzen Knoten auf einen weißen Knoten zeigen.

P3' vereinfacht zwar den Beweis, erfordert aber, daß die Mutatoroperationen

make k successor of j;
shade k

unteilbar sein müßen (siehe DLMSS-Algorithmus im vorhergehenden Abschnitt).

2.3.3 Zusammenfassung

Die von D. Gries benutzte Beweistechnik betrachtet ein paralleles Programm als eine Menge von parallel auszuführenden sequentiellen Prozessen. Daraus resultiert die Metho-

dik, zuerst Eigenschaften der einzelnen Prozesse nachzuweisen — ohne die restlichen Prozesse zu betrachten — und dann die Interferenzfreiheit aller Beweise untereinander herzuleiten. Dabei ergeben sich zwei Schwierigkeiten:

1. Um später bei der Überprüfung der Interferenzfreiheit erfolgreich zu sein, müßen die Annahmen, die in den Einzelbeweisen verwendet werden, Rücksicht auf die anderen Prozesse nehmen. Dadurch wird der Beweis für den einzelnen Prozeß komplizierter, als es die Methodik vermuten läßt.

2. Im Normalfall erfordert der Nachweis der Interferenzfreiheit Aufwendungen, die proportional zum Produkt der Anzahl der unteilbaren Anweisungen je Prozeß sind. Bei einem System aus zwei Prozessen, die aus n bzw. m atomaren Operationen bestehen, ist somit der erforderliche Arbeitsaufwand in der Größenordnung von $n \cdot m$.

In Anbetracht dieses Arbeitsaufwands kann die in [GRIE76] benutzte Beweistechnik nicht überzeugen. Sie führt im Fall des DLMSS-Algorithmus zu einem schwerfälligen und unhandlichen Beweis. Andererseits gelingt es D. Gries anhand der Beweisskizzen, die subtilen Eigenschaften eines parallelen Garbage Collectors aufzuzeigen und damit dem Leser wertvolle Einblicke zu ermöglichen.

2.4 H.T. Kung und S.W. Song

In [KUNG77a] und [KUNG77b] versuchen die Autoren zwei grundsätzlich verschiedene Aspekte eines parallelen Algorithmus zu betrachten. Zum einen liefern sie einen informellen Beweis der Korrektheit ihrer Lösung, zum anderen analysieren sie die Effizienz der Implementierung. Der von H.T. Kung und S.W. Song vorgestellte parallele Garbage Collector basiert im wesentlichen auf dem DLMSS-Algorithmus, besitzt jedoch zwei Erweiterungen:

1. Die grauen Knoten werden — in Form einer Referenz — in einer Liste außerhalb des Speichergeflechts aufbewahrt. Diese Maßnahme sorgt für eine Beschleunigung der Markierungsphase, da der Collector nicht mehr alle Knoten auf ihre Markierung hin prüfen muß, sondern alle grauen Knoten an einem zentralen Ort abholen kann. Die Liste wird asynchron von beiden Seiten her bearbeitet. Auf der einen Seite fügt der Mutator die Knoten an, die er wegen seiner eigenen Aktivitäten grau markieren mußte. Auf der anderen Seite holt der Collector den nächsten grauen Knoten ab, dessen Söhne er aktivieren muß (d.h. weiße Knoten werden grau), und er fügt die Knoten an, die er selbst grau markiert. Sobald der Collector beim Abholen des nächsten grauen Knoten eine leere Liste vorfindet, endet die Markierungsphase. Dieses Verfahren ist mit dem rekursiven depth-first Traversierungsalgorithmus vergleichbar, der von G.L. Steele in seinem Garbage Collector benutzt wird (siehe S. 13).

2. Zusätzlich zu den Farben weiß, grau und schwarz wird eine vierte Farbe eingeführt (*off-white*), mit der alle freien Knoten markiert sind. Damit wird verhindert, daß der

Collector einen freien Knoten grau markiert und an die in Punkt 1 erwähnte Liste anhängt. Die Zeit, die für die Markierungsphase aufgewendet werden muß, hängt damit nur noch von der Zahl der tatsächlich benutzten Knoten ab und nicht mehr wie bisher von der Gesamtanzahl der Knoten.

Beide Maßnahmen sorgen für eine Steigerung der Geschwindigkeit des Garbage Collectors. Die erste Maßnahme führt aber auch dazu, daß der Mutator einen größeren Aufwand betreiben muß, da er ja nicht nur Knoten grau färbt (wie im DLMSS-Algorithmus), sondern sie auch noch an die Liste der grauen Knoten anhängen muß. Die Autoren charakterisieren die Korrektheit ihres Systems durch 3 Bedingungen:

C1. Nur Knoten, die Müll sind, werden an die Liste der freien Knoten angehängt.

C2. Der Garbage Collector verändert nicht die Zeiger eines aktiven Knotens (d.h. ein Knoten, der von einer Wurzel aus erreichbar ist).

C3. Ein Knoten, der Müll geworden ist, wird nach endlicher Zeit als solcher erkannt und an die Liste der freien Knoten angehängt. Diese Zeit kann im voraus abgeschätzt werden.

Während C1 und C2 für die sichere Zusammenarbeit von Mutator und Collector sorgen, fordert C3 die Lebendigkeit des Collectors.

2.4.1 Beweistechnik

Die Korrektheit des Systems wird zuerst anhand eines Hilfsalgorithmus bewiesen, der dann durch schrittweises Verfeinern in den eigentlichen Algorithmus transformiert wird. Das Verfahren gleicht damit der Systematik, die in der hierarchischen Spezifikation meiner Arbeit Verwendung findet, wenngleich der Beweis in [KUNG77a] nur informell geführt wird. Ähnlich wie in [GRIE76] werden auch hier Hilfsvariablen und Hilfsoperationen eingeführt, um die Argumentation zu erleichtern.

2.4.2 Zusammenfassung

H.T. Kung und S.W. Song beschäftigen sich in [KUNG77a] vor allem mit den praktischen Aspekten eines parallelen Garbage Collectors. Ihre Verbesserungsvorschläge für den DLMSS-Algorithmus sind (zumindest aus heutiger Sicht) naheliegend.

2.5 S. Ramesh und S.L. Mehndiratta

Die bisherigen Versuche, den DLMSS-Algorithmus zu beweisen, beschränkten sich entweder auf informelle Argumentationen oder auf die Sicherheitseigenschaften des Collectors. Hauptaugenmerk der Untersuchungen war die Anforderung, daß kein Knoten in die Liste der freien Knoten aufgenommen wird, wenn er noch in der Datenstruktur des Mutators enthalten ist. Die Zurückhaltung im Bereich der Lebendigkeit findet seine Ursache in

den bis dato mangelhaften Werkzeugen zum Beweis von Lebendigkeitseigenschaften paralleler Programme. In [OWIC82] veröffentlichen Leslie Lamport und Susan Owicki eine praktikable Methode zum Beweis von Lebendigkeit, die auf temporaler Logik basiert. In [RAME83] dient diese Methode als Grundlage für einen formalen Beweis der Lebendigkeit des DLMSS-Algorithmus.

2.5.1 Beweistechnik

Temporale Logik erweitert die herkömmliche Logik um Begriffe zur Modellierung von Annahmen über die Zukunft. Die von Lamport und Owicki herangezogene Logik benutzt zwei temporale Operatoren:

$\Box P$ "P gilt ab jetzt für immer (*forever*)"

$\Diamond P$ "P gilt jetzt oder irgendwann in der Zukunft (*eventually*)"

In [OWIC82] wurde gezeigt, daß jede Lebendigkeitseigenschaft durch einen Ausdruck der Form $\Box(P \Rightarrow \Diamond Q)$ oder kurz $P \mapsto Q$ ausgedrückt werden kann. Der dadurch entstehende neue Operator ist übrigens dem in dieser Arbeit benutzten *leads-to* sehr ähnlich. Zur Abstützung am Programmtext werden Prädikate eingeführt, die Aussagen über den momentanen Status des Kontrollflusses machen:

at A gilt, wenn der Programmablauf kurz vor der Instruktion mit der Beschriftung A angekommen ist.

in A gilt, wenn sich der Programmablauf innerhalb des Instruktionsblocks mit der Beschriftung A befindet.

after A gilt, wenn sich der Programmablauf kurz nach der Instruktion mit Namen A befindet.

Für eine Beschreibung des Ausführungsmodells sei direkt auf [OWIC82] verwiesen. Die Formulierung von Sicherheitseigenschaften ist identisch mit dem erweiterten Hoare-Kalkül für parallele Programme, welches in Zusammenhang mit D. Gries in Abschnitt 2.3 beschrieben wurde.

2.5.2 Beweisidee

Zum Beweis der Lebendigkeit des DLMSS-Algorithmus benutzen Ramesh und Mehndiratta 4 Hilfsprädikate:

reach(k) Knoten k ist von der Wurzel aus erreichbar, oder er befindet sich in der Liste der freien Knoten.

grayreachable(k) Knoten k ist von einem grauen Knoten aus durch einen Pfad erreichbar, der nur über weiße Knoten führt.

whitereachable(k) Knoten k ist nur von weißen Knoten aus erreichbar.

blackreachable(k) Knoten k ist von einem schwarzen Knoten aus durch einen Pfad erreichbar, der nur über schwarze Knoten führt (insbesondere ist dann auch k schwarz).

Die zentrale Lebendigkeitsanforderung an den Collector formulieren die beiden Autoren folgendermaßen:

$$\neg \mathit{reach}(k) \;\mapsto\; \mathit{reach}(k)$$

Die rechte Seite wird erreicht, wenn der Collector den Knoten k als Müll erkannt hat und ihn an die Liste der freien Knoten anhängt. Der Beweis dieser Anforderung stützt sich auf die folgenden Annahmen:

$$\textbf{at}\ \mathit{Blacken} \;\mapsto\; \textbf{at}\ \mathit{Collect}$$

was soviel heißt wie, "die Markierungsphase des Collectors terminiert irgendwann", wobei *Blacken* der Name des Instruktionsblocks ist, der die Markierung übernimmt und *Collect* analog die Instruktionen zur Bereinigung bezeichnet. Zusammen mit dem Hoare-Tripel

$$\{\mathit{grayreachable}(k) \wedge \neg \mathit{reach}(k)\} \quad \mathit{Blacken} \quad \{\mathit{blackreachable}(k) \wedge \neg \mathit{reach}(k)\}$$

ergibt sich

$$\textbf{at}\ \mathit{Blacken} \wedge \neg \mathit{reach}(k) \wedge \mathit{grayreachable}(k) \;\mapsto$$
$$\textbf{at}\ \mathit{Collect} \wedge \neg \mathit{reach}(k) \wedge \mathit{blackreachable}(k)\ .$$

Weitere Lemmas sind

$$\textbf{at}\ \mathit{Collect} \wedge k\ \mathit{is\ black} \wedge \neg \mathit{reach}(k) \;\mapsto$$
$$\textbf{at}\ \mathit{Blacken} \wedge \mathit{whitereachable}(k) \wedge \neg \mathit{reach}(k)$$

und

$$\textbf{at}\ \mathit{Blacken} \wedge \mathit{whitereachable}(k) \wedge \neg \mathit{reach}(k) \;\mapsto$$
$$\textbf{at}\ \mathit{Collect} \wedge k\ \mathit{is\ white} \wedge \neg \mathit{reach}(k)\ .$$

Aus den drei letzten Eigenschaften kann gefolgert werden, daß jeder Müllknoten nach spätestens einer kompletten Markierungsphase weiß ist, und dann die Bereinigungsphase ansteht, die die weißen Knoten an die Liste der freien Knoten anhängt. Daraus folgt die Behauptung, daß jeder Müllknoten irgendwann an die Liste der freien Knoten angehängt wird.

2.5.3 Zusammenfassung

In [RAME83] wird ein relativ übersichtlicher Beweis der Lebendigkeit des DLMSS-Algorithmus vorgeführt, der in einigen Punkten auf bereits bewiesene Annahmen aus

[GRIE76] zurückgreift. Eine dieser Annahmen betrifft die Terminierung der Markierungsphase, die von D. Gries allerdings mehr oder weniger informell bewiesen wurde.

2.6 M. Ben-Ari

In [BENA84] wurde ein Algorithmus vorgestellt, der einige Unterschiede im Vergleich mit dem DLMSS-Algorithmus aufweist. Ziel der Arbeit war es, einen Collector zu entwickeln, dessen Korrektheit möglichst einfach zu beweisen ist.

2.6.1 Ben-Ari's Algorithmus

Anstelle von drei Farben im DLMSS-Algorithmus werden nur noch zwei Farben als Knotenmarkierung benötigt: weiß und schwarz. Zu Beginn der Markierungsphase sind alle Knoten weiß. Die Markierungsphase kann selbst wieder in drei Unterphasen zerlegt werden:

1. Am Anfang der Markierungsphase werden der (oder die) Wurzelknoten schwarz gefärbt.
2. In der Propagierungsphase wird von jedem Knoten genau einmal die Farbe getestet: Ist der Knoten schwarz, dann werden seine Söhne ebenfalls schwarz gefärbt.
3. In der Zählphase wird die Anzahl der schwarzen Knoten ermittelt. Hat sie im Vergleich zur vorhergehenden Zählphase zugenommen, dann wird bei Punkt 2 fortgefahren, andernfalls ist die Markierungsphase beendet.

Wie bereits beim DLMSS-Algorithmus, wird auch hier gefordert, daß der Mutator mit dem Collector kooperiert. Bei Ben-Ari's Algorithmus hat der Mutator selbständig Knoten schwarz zu färben, wenn er neue Kanten auf sie richtet.

2.6.2 Korrektheit

Ben-Ari zeigt, daß am Ende der Markierungsphase alle weißen Knoten Müll sind. Als Beweistechnik kommt die bereits bei D. Gries in Abschnitt 2.3 erläuterte Methode zur Anwendung [LAMP77, OWIC76]. Einige Stellen im Programmtext werden mit Invarianten assoziiert, deren Gültigkeit — so wird gezeigt — durch keine andere Instruktion verletzt wird. Der Nachweis der Lebendigkeit besteht im wesentlichen darin, zu zeigen, daß die Markierungsphase terminiert. Da die Propagierungsphase nur dann wiederholt wird, wenn die Anzahl der schwarzen Knoten zugenommen hat und die Anzahl der Knoten nach oben begrenzt ist, kann angenommen werden, daß die Markierungsphase irgendwann terminiert. Der Nachweis der Lebendigkeit wird mit Hilfe temporal-logischer Formeln ausgeführt, wie sie z.B. in [PNUE81] beschrieben werden.

2.6.3 Zusammenfassung

Tatsächlich gelingt es M. Ben-Ari, einen knappen Beweis der Sicherheit und der Lebendigkeit für seinen Algorithmus anzugeben. Allerdings kann seine Beweistechnik nicht als formal bezeichnet werden (zumindest nicht im Sinne einer maschinellen Überpüfbarkeit). Dieses Attribut kann jedoch keiner der bisher beschriebenen Beweise von parallelen Garbage Collectoren für sich in Anspruch nehmen. Die Reduzierung des Speicheraufwands von 2 Bit auf 1 Bit je Knoten im Vergleich zum DLMSS-Algorithmus erkauft man sich durch eine geringere Effizienz, was man jedoch angesichts des übersichtlichen Beweises verschmerzen kann. Ein weiterer, noch übersichtlicherer Beweis von Ben-Ari's Algorithmus findet sich in [SNEP87].

2.7 K.M. Chandy und J. Misra

Ein besonders gelungener Beweis eines parallelen Garbage Collectors findet sich in [CHAN88]. In diesem Buch veröffentlichen die beiden Autoren eine neue Programmnotation mit der dazu passenden Beweislogik. Der größte Teil des Buches widmet sich jedoch einer stattlichen Sammlung von Fallbeispielen, die keinen Zweifel daran lassen, daß die von K.M. Chandy und J. Misra entworfene Programmlogik geeignet ist, um parallele Systeme zu implementieren und zu spezifizieren. Eines dieser Fallbeispiele beschäftigt sich mit dem Thema "On-the-fly Garbage Collection".

Bevor ich diese Arbeit vorstelle, gebe ich zuerst eine kurze Einführung in das Programmiermodell, das Chandy und Misra benutzen. Die dazugehörige Programmlogik werde ich nicht an dieser Stelle beschreiben, da sie weitgehend mit der Beweislogik übereinstimmt, die ich in meiner Arbeit verwende. Für Details sei auf Abschnitt 4.1 verwiesen.

2.7.1 UNITY-Programme

Ein UNITY-Programm besteht im wesentlichen aus drei Sektionen: der Variablendeklaration, der Angabe ihrer Anfangswerte und einer Menge von bedingten (Mehrfach-) Zuweisungen. Eine Programmausführung beginnt bei irgendeinem Zustand, der die Anfangswerte der Variablen respektiert, und läuft endlos weiter. In jedem Programmschritt wird irgendeine der bedingten Zuweisungen ausgewählt. Sie wird ausgeführt, falls ihre Bedingung wahr ist. Die Auswahl der Zuweisungsinstruktion wird von einer Fairness-Regel kontrolliert: Jede Instruktion muß unendlich oft selektiert werden.

Für Programme, die nach endlicher Zeit terminieren, wird der Fixpunktbegriff eingeführt. Ein Zustand einer Programmausführung, in der alle Zuweisungsinstruktionen den Zustand unverändert lassen, wird als Fixpunkt bezeichnet. Eine Programmausführung gilt als terminiert, sobald ein Fixpunkt erreicht wird.

2.7.2 Ein kurzes Beispiel: Sortieren

Ein Feld von natürlichen Zahlen $A[0..N]$, $N \geq 0$, soll in aufsteigender Reihenfolge sortiert werden:

Program *sort*
assign
$$\langle \,[\!]\, i \;:\; 0 \leq i < N \;::\; A[i], A[i+1] \;:=\; A[i+1], A[i] \quad \text{if} \quad A[i] \;>\; A[i{+}1] \,\rangle$$
end {*sort*}

Sobald ein Fixpunkt erreicht ist, liegen die Elemente von Feld A in sortierter Reihenfolge vor. Als schematische Abkürzung wird in diesem Programm eine quantifizierte Zuweisung benutzt, die für N alternative Zuweisungen steht, jede für ein anderes Elementepaar von A. Das Zeichen $[\!]$ ist der "Auswahl-Operator", der ausdrückt, daß immer nur eine der spezifizierten Zuweisungen ausgewählt wird. Die ausgewählte Zuweisung wird genau dann ausgeführt, wenn die Bedingung hinter dem *if* wahr ist. Zwei benachbarte Elemente werden also genau dann ausgetauscht, wenn sie nicht in der richtigen Reihenfolge stehen. Eine Mehrfach-Zuweisung

$$A[i], A[i+1] \;:=\; A[i+1], A[i]$$

wird in einem einzigen Schritt ausgeführt und könnte auch durch den "Parallel-Operator" $\|$ ausgedrückt werden:

$$A[i] \;:=\; A[i+1] \;\|\; A[i+1] \;:=\; A[i]$$

2.7.3 Ein paralleler Garbage Collector als UNITY-Programm

In [CHAN88] konzentrieren sich die Anstrengungen für Implementierung und Beweis eines Collectors auf dessen Markierungsphase. Der Collector wird daher nicht als reaktives System verstanden, sondern als eine Funktion, die immer wieder nebenläufig zum Mutator berechnet wird. Am Anfang sind alle Knoten — bis auf die Wurzel — unmarkiert. Während der Mutator Kanten erzeugen und löschen kann, hat der Collector die Aufgabe, solange Markierungen an die Knoten anzubringen, bis alle benutzten Knoten markiert sind. Damit ist bereits die Sicherheitsanforderung formuliert: am Ende der Markierungsphase sind höchstens die Müllknoten unmarkiert. Die Lebendigkeitsanforderung kann besonders einfach ausgedrückt werden: alle Knoten, die bereits vor dem Beginn der Markierungsphase Müll waren, müssen unmarkiert bleiben (Knoten, die bereits am Anfang Müll sind, werden als *manure* bezeichnet). Damit ist sichergestellt, daß Müllknoten höchstens eine Markierungsphase lang unerkannt bleiben; spätestens beim nächstenmal sind sie *manure* und bleiben dann unmarkiert. Das Programm *marker* implementiert die beschriebene "Markierungsfunktion":

Program *marker*
always
$$R = E^* \quad \{R \text{ ist die reflexive, transitive Hülle von } E\}$$

```
        [] ⟨[] u :: u.food = R[root,u]  [] u.garbage = ¬u.food⟩
    initially
        ⟨[] u :: u.manure = u.garbage  [] u.m = (u = root)⟩
        [] b = empty
        [] over = false    {ma.flag, mu.flag können beliebige Werte haben}
    assign
        ⟨[] u,v ::
              E[u,v], v.m := true, true                 if add[u,v]
                    || mu.flag := false                 if add[u,v] ∧ ¬v.m
              [] E[u,v] := false                        if del[u,v]
              [] v.m := v.m ∨ (u.m ∧ E[u,v])
                    || ma.flag := false                 if ¬v.m ∧ (u.m ∧ E[u,v])
              [] b := b ∪ {(u,v)}                       if u.m ∧ E[u,v] ⇒ v.m
        ⟩
        [] b, ma.flag, mu.flag := empty, true, true     if ¬ma.flag ∨ ¬mu.flag
        [] over := (b = VV) ∧ ma.flag ∧ mu.flag
end {marker}
```

Die Relation E wird durch den Mutator verändert, während der Markierer arbeitet. $E[u,v]$ ist genau dann *wahr*, wenn von Knoten u nach Knoten v eine gerichtete Kante existiert. Den Zusammenhang zu einem beliebigen Mutatorprozess stellen die Prädikate *add* und *del* her: $add[u,v]$ gilt, wenn eine Kante von u nach v errichtet werden soll, $del[u,v]$ gilt, wenn die Kante gelöscht werden soll. Darüberhinaus wird angenommen, daß gilt: $add[u,v] \Rightarrow v.food$, d.h. der Mutator richtet neue Kanten nur auf benutzte Knoten.

Sobald das Prädikat *over* gilt, ist der Markierer terminiert. Der zugrundeliegende Algorithmus ist stark beeinflußt durch das Programmiermodell von UNITY und hat auf den ersten Blick wenig Ähnlichkeit mit den bisher beschriebenen Collectoren. Am meisten Verwandschaft besteht zu Ben-Ari's Algorithmus: die Knotenmarkierung m bzw. $\neg m$ hat die gleiche Aufgabe wie die Farben *schwarz* und *weiß*. Auch die Terminierungserkennung ist ähnlich: Eine Menge b wird während des Markierungsvorgangs mit Kanten gefüllt, die eine bestimmte Eigenschaft aufweisen, nämlich:

$$(u,v) \in b \Rightarrow (u.m \wedge E[u,v] \Rightarrow v.m)$$

Wenn sich in b alle Kanten befinden (= VV), dann terminiert der Markierer. In Ben-Ari's Algorithmus wird "nebenher" die Anzahl der schwarzen Knoten gezählt. Nimmt sie nicht mehr zu, dann ist die Markierungsphase beendet.

2.7.4 Zusammenfassung

K.M. Chandy und J. Misra stellen keinen wirklich neuen Algorithmus vor, sondern legen auf die Technik Wert, mit der der Markierer entworfen und bewiesen wird. Die Qualität, die dabei in dieser Beziehung erreicht wird, übertrifft alle bisherigen Arbeiten zum

Thema der parallelen Garbage Collectoren. Trotzdem bleiben noch einige Wünsche offen:

Das in [CHAN88] modellierte System ist kein kompletter (reaktiver) Garbage Collector, sondern nur ein Auschnitt davon. Wenngleich es sich auch beim Markierer um den wichtigsten Teil eines Collectors handelt, so ist es doch wünschenswert, den reaktiven Charakter eines Systems bei seiner Modellierung zu erhalten und nicht auf die funktionalen Aspekte zu reduzieren.

Dem UNITY-Ausführungsmodell wird globale Fairness bei der Auswahl der bedingten Zuweisungen zugrundegelegt. Damit werden viele Lebendigkeitsprobleme, die in nebenläufigen Programmen auftreten können, auf die Ebene des Schedulers verlagert. Diese Verfahrensweise birgt zwei Nachteile:

1. Die Implementierung globaler Fairness ist ein nicht-triviales Problem. Unklar ist, ob globale Fairness überhaupt (im allgemeinen) korrekt implementiert werden kann.

2. Die Lokalisierung einer Fairness-Anforderung kann für das Verständnis eines parallelen Programms äußerst hilfreich sein. Tatsächlich benötigt ein nebenläufiger Garbage Collector nur eine lokale Fairness-Annahme, die die Schnittstelle zum Mutator betrifft (siehe Abschnitt 10.4.1). In einem Ausführungsmodell mit globaler Fairness geht diese ganz spezielle und lokale Anforderung in der allgemeinen Fairness-Annahme unter.

2.8 Weiterführende Literatur

Das Problem der Speicherbereinigung bzw. -Wiedergewinnung ist äußerst vielschichtig und kompliziert, gleichzeitig aber von großer Bedeutung für die maschinelle Informationsverarbeitung. Es verwundert daher kaum, daß es eine Vielzahl von Veröffentlichungen zum Thema “parallele Garbage Collectoren” gibt. Die in den vorhergehenden Abschnitten vorgestellten Arbeiten zeichnen sich entweder durch ihren innovativen Charakter aus oder durch die formale Darstellungsweise. Ein weiteres Auswahlkriterium war natürlich auch, in welchem Maße ihre Themen mit der vorliegenden Arbeit zusammenhängen.

Die im folgenden aufgeführten Literaturhinweise sollen unseren Streifzug durch die Literatur vervollständigen. Wir führen Veröffentlichungen auf, die sich mit den folgenden Themen beschäftigen: Verbesserung bestehender Algorithmen zur Effizienzsteigerung, Erweiterung von parallelen Garbage Collectoren auf Systeme mit mehreren Collectoren und mehreren Mutatoren auf einem gemeinsamen Speicher, nebenläufige Garbage Collectoren in verteilten Systemen ohne globalen Speicher, theoretische Analyse des Verhaltens und der Effizienz, sowie praktische Untersuchung der Effizienz durch Simulation.

2.8.1 Mehrere Prozessoren auf einem gemeinsamen Speicher

In [LAMP76] wird eine Erweiterung des DLMSS-Algorithmus für Systeme mit mehreren Collector-Prozessoren und/oder mehreren Listen-Prozessoren betrachtet. Die Prozessoren arbeiten dabei auf einem gemeinsamen Speicher. In [NEWM82] werden Lamports Vorschläge konkretisiert und die Effizienz einer eigenen Version durch Simulation überprüft. Das gleiche Thema wird erneut in [NEWM87] aufgegriffen; dort wird ein hybrider Algorithmus für mehrere Collectoren und mehrere Listenprozessoren vorgestellt, der verschiedene, bereits vorhandene Methoden kombiniert, um mehr Effizienz zu erreichen. Die Behauptungen werden durch Simulationen nachgewiesen. In [PIXL88] schließlich wird das gleiche Thema anhand von Ben-Ari's Algorithmus behandelt, wobei auch ein (informeller) Korrektheitsbeweis gegeben wird.

2.8.2 Hardware-Unterstützung, virtueller Speicher und andere Details

In [SHIN86] wird beschrieben, wie ein paralleler Garbage Collector durch Hardware-Unterstützung beschleunigt werden kann: ein *assoziativer Speicher* ermöglicht das schnelle Auffinden von Zellen mit einer bestimmten Farbmarkierung. Als Nebeneffekt ergibt sich, daß durch die physikalische Abtrennung der Speicherzellen von ihren Farbmarkierungen die Zugriffskonflikte zwischen Collector und Mutator besser vermieden werden.

Ein besonders kurzer und einfacher Nachweis der *Lebendigkeit* des DLMSS-Algorithmus findet sich in [RUBY87], wenngleich auch die Beweistechnik nicht überzeugen kann. In [LANG87] beschäftigen sich die Autoren vor allem mit dem Aspekt der *Speicherkompaktifizierung*, die für den praktischen Einsatz eines parallelen Garbage Collectors von großer Bedeutung ist. In [EHN89] werden einige Veränderungen des DLMSS-Algorithmus vorgestellt, die zu einer *höheren Effizienz* führen. Systeme mit *virtuellem Speicher* stellen eine besondere Herausforderung im Zusammenhang mit nebenläufigen Garbage Collectoren dar. Diesem speziellen Thema widmen sich die Autoren von [RAM85] und [ABRA87].

2.8.3 On-the-fly Garbage Collection in verteilten Systemen

Probleme ganz anderer Art ergeben sich bei der Wiedergewinnung von Speicher in verteilten Systemen. Diese Systeme zeichnen sich dadurch aus, daß es keinen globalen Speicher gibt. Jeder Prozessor besitzt seinen eigenen lokalen Speicher und kommuniziert mit anderen Prozessoren durch Nachrichtenaustausch über ein globales Netzwerk. Auch hier existiert ein Speichergeflecht, das durch einen gerichteten Graphen mit Knoten und Kanten modelliert werden kann, allerdings mit dem wesentlichen Unterschied, daß der Graph über das ganze System verteilt ist. Eine weitere Schwierigkeit ergibt sich aus der Tatsache, daß eine solche Architektur nur dann effizient genutzt werden kann, wenn jeder Prozessor hauptsächlich den eigenen Speicher manipuliert und nur gelegentlich mit anderen Prozessoren kommuniziert.

Einer der ersten Vorschläge für einen "verteilten Garbage Collector" stammt von John Hughes. In [HUGH85] stellt er einen Algorithmus vor, der es jedem Prozessor im

Netzwerk ermöglicht, innerhalb des eigenen Speichers Müll zu erkennen und gleichzeitig etwas für eine globale Speicher-Wiedergewinnung beizutragen. Die zentrale Idee ist, Speicherobjekte bei ihrer Erzeugung mit Zeitmarken einer globalen Uhr zu versehen. Mehrere verschränkt ablaufende, globale Collectoren stempeln dann den Wurzeln der lokalen Speichergraphen ihre eigene Startzeit auf. Die lokalen Collectoren propagieren die Zeitmarken der Wurzelobjekte auf ihre Nachfolgeknoten im lokalen Teil des Graphen. Am Ende der lokalen Propagierung versendet ein lokaler Collector Markierungsnachrichten an die entfernten Objekte, deren Zeitmarke sich innerhalb des lokalen Graphen verändert hat. Nachdem erkannt wurde, daß eine bestimmte globale Markierungsphase terminiert ist, werden alle Objekte freigegeben, deren Zeitmarke älter ist als die Startzeit der terminierten Markierungsphase.

Die Verwendung von Zeitmarken als Stempel auf Speicherobjekten erinnert an die Speicherwiedergewinnung im Zusammenhang mit dem Time-Warp-Verfahren zur Synchronisation der verteilten ereignisgesteuerten Simulation [JEFF85a]. Allerdings wird dort keine globale Uhr eingesetzt, sondern jeder Prozessor besitzt eine eigene lokale Uhr, die eine für ihn gültige "virtuelle Zeit" anzeigt [JEFF85b]. Die lose Synchronisation der lokalen Uhren wird durch das von David R. Jefferson et al. entworfene Time-Warp-Verfahren ermöglicht. Weitere interessante Arbeiten zum Thema der verteilten Garbage Collectoren finden sich in [ALI86], [AUGU87], [BEVA89] und [SCHE89].

2.8.4 Theoretische Analysen der Effizienz

In [HICK84] werden Lastmodelle für parallele Garbage Collectoren aufgestellt. Sie sind Grundlage für die Analyse verschiedener Betriebsmodi eines Collectors. In [JULI84] stehen die Speicherobjekte im Vordergrund. Vorgestellt wird ein Modell für die Veränderungen im Arbeitsspeicher in Abhängigkeit von verschiedenen stochastischen Parametern für die Verteilung der Größe und der Lebensdauer von Speicherzellen sowie der Zugriffshäufigkeit.

TEIL **2**

Theoretische Grundlagen

3 Systemmodellierung mit Petrinetzen

Auf der Suche nach einer Programmnotation, die geeignet ist, um verteilte Systeme zu modellieren, wird man früher oder später auf Petrinetze stossen. Durch keine andere "Sprache" kann wohl Parallelität und Verteiltheit adäquater beschrieben werden, wie durch Netze. Im folgenden Abschnitt werden einige grundlegende netztheoretische Begriffe in *Elementaren Netz-Systemen* vorgestellt und anhand eines einführenden Beispiels erläutert. Im darauffolgenden Abschnitt beschäftigen wir uns dann mit der Erweiterung des Netzbegriffs auf Netze mit strukturierten Marken, die bei größeren Systemdarstellungen einfacher handhabbar sind und sich durch eine bessere Lesbarkeit auszeichnen.

3.1 EN-Systeme

Abbildung 3.1 zeigt ein EN-System (Elementares-Netz-System), welches das Problem des gegenseitigen Ausschlusses modelliert. Wir haben zwei Kinder mit einem Spielzeug und gehen davon aus, daß das Spielzeug nur für eines der beiden Kinder reicht. Während ein Kind spielt, muß das andere Kind warten. Jedes der beiden Kinder befindet sich in einem von drei Zuständen: 1) ein Kind schreit, 2) es will spielen und wartet auf das Spielzeug oder 3) das Kind spielt (sofern es sich durchsetzte und das Spielzeug in seinen Besitz bringen konnte).

In der Repräsentation dynamischer Systeme spielen Zustände und Ereignisse eine wichtige Rolle. Ereignisse verändern Zustände, und Zustände ermöglichen Ereignisse. Um die Verteiltheit eines dynamischen Systems begreifen zu können, müssen wir von der Vorstellung eines globalen — und daher zentralen — Systemzustands abrücken und zu einem lokalisierten — und daher verteilten — Zustandsbegriff gelangen. In der Netztheorie werden dynamische Systeme als eine Menge von Zustandsträgern und einer Menge von lokalen Zustandsübergängen aufgefaßt, die zueinander in einer festen Nachbarschaftsbeziehung stehen. Als abstrakte graphische Notation ergibt sich ein zweisortiger Graph, der durch ein mathematisches Objekt Netz beschrieben wird:

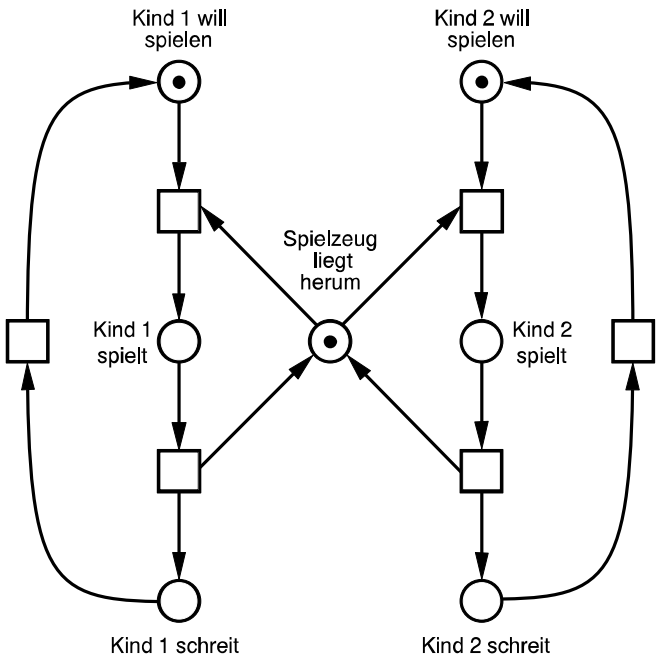


Abb. 3.1. Modell des gegenseitigen Ausschlusses

3.1.1 Definition *Netz*

Ein Tripel $N = (S,T,F)$ heißt *Netz* g.d.w.
(i) S und T sind disjunkte, nicht-leere Mengen.
(ii) $F \subseteq (S \times T) \cup (T \times S)$, die zweistellige Flußrelation von N.

Die Elemente der Menge T werden als Quadrate dargestellt und als lokale Zustandsübergänge interpretiert. Ein Zustandsübergang verändert die Zustandsträger in der Nachbarschaft des T-Elements. Entsprechend bezeichnen wir die Elemente der Menge S als lokale Zustandsträger bzw. Zustände und stellen sie als Kreise dar. Die Nachbarschaftsbeziehung zwischen S- und T-Elementen wird durch die Flußrelation F festgelegt, die wir graphisch als Pfeile notieren.

3.1.2 Schreibweise

Sei $N = (S,T,F)$ ein Netz.
(i) Treten keine Mehrdeutigkeiten auf, so schreiben wir N für $S \cup T$.
(ii) Die Komponenten S, T und F werden mit dem Index N versehen (S_N, T_N, F_N), falls die Zuordnung nicht aus dem Kontext ersichtlich ist.

3.1.3 Definition *Vorbereich, Nachbereich*

Sei $N = (S,T,F)$ ein Netz. Für $x \in N$ heißt
(i) $\bullet x = \{y \mid y\ F\ x\}$ der *Vorbereich* von x.
(ii) $x\bullet = \{y \mid x\ F\ y\}$ der *Nachbereich* von x.

Im einfachsten Fall interpretieren wir die S- und T-Elemente als Bedingungen und Ereignisse, so auch in Abbildung 3.1. Jedem S-Element ordnen wir einen Wahrheitswert zu, d.h. eine Bedingung ist entweder gültig oder nicht. Gültige Bedingungen werden graphisch durch einen kleinen schwarzen Kreis, d.h. einer Marke, innerhalb des S-Elements symbolisiert. Die Menge aller gültigen Bedingungen charakterisiert den Systemzustand. Ein Systemzustand wird auch als Fall bezeichnet. Die T-Elemente sind Ereignisse, die elementare Zustandsänderungen bewirken.

Ein Ereignis kann eintreten, wenn alle Bedingungen seines Vorbereichs gelten, die seines Nachbereichs aber nicht. Wir sprechen dann auch von einem aktivierten Ereignis. Tritt ein Ereignis ein, dann werden alle Marken aus dem Vorbereich in den Nachbereich transportiert, d.h. die Vorbedingungen verlieren ihre Gültigkeit, während die Nachbedingungen gültig werden.

3.1.4 Definition *Fall, Aktivierung, Folgefall*

Sei $N = (B,E,F)$ ein Netz und $e \in E$.
(i) Eine Teilmenge $M \subseteq B$ heißt *Fall*.
(ii) Sei M ein Fall. e heißt *M-aktiviert*[1] g.d.w.
$\bullet e \subseteq M$ *und* $e\bullet \cap M = \emptyset$.
(iii) Sei M ein Fall und sei e M-aktiviert.
$M' = (M \setminus \bullet e) \cup e\bullet$ heißt *Folgefall* von M unter e.
Wir erhalten M' durch das Eintreten von e im Fall M und schreiben $M\ [e\rangle\ M'$.

Sobald wir über Netze reden, deren S- bzw. T-Elemente als Bedingungen und Ereignisse interpretiert werden, nennen wir die Menge der S-Elemente B und die Menge der T-Elemente E.

Abbildung 3.1 zeigt ein System aus Bedingungen und Ereignissen zusammen mit einem Anfangsfall, in dem beide Kinder spielen wollen und das Spielzeug noch herumliegt. Es handelt sich um die graphische Darstellung eines EN-Systems.

3.1.5 Definition *EN-System*

Ein Viertupel $N = (B,E,F,M^\circ)$ heißt *EN-System* (Elementares-Netz-System) g.d.w.
(i) (B,E,F) ist ein Netz.

---

1) Gemäß dieser Definition sind Ereignisse, die durch Schlingen mit einer Bedingung verbunden sind, niemals aktiviert. Eine Schlinge entsteht, wenn eine Bedingung sowohl im Vor- als auch im Nachbereich eines Ereignisses liegt. Wir werden später bei den High-Level-Netzen diese Beschränkung aufheben.

(ii) $M^\circ \subseteq B$ ist der *Anfangsfall* des Systems.
(iii) Es gibt keine isolierten Elemente, d.h. $dom(F) \cup codom(F) = B \cup E$, wobei
  (a) $dom(F) = \{x \mid \exists y : x\ F\ y\}$
  (b) $codom(F) = \{x \mid \exists y : y\ F\ x\}$.

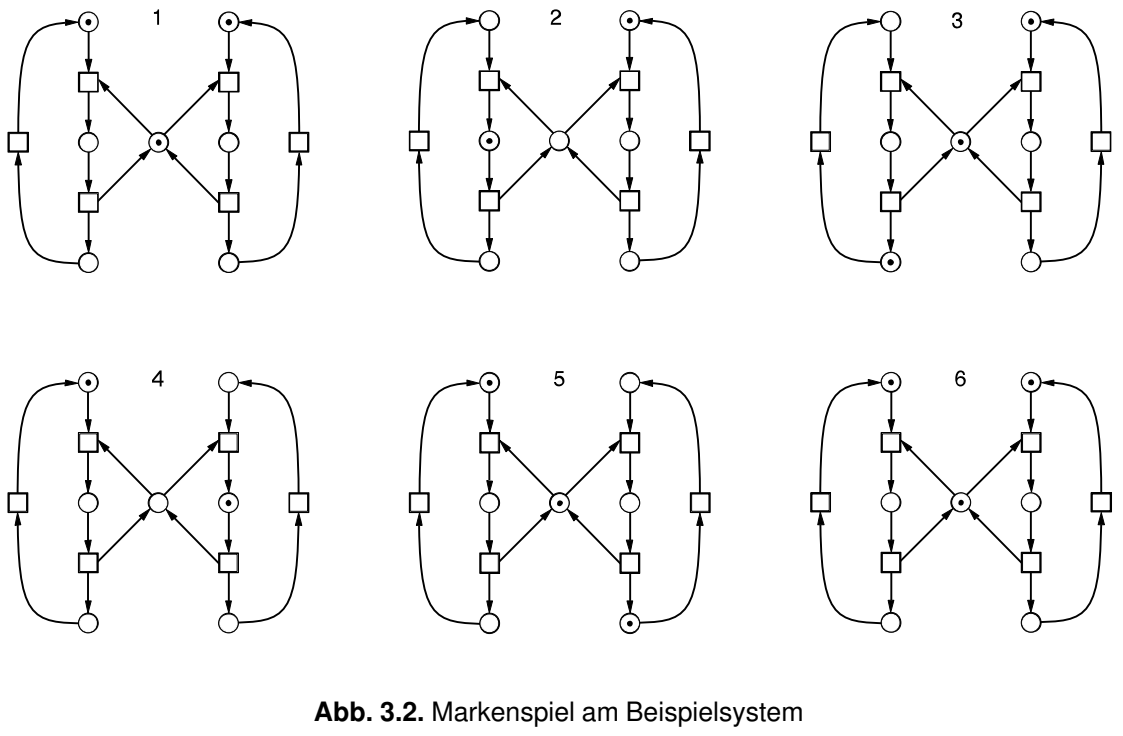


Abb. 3.2. Markenspiel am Beispielsystem

Abbildung 3.2 zeigt eine Folge von Situationen, die sich aus dem Anfangsfall durch das Eintreten von Ereignissen ergeben, veranschaulicht anhand des "Markenspiels": Zuerst spielt Kind 1, dann spielt Kind 2. Am Anfang sind die beiden Ereignisse, die die Kinder zum Spielen bringen, in Konflikt zueinander, da nur eines der beiden Kinder das Spielzeug nehmen kann.

3.1.6 Abläufe

Ein Petrinetz beschreibt ein System durch die Angabe einer Konstruktionsvorschrift für Abläufe. Wir betrachten daher die Menge aller konstruierbaren Abläufe als die Semantik (Bedeutung) eines Programms[2]. Die Formalisierung eines Ablaufbegriffs wird erst in Abschnitt 3.6.1 im Zusammenhang mit Netzen mit strukturierten Marken ausgeführt. Statt

2) In dieser Arbeit werden Petrinetze ausschließlich als Implementierungssprache benutzt. Wir bezeichnen daher Petrinetze als Programme.

dessen soll an dieser Stelle eine kurze informelle Diskussion der möglichen Ablaufsemantiken einen groben Überblick vermitteln.

- **Halbordnungs-Semantik**: Ein Prozess ist eine Halbordnung über der Menge der Ereignisse. Zwei Ereignisse a, b sind geordnet, d.h. $a < b$, falls sie zum gleichen Prozess gehören, und a vor b eintritt [LAMP78]. Die beiden Ereignisse stehen in kausalem (d.h. ursächlichem) Zusammenhang. Gilt weder $a < b$ noch $b < a$, dann sind die beiden Ereignisse kausal unabhängig und treten nebenläufig zueinander ein. Es gibt keinen globalen Zustandsbegriff.

- **Schritt-Semantik**: Ein Ablauf wird als Folge von globalen Zuständen aufgefaßt, wobei zwischen zwei Zuständen eine beliebige Anzahl von paarweise unabhängigen Ereignissen zugleich (d.h. parallel) eintreten kann (= **Schritt**). Zwei Ereignisse werden als unabhängig bezeichnet, wenn sie nicht in Konflikt zueinander stehen, d.h. sie schließen sich nicht gegenseitig aus.

- **Interleaving-Semantik**: Auch hier wird ein Ablauf als Folge von globalen Zuständen aufgefaßt, allerdings tritt zwischen zwei Zuständen immer nur ein Ereignis ein, d.h. Schritte von parallel eintretenden Ereignissen werden in einzelne Ereignisse aufgebrochen. Die Parallelität wird durch beliebiges Hintereinanderausführen von unabhängigen Aktionen simuliert.

Offensichtlich spiegelt die Halbordnungs-Semantik am ehesten den Charakter eines verteilten Systems wieder, da kein globaler Zustandsbegriff existiert. Die anderen beiden Semantiken sind zwar weit verbreitet, nutzen aber die Ausdruckskraft der Petrinetze nicht voll aus. Andererseits zeichnet sich vor allem die Interleaving-Semantik durch eine bestechende Einfachheit aus, was sich bei der Formalisierung der Theorie als Vorteil erweist. Daher verwenden wir in dieser Arbeit die Interleaving-Semantik.

3.2 Abstrakte Datentypen

In EN-Systemen werden Petrinetze auf eine sehr einfache Art und Weise interpretiert. Eine direkte Modellierung durch EN-Systeme ist daher nur für kleine und überschaubare Probleme möglich. Vielmehr dienen EN-Systeme eher zur Veranschaulichung des grundsätzlichen Charakters von verteilten Systemen, sie sind gleichsam die Turing-Maschinen für die Analyse nebenläufiger Vorgänge[3]. Obwohl man sich vorstellen kann, daß alle verteilten Abläufe letztendlich auf der Ebene der EN-Systeme beschrieben werden und daher EN-Systeme das Referenzmodell für verteilte Maschinen sind [REIS90b], so ist es doch wünschenswert, eine Klasse von Netzen verfügbar zu haben, mit der auch umfangreichere Systeme anschaulich modelliert werden können.

---

3) Turing-Maschinen und EN-Systeme sind allerdings keine gleichwertigen Kontrahenten, was die Ausdrucksstärke betrifft: Turing-Maschinen besitzen i.a. unendliche Zustandsräume, während EN-Systeme immer über endliche Resourcen argumentieren.

Zu diesem Zweck führen wir Netze mit strukturierten Marken ein. Ein S-Element repräsentiert nicht mehr einfach nur eine Bedingung, die erfüllt oder unerfüllt sein kann, sondern ein ganzes "Bedingungsschema", welches differenzierte Informationen über Datenobjekte enthält. Wir bezeichnen solche Bedingungsschemata auch als *Prädikate*. Analog dazu wird aus einem Ereignis ein Ereignisschema, das für eine ganze Menge von Ereignissen steht. Ein Ereignisschema bezeichnen wir auch als *Prädikat-Transformer*. Die strukturierten Marken, die sich in den Prädikaten befinden können, sind in unserem Netzkalkül die Elemente der Trägermenge eines abstrakten Datentyps. Diese wiederum werden syntaktisch durch Terme beschrieben. Die Grundlage dieses Netzkalküls bildet die Theorie zur algebraischen Spezifikation von abstrakten Datentypen. Im folgenden werden einige Begriffe und Konzepte dieser Theorie aus der Literatur rekapituliert [REIS90a, EHRI85].

3.2.1 Definition *Signatur*

Eine *Signatur* $\Sigma = (S,OP)$ besteht aus
(i) einer Menge S von *Sorten* und
(ii) einer Familie $OP = (OP_{w,s})_{w \in S^*, s \in S}$ von *Operationssymbolen*.
Wir unterscheiden insbesondere die Mengen $K_s := OP_{\lambda,s}$ der *Konstantensymbole* (λ bezeichnet das leere Wort über S).

3.2.2 Definition Σ-*Algebra*

Sei $\Sigma = (S,OP)$ eine Signatur. Eine Σ-*Algebra* $A = (S_A,OP_A)$ besteht aus
(i) einer Familie $S_A = (A_s)_{s \in S}$ von *Grundmengen* und
(ii) einer Familie $OP_A = (N_A)_{N \in OP}$ von *Operationen* $N_A : A_{s_1} \times \ldots \times A_{s_n} \to A_s$ für alle $N \in OP_{s_1 \ldots s_n,s}$.
Offensichtlich gilt $N_A \in A_s$ g.d.w. $N \in K_s$.

3.2.3 Definition Σ-*Variable*

Sei $\Sigma = (S,OP)$ eine Signatur. $X = (X_s)_{s \in S}$ heißt Familie von Σ-*Variablen* g.d.w. alle Elemente aus X_s disjunkt zu OP sind.

3.2.4 Definition *Terme*

Sei $\Sigma = (S,OP)$ eine Signatur und X eine Familie von Σ-Variablen. Die Menge $T_{OP,s}(X)$ von *Termen* der Sorte s über OP und X ist induktiv definiert durch
(i) $X_s \cup K_s \subseteq T_{OP,s}(X)$, und
(ii) falls $u_1 \in T_{OP,s_1}(X)$, …, $u_n \in T_{OP,s_n}(X)$, dann auch $N(u_1,...,u_n) \in T_{OP,s}(X)$ für $N \in OP_{s_1 \ldots s_n,s}$ und $n \geq 1$.

3.2.5 Definition *Grundterme, Σ-Terme, Σ-Grundterme*

Sei $\Sigma = (S,OP)$ eine Signatur und X eine Familie von Σ-Variablen.

(i) Die Menge $T_{OP,s} := T_{OP,s}(\emptyset)$ enthält die *Grundterme der Sorte s*. Grundterme sind also variablenfrei.

(ii) $T_{OP}(X) := \bigcup_{s \in S} T_{OP,s}(X)$ ist die Menge der Σ-*Terme über X*.

(iii) $T_{OP}(\emptyset)$ ist die Menge der Σ-*Grundterme*.

3.2.6 Definition *Auswertung (evaluation)*

Sei $\Sigma = (S,OP)$ eine Signatur und A eine Σ-Algebra. Eine *Auswertung* ist eine Abbildung $eval : T_{OP} \to A$ von Σ-Grundtermen in die Σ-Algebra A und ist induktiv definiert durch

(i) $eval(N) = N_A$ für alle Konstantensymbole $N \in K_s$ und

(ii) $eval(N(u_1,...,u_n)) = N_A(eval(u_1),...,eval(u_n))$ für alle $N(u_1,...,u_n) \in T_{OP}$.

3.2.7 Definition *Belegung (assignment)*

Sei $\Sigma = (S,OP)$ eine Signatur, X eine Familie von Σ-Variablen und A eine Σ-Algebra. Eine *Belegung* ist eine Abbildung $ass : X \to A$ mit $ass(x) \in A_s$ g.d.w. $x \in X_s$. ass wird kanonisch zu $\overline{ass} : T_{OP}(X) \to A$ erweitert, wobei $\overline{ass}$ induktiv definiert ist durch

(i) $\overline{ass}(x) = ass(x)$ für $x \in X$,

(ii) $\overline{ass}(N) = N_A$ für $N \in K_s$ und

(iii) $\overline{ass}(N(u_1,...,u_n)) = N_A(\overline{ass}(u_1),...,\overline{ass}(u_n))$ für $N(u_1,...,u_n) \in T_{OP}(X)$.

3.2.8 Definition Σ-*Gleichung, Gültigkeit*

Sei $\Sigma = (S,OP)$ eine Signatur und X eine Familie von Σ-Variablen.

(i) Eine Σ-*Gleichung der Sorte s* über X ist ein Paar (L,R) von Termen $L,R \in T_{OP,s}(X)$.

(ii) Sei A eine Σ-Algebra. Eine Σ-Gleichung (L,R) über X ist *gültig in A* g.d.w. für alle Belegungen $ass : X \to A$ gilt: $\overline{ass}(L) = \overline{ass}(R)$.

3.2.9 Definition *(Algebraische) Spezifikation, SPEC-Algebra*

Sei $\Sigma = (S,OP)$ eine Signatur und A eine Σ-Algebra.

(i) Eine *Spezifikation* $SPEC = (S,OP,E)$ besteht aus der Signatur $\Sigma = (S,OP)$ und einer Menge E von Σ-Gleichungen.

(ii) Sei $SPEC = (S,OP,E)$ eine Spezifikation mit der Signatur Σ. A heißt *SPEC-Algebra* g.d.w alle Gleichungen aus E in A gültig sind.

Der Begriff “Spezifikation” wird in dieser Arbeit im Zusammenhang mit zwei verschiedenen Aspekten eines Systems benutzt. Zum einen wird durch eine “algebraische” Spezifikation ein abstrakter Datentyp definiert, auf dem ein Petrinetz operiert. Zum anderen

wird durch eine temporal-logische Spezifikation das dynamische Verhalten des Netzes beschrieben. Falls die jeweilige Bedeutung des Begriffs “Spezifikation” nicht aus dem Kontext ersichtlich ist, dann schreiben wir “algebraische Spezifikation” für den Begriff, der eben definiert wurde, bzw. “Systemspezifikation” für die andere Bedeutung.

3.2.10 Definition *Kongruenz von Grundtermen*

Sei $SPEC = (S,OP,E)$ eine Spezifikation. Zwei Grundterme $u,v \in T_{OP}$ heissen *kongruent* in SPEC g.d.w. $eval_A(u) = eval_A(v)$ für alle SPEC-Algebren A. Wir schreiben $u \equiv_E v$.

3.2.11 Lemma

Sei $SPEC = (S,OP,E)$ eine Spezifikation. Die Relation $\equiv_E$ ist eine Kongruenzrelation über T_{OP}, insbesondere bleibt bei der Substitution von kongruenten Termen die Kongruenz erhalten: $u \equiv_E v$ impliziert $N(\ldots u \ldots) \equiv_E N(\ldots v \ldots)$ (vgl. [EHRI85] Fakt 3.11). Wenn E aus dem Kontext ersichtlich ist, dann schreiben wir $[u]$ für die Äquivalenzklasse des Terms u unter $\equiv_E$.

3.2.12 Definition *Quotienten-Term-Algebra*

Sei $SPEC = (S,OP,E)$ eine Spezifikation. Die *Quotienten-Term-Algebra* $T_{SPEC} = ((Q_s)_{s \in S}, (N_Q)_{N \in OP})$ ist definiert durch

(i) die Äquivalenzklassen $Q_s = \{[u] \mid u \in T_{OP,s}\}$ von $\equiv_E$ als Trägermengen und

(ii) die Operationen N_Q definiert durch $N_Q([u_1],\ldots,[u_n]) := [N(u_1,\ldots,u_n)]$.

3.2.13 Definition *Kongruenz von Termen*

Sei $SPEC = (S,OP,E)$ eine Spezifikation und X eine Familie von (S,OP)-Variablen. Die Kongruenz $\equiv_E$ auf Grundtermen wird erweitert auf Terme $u,v \in T_{OP}(X)$ durch: $u \equiv_E v$ g.d.w. für alle Belegungen $ass : X \to T_{OP}$ gilt $\overline{ass}(u) \equiv_E \overline{ass}(v)$.

3.2.14 Definition *Initiale Semantik, Abstrakter Datentyp, Korrektheit*

Sei $SPEC$ eine Spezifikation mit der Signatur Σ.

(i) Die *initiale Semantik* von $SPEC$ ist die Klasse $ADT(SPEC)$ aller zur Quotienten-Term-Algebra T_{SPEC} isomorphen Σ-Algebren. $ADT(SPEC)$ ist der *abstrakte Datentyp* der durch $SPEC$ definiert wird.

(ii) Sei A eine Σ-Algebra. Die Spezifikation $SPEC$ heißt *korrekt* im Sinne von A, wenn A isomorph zu T_{SPEC} ist, d.h. $A \in ADT(SPEC)$.

Um also einen abstrakten Datentyp zu definieren, reicht es aus, seine Spezifikation anzugeben, bestehend aus einer Signatur und einer Menge von Gleichungen. Im folgenden werden die vorgestellten Begriffe anhand eines Beispiels erläutert.

Eine algebraische Spezifikation der natürlichen Zahlen mit der Null, der Nachfolger-Operation und der Addition ist gegeben durch:

nat =
sorts: *nat*
opns: $0 : \rightarrow nat$
$SUCC : nat \rightarrow nat$
$ADD : nat\ nat \rightarrow nat$
eqns: $n,m \in nat$
$ADD(n,0) = n$
$ADD(n,SUCC(m)) = SUCC(ADD(n,m))$

Den Sorten der Signatur wird dabei das Schlüsselwort “sorts” vorangestellt, den Operationsymbolen das Schlüsselwort “opns” und den Gleichungen das Wort “eqns”. Die “0" ist ein Konstantensymbol bzw. ein Operationssymbol für eine Operation mit 0 Argumenten. Wenn (L,R) eine Gleichung über Variablen $X = \{x_1,\ldots,x_n\}$ ist und $x_i \in X_{s_i}$, dann schreiben wir in der anschaulicheren Form

$$x_1 \in s_1, \ldots, x_n \in s_n$$
$$L = R$$

Die Algebra $NAT = (\mathbb{N},0,+1,+)$ der natürlichen Zahlen mit den uns vertrauten Operationen für Nachfolgerbildung und Addition ist isomorph zu der aus der Spezifikation **nat** konstruierbaren Quotienten-Term-Algebra $T_{\mathbf{nat}}$, d.h. $NAT \in ADT(\mathbf{nat})$. Die Trägermenge der Quotienten-Term-Algebra $T_{\mathbf{nat}}$ sind gerade die Äquivalenzklassen der Terme über **nat**, die durch die Operationssymbole 0, *SUCC* und *ADD* generiert werden können. Die Operationen von $T_{\mathbf{nat}}$ werden aus den Gleichungen von **nat** abgeleitet. Die Konstruktion der Quotienten-Term-Algebra aus ihrer Spezifikation wird an dieser Stelle nicht ausgeführt. Beispiele zur Konstruktion von Quotienten-Term-Algebren finden sich in [EHRI85] (z.B. Abschnitt 2.3: $T_{\mathbf{string}}$).

3.2.15 Definition *Erweiterung einer Spezifikation*

Sei $SPEC = (S,OP,E)$ eine Spezifikation und seien $S1$, $OP1$ und $E1$ zusätzliche Sorten, Operationssymbole und Gleichungen. Eine Spezifikation $SPEC1 = (S \dot\cup S1, OP \dot\cup OP1, E \dot\cup E1)$ heißt *Erweiterung* der Spezifikation *SPEC*, wobei $\dot\cup$ für die disjunkte Vereinigung von Mengen steht. Wir schreiben $SPEC1 = SPEC + (S1,OP1,E1)$.

3.3 Multimengen-Spezifikationen

Indem wir die Zustandsträger eines Petrinetzes als abstrakte Datentypen auffassen, gelangen wir zu einem sehr leistungsfähigen Netzkalkül. Die S-Elemente repräsentieren dabei

im Normalfall Mengen von Objekten aus der Trägermenge eines abstrakten Datentyps. Dadurch werden die S-Elemente zu Prädikate über einer bestimmten Sorte. Diese Anschauungsweise führt zu einem sehr natürlichen Operationsmodell für Petrinetze, in dem die verteilten Zustände im System zu dynamischen Prädikaten werden (d.h. Prädikate mit variabler Extension) und die T-Elemente zu "Prädikat-Transformern", die die Extension ihrer benachbarten Prädikate ändern.

Um die Eigenschaften eines Petrinetzes elegant analysieren zu können, muß diese Sichtweise jedoch erweitert werden. S-Elemente werden nicht mit Mengen von Objekten assoziiert, sondern mit Multimengen von Objekten, d.h. in jedem S-Element können sich mehrere Kopien des gleichen Objekts befinden. Die einzelnen Kopien sind dabei nicht unterscheidbar.

3.3.1 Definition *(nicht-negative) Multimengen*

Sei D eine beliebige Menge.
(i) Eine Abbildung $M : D \to \mathbb{Z}$ heißt *Multimenge* über der Menge D.
(ii) Eine Abbildung $M : D \to \mathbb{N}$ heißt *nicht-negative Multimenge* über D.
(iii) Die leere Multimenge $\tilde{\emptyset}_D$ über D ist gegeben durch $\tilde{\emptyset}_D(d) = 0$ für alle $d \in D$.

Einzelne Elemente $d \in D$ können dabei als ein-elementige Multimengen m_d aufgefaßt werden, bei denen $m_d(x) = 1$, falls $x = d$ und $m_d(x) = 0$, sonst.

3.3.2 Definition *Addition und Subtraktion von Multimengen*

Seien M_1 und M_2 Multimengen über einer Menge D und $d \in D$.
(i) Die *Addition* $M_1 + M_2$ ist komponentenweise definiert durch
$(M_1 + M_2)(d) = M_1(d) + M_2(d)$.
(ii) Die *Subtraktion* $M_1 - M_2$ ist komponentenweise definiert durch
$(M_1 - M_2)(d) = M_1(d) - M_2(d)$.

Jede Spezifikation *SPEC* kann zu ihrer korrespondierenden Multimengen-Spezifikation **m**_*SPEC* erweitert werden. Dabei wird jedem Konstantensymbol $N \in K_s$ ein Term $MAKE_s(N)$ zugeordnet, der die ein-elementige Multimenge bezeichnet. Der Term $\tilde{\emptyset}_s$ bezeichnet die leere Multimenge der Sorte s, und Addition und Subtraktion von Multimengen werden passend definiert:

3.3.3 Definition *Multimengen-Spezifikation*

Sei $SPEC = (S,OP,E)$ eine Spezifikation. Die zu *SPEC* korrespondierende Multimengen-Spezifikation **m**_*SPEC* ist definiert durch

m_*SPEC* = *SPEC* + { für jede Sorte $s \in S$ }
sorts: m_s
opns: $\tilde{\emptyset}_s : \to m_s$

$MAKE_s : \ s \ \rightarrow \ m_s$
$+_s : \ m_s \ m_s \ \rightarrow \ m_s$
$-_s : \ m_s \ m_s \ \rightarrow \ m_s$
$-_s : \ m_s \ \rightarrow \ m_s$

eqns: $p,q,r \in m_s$
$+_s(p,\tilde{O}_s) \ = \ p$
$+_s(p,q) \ = \ +_s(q,p)$
$+_s(p,+_s(q,r)) \ = \ +_s(+_s(p,q),r)$
$+_s(-_s(p),q) \ = \ \tilde{O}_s$
$-_s(p,q) \ = \ +_s(p,-_s(q))$

3.3.4 Schreibweise

(i) Zu einer gegebenen Spezifikation $SPEC = (S,OP,E)$ bezeichnet $(\hat{S},\hat{OP},\hat{E})$ die zugehörige Erweiterung **m**_$SPEC$.
(ii) Innerhalb von Multimengen-Termen werden die Sorten-Indices s weggelassen, wenn sie aus dem Kontext ersichtlich sind, z.B. $\tilde{O}$ statt $\tilde{O}_s$.
(iii) Statt $+(u,v)$ oder $-(u,v)$ benutzen wir die Infix-Schreibweise $u + v$ bzw. $u - v$.
(iv) Statt $MAKE(a)$ schreiben wir a.
(v) Klammern werden weggelassen, sofern die üblichen Vorrangregeln die beabsichtigte Klammerung offensichtlich werden lassen.

Für die Konstantensymbole a und b einer Sorte s, bezeichnet beispielsweise $a - b$ sowohl den Term $-_s(MAKE_s(a),MAKE_s(b))$, als auch den Term $+_s(MAKE_s(a),-_s(MAKE_s(b)))$.

Um später die Terme, die nicht-negative Multimengen beschreiben, von den anderen unterscheiden zu können, führen wir die folgenden Konzepte ein:

3.3.5 Definition *Nicht-negative Terme*

Sei $SPEC = (S,OP,E)$ eine Spezifikation und X eine Familie von (S,OP)-Variablen.

(i) Die Menge der *nicht-negativen Operationssymbole NOS* für $SPEC$ ist definiert durch $NOS \ := \ \{\tilde{O}_s \mid s \in S\} \ \cup \ \{MAKE_s \mid s \in S\} \ \cup \ \{+_s \mid s \in S\}$.
(ii) Die Menge der *nicht-negativen Terme* $T_{OP}{}^{+}(X)$ von $SPEC$ ist definiert durch $T_{OP}{}^{+}(X) \ := \ \{u \in T_{\hat{O}\hat{P}}(X) \mid u \equiv_{\hat{E}} v \ \text{ für irgendein } \ v \in T_{NOS}(X)\}$.
(iii) Analog zu 3.2.5 definieren wir $T_{OP}{}^{+} \ := \ T_{OP}{}^{+}(\emptyset)$ (die nicht-negativen Grundterme) und für alle Sorten $s \in S$: $T_{OP}{}^{+}{}_{,m_s}(X) \ := \ T_{OP}{}^{+}(X) \ \cap \ T_{\hat{O}\hat{P},m_s}(X)$ (nicht-negative Terme der Sorte s), sowie $T_{OP}{}^{+}{}_{,s} \ := \ T_{OP}{}^{+}{}_{,m_s}(\emptyset)$ (die nicht-negativen Grundterme der Sorte s).
(iv) Seien $u,v \in T_{\hat{O}\hat{P}}(X)$ zwei Multimengen-Terme über X, dann heißt
  (a) *u kleiner oder gleich v* in $SPEC$ g.d.w. $v - u$ ist nicht-negativ in $SPEC$. Wir schreiben dann $u \leq_{\hat{E}} v$.

(b) *u größer als v* in *SPEC* g.d.w. nicht $u \leq_{\hat{E}} v$ gilt. Wir schreiben dann $u >_{\hat{E}} v$.

Ein Multimengen-Term ist also genau dann nicht-negativ, wenn er kongruent zu einem Multimengen-Term ist, der nur aus den Symbolen Õ, *MAKE* und + aufgebaut ist. Die Kongruenz zweier Terme wird dabei durch die Gleichungen bestimmt, die der erweiterten Spezifikation **m**_*SPEC* zugrundeliegen. So ist z.B. der Term $a + a - b$ kongruent zu einem Term *a* und damit nicht-negativ, wenn die Gleichung $a = b$ vorhanden ist.

3.4 Netze mit strukturierten Marken

Nachdem wir in den vorhergehenden Abschnitten die grundlegenden Konzepte der algebraischen Spezifikation formuliert haben, können wir jetzt über Netze sprechen, die auf den Objekten eines abstrakten Datentyps operieren. Die einzelnen Objekte, die mit den jeweiligen S-Elementen eines Netzes assoziiert werden, stellen sich dabei als strukturierte Marken dar. Die T-Elemente bewegen diese Marken und verändern dabei u.U. die dahinterstehenden Objekte gemäß ihrer algebraischen Spezifikation.

Ein Netz mit strukturierten Marken ist daher ein Netz, das im Sinne der Signatur einer algebraischen Spezifikation beschriftet wird. Bei der Beschriftung durch eine Spezifikation *SPEC* legen wir immer die zugehörige Multimengen-Spezifikation **m**_*SPEC* zugrunde. Jeder Stelle *p* (d.h jedem S-Element) ordnen wir eine Sorte $\varphi(p)$ und eine Startmarkierung $M^\circ(p)$ zu, wobei letzteres ein nicht-negativer Multimengen-Grundterm ist. Jeder Pfeil *f* der Flußrelation wird durch einen nicht-negativen Multimengen-Term $\lambda(f)$ beschriftet. Sowohl M° als auch λ müssen die Sorten der jeweiligen Stellen respektieren, d.h. $M^\circ(p)$ ist von der Sorte $m_{\varphi(p)}$ und $\lambda(f)$ hat die gleiche Sorte, wie die zu *f* gehörende Stelle. Wir bezeichnen im übrigen die T-Elemente als Transitionen und die S-Elements als Stellen, sobald wir über beschriftete Netze reden, und schreiben *P* (für engl. *Places*) anstelle von *S*, um eine Verwechslungsgefahr mit der Sortenmenge *S* einer Signatur auszuschliessen. Die folgenden Definitionen formalisieren die genannten Begriffe [REIS90a]:

3.4.1 Definition *SPEC-Beschriftung von Netzen*

Sei $N = (P,T,F)$ ein Netz, $SPEC = (S,OP,E)$ eine Spezifikation, $\Sigma = (S,OP)$ ihre Signatur und *X* eine Familie von Σ-Variablen.

(i) Eine Abbildung $\varphi : P \to S$ heißt *Sortenbelegung* von *N*. Für jede Stelle $p \in P$ bezeichnet dann $\tilde{p}$ die Multimengen-Sorte $m_{\varphi(p)}$.

(ii) Eine Abbildung $M^\circ : P \to T_{OP^+}$ mit $M^\circ(p) \in T_{OP^+,\tilde{p}}$ für jede Stelle $p \in P$ heißt φ-*respektierende Startmarkierung* von *N*.

(iii) Eine φ-*respektierende Kantenbeschriftung* von *N* ist eine Abbildung $\lambda : F \to T_{OP^+}(X)$ mit $\lambda(f) \in T_{OP^+,\tilde{p}}(X)$ für alle $f = (t,p)$ oder $f = (p,t)$.

(iv) Eine Σ-*Beschriftung* von N ist eine Tripel $ins = (\varphi, M^\circ, \lambda)$, bestehend aus einer Sortenbelegung φ von N, einer φ-respektierenden Startmarkierung M° und der φ-respektierenden Kantenbeschriftung von N.

(v) Sei *ins* eine Σ-Beschriftung von N. Dann heißt das Paar (ins, E) die *SPEC-Beschriftung* von N und das Tripel (N, ins, E) heißt *SPEC-beschriftetes Netz*. Wenn *ins* und E aus dem Kontext ersichtlich sind, dann bezeichnen wir N einfach nur als beschriftet.

Im folgenden führen wir einige Abkürzungen ein, die es uns leichter machen, über das dynamische Verhalten eines beschrifteten Netzes zu sprechen:

3.4.2 Schreibweise

Sei $N = (P,T,F)$ ein Netz, $\Sigma = (S,OP)$ eine Signatur, X eine Familie von Σ-Variablen und $(\varphi, M^\circ, \lambda)$ die Σ-Beschriftung von N über X.

(i) Für alle $(x,y) \in (T \times P) \cup (P \times T)$ schreiben wir abkürzend

$$\overrightarrow{x,y} = \begin{cases} \lambda(x,y) & \text{falls } (x,y) \in F \\ \tilde{0} & \text{sonst} \end{cases}$$

(ii) Für jede Transition $t \in T$ definieren wir den Vektor $\underline{t} : P \to T\hat{O}_P(X)$ durch

$$\underline{t}(p) = \overrightarrow{t,p} - \overrightarrow{p,t}$$

Damit können wir das dynamische Verhalten eines beschrifteten Netzes wie folgt definieren:

3.4.3 Definition *Markierung, Aktiviertheit, Folgemarkierung*

Sei $SPEC = (S,OP,E)$ eine Spezifikation, $\Sigma = (S,OP)$ ihre Signatur, $N = (P,T,F)$ ein Netz und $(\varphi, M^\circ, \lambda)$ die Σ-Beschriftung von N.

(i) Eine *Markierung* M von N ist eine Abbildung $M : P \to T_{OP}{}^{+}$ mit $M(p) \in T_{OP}{}^{+}, \tilde{p}$ für jedes $p \in P$.

(ii) Sei M eine Markierung und $\beta : X \to T_{OP}$ eine Belegung. Eine Transition $t \in T$ heißt *M-aktiviert unter der Belegung* β g.d.w.

$$\overrightarrow{\beta(p,t)} \leq_{\hat{E}} M(p) \quad \text{für alle } p \in P$$

Wir schreiben dann $M\ \overrightarrow{t,\beta}$.

(iii) Sei M eine Markierung und $t \in T$ M-aktiviert unter einer Belegung $\beta : X \to T_{OP}$. M' heißt *Folgemarkierung* von M g.d.w.

$$M'(p) = M(p) - \overrightarrow{\beta(p,t)} + \overrightarrow{\beta(t,p)} \quad \text{für alle } p \in P.$$

Wir erhalten M' durch das Schalten von t unter der Belegung β und schreiben $M\ \overrightarrow{t,\beta}\ M'$.

(iv) Wir definieren $[M\rangle$ als die kleinste Menge von Markierungen, so daß

(a) $M \in [M\rangle$.
(b) wenn $M' \in [M\rangle$ und $M' \overrightarrow{t,\beta} M''$ dann auch $M'' \in [M\rangle$.
$[M^{\circ}\rangle$ ist die Menge der *erreichbaren Markierungen* von N.

Als abkürzende Schreibweise erhalten wir aus 3.4.2 und 3.2.7:

3.4.4 Korollar

Wenn $M \overrightarrow{t,\beta} M'$ dann gilt $M'(p) \equiv_{\hat{E}} M(p) + \beta(\underline{t}(p))$ für alle Stellen $p \in P$.

Indem wir die Markierungen und die Abbildungen $\underline{t}$ als p-indizierte Vektoren auffassen, und sowohl die Multimengen-Summe als auch die erweiterte Belegung β auf die Komponenten eines Vektors anwenden, erhalten wir in der Vektorschreibweise:

3.4.5 Korollar

Wenn $M \overrightarrow{t,\beta} M'$, dann gilt $M' \equiv_{\hat{E}} M + \beta(\underline{t})$.

3.5 S-Invarianten und verwandte Netzeigenschaften

Von besonderem Interesse für die Analyse von Systemen sind Eigenschaften, die sich während eines Programmablaufs nicht ändern. Eine dieser Klassen von Eigenschaften können aus den sog. S-Invarianten oder auch Stellen-Invarianten hergeleitet werden. Eine S-Invariante ist eine Gewichtungsfunktion W, die jeder Stelle eines Netzes ein Gewicht so zuordnet, daß die Summe aller gewichteten Stelleninhalte für alle erreichbaren Markierungen M konstant bleibt. In unserem Netzkalkül kann diese Gewichtungsfunktion durch einen stellen-indizierten Vektor W von Multimengen-Termen dargestellt werden. Darüberhinaus wird sich herausstellen, daß S-Invarianten durch ein Skalarprodukt von Vektoren über Multimengen-Terme formal definiert und überprüft werden können, wobei als Produkt auf den Komponenten die Substitution von Termen benutzt wird. Das Produkt $M \cdot W$ für eine S-Invariante W und eine Markierung M ergibt daher einen Multimengen-Grundterm. Aus der Tatsache, daß dieses Produkt für alle Markierungen M konstant bleibt, lassen sich Sicherheitseigenschaften ableiten.

Der besondere Vorteil von S-Invarianten ist der, daß sie sich als Lösungen eines homogenen linearen Gleichungssystems beschreiben lassen. Als Grundlage dieses Gleichungssystems dient die sog. Inzidenzmatrix eines beschrifteten Netzes N mit den Stellen P und den Transitionen T: die Inzidenzmatrix ist eine $P \times T$-Matrix $\underline{N}$ mit den Einträgen $\underline{N}(p,t) = \underline{t}(p)$ (vgl. 3.4.2). Legen wir als Produkt über Komponenten die bereits erwähnte Term-Substitution zugrunde, dann sind die S-Invarianten gerade die Lösungen des Gleichungssystems $\underline{N}^{\tau} \cdot i \equiv_{\hat{E}} \underline{\tilde{O}}_s$ (N^{τ} bezeichnet die Transponierte von N). S-Invarianten sind übrigens sortenbehaftet.

Das Produkt zweier Terme basiert, wie bereits erwähnt, auf der Termsubstitution, die wiederum durch eine besondere Art der Variablenbelegung definiert wird: Eine Belegung

ass_u ersetzt in einem Term u' alle Variablen durch den Term u, wenn sie die gleiche Sorte wie u haben, alle anderen Variablen bleiben unverändert. So werden bestimmte Variablen in u' durch den Term u substituiert.

3.5.1 Definition *Konstante u-Belegung*

Sei $\Sigma = (S,OP)$ eine Signatur, X eine Familie von Σ-Variablen, X_s die Σ-Variablen der Sorte $s \in S$ und $u \in T_{OP,s}(X)$. Die *konstante u-Belegung* $ass_u : X \to T_{OP}(X)$ ist definiert durch

$$ass_u(x) = \begin{cases} u & \text{falls } x \in X_s \\ x & \text{sonst} \end{cases}$$

Wir erweitern die konstanten u-Belegungen ass_u auf Multimengen-Terme auf die übliche Art und Weise mit $\overline{ass_u} : T_{\hat{O}P}(X) \to T_{\hat{O}P}(X)$ (vgl. 3.2.7) und können damit ein Produkt auf Multimengen-Terme definieren:

3.5.2 Definition *Produkt von Multimengen-Terme*

Sei $\Sigma = (S,OP)$ eine Signatur, X eine Familie von Σ-Variablen und $u,u' \in T_{\hat{O}P}(X)$. Das *Produkt* $u \cdot u' \in T_{\hat{O}P}(X)$ wird induktiv über den Aufbau von u definiert durch
(i) $MAKE(u) \cdot u' = \overline{ass_u}(u')$ g.d.w. $u \in T_{OP}(X)$,
(ii) $(u_1 + u_2) \cdot u' = u_1 \cdot u' + u_2 \cdot u'$ und
(iii) $(-u) \cdot u' = -(u \cdot u')$.

Für Vektoren von Multimengen-Terme führen wir darüberhinaus das in der linearen Algebra übliche Skalarprodukt ein:

3.5.3 Definition *Skalarprodukt für Vektoren von Multimengen-Terme*

Sei $\Sigma = (S,OP)$ eine Signatur, X eine Familie von Σ-Variablen und P eine endliche Menge. Für Vektoren von Multimengen $\underline{u},\underline{u}' : P \to T_{\hat{O}P}(X)$ ist das Produkt $\underline{u} \cdot \underline{u}' \in T_{\hat{O}P}(X)$ definiert durch

$$\underline{u} \cdot \underline{u}' = \sum_{p \in P} \underline{u}(p) \cdot \underline{u}'(p)$$

Wir können damit den Begriff der S-Invariante formal definieren:

3.5.4 Definition *S-Invariante*

Sei $SPEC = (S,OP,E)$ eine Spezifikation, $\Sigma = (S,OP)$ ihre Signatur, $N = (P,T,F)$ ein über $SPEC$ beschriftetes Netz, X eine Familie von Σ-Variablen und Y eine zu X disjunkte Familie von Σ-Variablen. Ein Vektor $i : P \to T_{\hat{O}P,m_s}(Y)$ ist eine *Stellen-Invariante* von N der Sorte $s \in S$ g.d.w.

$$M \cdot i \equiv\hat{}_E\ M' \cdot i \quad \text{für alle Markierungen } M, M' \text{ von } N \text{ mit } M' \in [M\rangle.$$

Das nun folgende Theorem stellt einen Zusammenhang zwischen S-Invarianten und den Lösungen des Gleichungssystems $\underline{N}^{\tau} \cdot i \equiv\hat{}_E\ \underline{\tilde{O}}_s$ her:

3.5.5 Theorem

Sei $SPEC = (S,OP,E)$ eine Spezifikation, $\Sigma = (S,OP)$ ihre Signatur, $N = (P,T,F)$ ein über $SPEC$ beschriftetes Netz, X eine Familie von Σ-Variablen und Y eine zu X disjunkte Familie von Σ-Variablen. Ein Vektor $i : P \to T\hat{O}_{P,m_s}(Y)$ ist eine Stellen-Invariante der Sorte $s \in S$ g.d.w.

$$\underline{t} \cdot i \equiv\hat{}_E\ \tilde{O}_s \quad \text{für alle } t \in T$$

(Beweis siehe [REIS90a], Theorem 5.6 und 5.8).

Aus 3.5.2 können wir folgern, daß die Summe von S-Invarianten wieder eine S-Invariante ergibt:

3.5.6 Korollar *Additivität von S-Invarianten*

Wenn i_1 und i_2 S-Invarianten eines beschrifteten Netzes N sind, dann sind auch $i_1 + i_2$ und $-i_1$ S-Invarianten von N.

Die Stellen p, die an einer nicht-negativen S-Invariante i "beteiligt" sind (d.h. nicht $i(p) \equiv\hat{}_E\ \tilde{O}$), zeichnen sich durch eine besondere Eigenschaft aus: die Anzahl der Kopien eines Objekts auf diesen Stellen kann nicht unbeschränkt wachsen. Mit Kenntnis der Startmarkierung können wir sogar eine Grenze für diese Anzahl angeben. Die im folgenden definierten Begriffe beschäftigen sich mit dem Zusammenhang zwischen S-Invarianten und der Anzahl der Marken auf den Stellen. Insbesondere interessieren uns Netze, bei denen unter allen erreichbaren Markierungen auf jeder Stelle höchstens eine Kopie eines Objekts vorhanden ist. Wir bezeichnen solche Netze als "1-sicher".

Um formal über 1-Sicherheit sprechen zu können, benötigen wir vorher den Begriff der Mengen-Terme. Es handelt sich dabei um einen Teil der Multimengen-Terme, die so aufgebaut sind, daß sie von jedem möglichen Objekt höchstens ein Element benennen. Sie repräsentieren daher Mengen von Objekten.

3.5.7 Definition *Mengen-Term*

Sei $SPEC = (S,OP,E)$ eine Spezifikation und X eine Familie von (S,OP)-Variablen.

(i) Für jeden Multimengen-Term $u \in T_{NOS}(X)$ (vgl. 3.3.5) definieren wir die Menge $T_{OP}(u)$ der in u enthaltenen (S,OP)-Terme induktiv über den Termaufbau durch
   (a) $T_{OP}(\tilde{O}) = \emptyset$.
   (b) $T_{OP}(u) = \{u\}$ g.d.w. $u \in T_{OP}(X)$.
   (c) $T_{OP}(u_1 + u_2) = T_{OP}(u_1) \cup T_{OP}(u_2)$.

(ii) Ein $u \in T_{NOS}(X)$ heißt *Mengen-Term* g.d.w. über der Struktur von u induktiv gilt
   (a) $u = \tilde{O}$ oder
   (b) u hat die Form $MAKE(u_1)$ und $u_1 \in T_{OP}(X)$ oder
   (c) u hat die Form $u_1 + u_2$, u_1 und u_2 sind Mengen-Terme und für alle $v_i \in T_{OP}(u_i)$ gilt nicht $v_1 \equiv_{\hat{E}} v_2$.

(iii) Ein $u \in T_{\hat{O}P}(X)$ heißt *Mengen-Term*, wenn es einen Mengen-Term $v \in T_{NOS}(X)$ gibt mit $u \equiv_{\hat{E}} v$.

Wir bezeichnen die Menge der Mengen-Terme über X mit $T_{OP}^{1}(X)$.

3.5.8 Definition *1-sicheres Netz*

Sei $SPEC = (S,OP,E)$ eine Spezifikation und $N = (P,T,F)$ ein über $SPEC$ beschriftetes Netz. N heißt *1-sicher* g.d.w. für alle Stellen $p \in P$ und alle erreichbaren Markierungen M von N ist $M(p)$ ein Mengenterm.

Zum Erkennen der 1-Sicherheit fehlt uns jetzt noch der Begriff der “nicht-negativen S-Invarianten”:

3.5.9 Definition *Nicht-negative S-Invariante*

Sei $SPEC$ eine Spezifikation und $N = (P,T,F)$ ein über $SPEC$ beschriftetes Netz. Eine S-Invariante i von N heißt *nicht-negativ*, g.d.w.

$i(p)$ ist nicht-negativ für alle $p \in P$.

Das zentrale Theorem, um 1-sichere Netze zu erkennen, lautet dann:

3.5.10 Theorem

Sei $SPEC = (S,OP,E)$ eine Spezifikation, $N = (P,T,F)$ ein über $SPEC$ beschriftetes Netz und M° die Startmarkierung von N. N ist 1-sicher, falls für alle Stellen $p \in P$ gilt: es gibt eine nicht-negative S-Invariante i von N und einen Mengenterm u mit

$$i(p) >_{\hat{E}} \tilde{O} \quad \text{und} \quad u \cdot i(p) \equiv_{\hat{E}} M^{\circ} \cdot i \quad \text{und } M^{\circ} \cdot i \text{ ist ein Mengen-Term.}$$

(ohne Beweis)

Wir werden uns später bei der Definition einer Logik zur Analyse und Spezifikation des dynamischen Verhaltens ausschließlich um 1-sichere Netze kümmern. Die Stellen von Netzen mit dieser Eigenschaft können nämlich als Prädikate über den Objekten einer bestimmten Sorte aufgefaßt werden. Doch bevor wir das dynamische Verhalten eines modellierten Systems spezifizieren oder analysieren können, müssen wir zuerst über Abläufe reden, die in dieser Arbeit den Kern der Semantik eines Petrinetzes bilden.

3.6 Abläufe und Semantik von beschrifteten Netzen

Ein beschriftetes Petrinetz ist für uns eine formale Vorschrift zur Konstruktion von Abläufen. Unter einem Ablauf verstehen wir eine unendliche Sequenz von Zuständen. Wir gelangen von einem Zustand zum nächsten, indem wir höchstens eine Transition schalten lassen. Wir legen also die Interleaving-Semantik zugrunde (vgl. Abschnitt 3.1.6) und argumentieren über globale Systemzustände. Die Parallelität unabhängiger Ereignisse simulieren wir, indem wir sie beliebig hintereinander ausführen. Die Bedeutung eines Petrinetzes ist daher nicht etwa ein bestimmter Ablauf, sondern alle Abläufe, die wir gemäß dem beschrifteten Netz konstruieren können.

Ein Ablauf ist, wie bereits erwähnt, eine unendliche Sequenz von Zuständen, wobei jeder Zustand durch eine Markierung M_i repräsentiert wird. Für funktionale Systeme, die nach endlicher Zeit stoppen, führen wir den Begriff des Endzustands ein, von dem aus keine weiteren Aktionen mehr möglich sind. Um auch diesen Fall durch eine unendliche Sequenz darstellen zu können, definieren wir, daß alle Folgezustände des Endzustands mit dem Endzustand übereinstimmen müssen. Darüberhinaus fordern wir, daß es keine "persistent aktivierten" Transitionen gibt, d.h. jede Transition, die lange genug (kontinuierlich) aktiviert war, darf nach endlicher Zeit schalten.

3.6.1 Definition *Ablauf*

Sei $SPEC = (S,OP,E)$ eine Spezifikation, $N = (P,T,F)$ ein über $SPEC$ beschriftets Netz, M° die Startmarkierung von N und X eine Familie von (S,OP)-Variablen. Eine Abbildung $A : \mathbb{N} \to [M^\circ\rangle$ heißt *Ablauf von N* g.d.w.

(i) $A(0) = M^\circ$.

(ii) Zu jedem $i \in \mathbb{N}$ gibt es entweder ein $t \in T$ und eine Belegung $\beta : X \to T_{OP}$ mit $A(i)\ \overrightarrow{t,\beta}\ A(i+1)$ oder $A(i)$ ist *Endzustand*, d.h. es gibt kein $t \in T$ und keine Belegung β mit $A(i)\ \overrightarrow{t,\beta}$ und für alle $j > i$ gilt: $A(j) = A(i)$.

(iii) Sei $i \in \mathbb{N}$, $t \in T$ und $\beta : X \to T_{OP}$. Wenn $A(i)\ \overrightarrow{t,\beta}$, dann existiert ein $j > i$ und es gilt nicht $A(j)\ \overrightarrow{t,\beta}$.

Manchmal ist es nötig, noch mehr Lebendigkeit zu fordern, als wir in Definition 3.6.1 Punkt (iii) festgelegt haben. Wir können uns einen Ablauf vorstellen, in der eine Transition unendlich oft aktiviert ist, aber eben nicht kontinuierlich. In diesem Fall kommt unsere bisherige "Fairness-Anforderung" aus 3.6.1 (iii), die wir auch als schwache Fairness bezeichnen, nicht zur Anwendung, da die Transition nicht persistent aktiviert ist. Trotzdem wünschen wir uns, daß eine Transition, die unendlich oft aktiviert ist, auch gelegentlich schaltet. Da wir aber diese Art der Fairness sehr sorgsam und nur bei bestimmten Problemen anwenden wollen, kommt eine globale Fairness-Annahme nicht in Frage.

Wir führen deshalb den Begriff der "lokalen Fairness" ein, mit dessen Hilfe wir die Stellen im Netz, bei denen zusätzliche Fairness gefordert werden muß, genau lokalisie-

ren. Eine Stelle p heißt "fair", wenn alle Transitionen t im Nachbereich von p fair behandelt werden, d.h. in keinem Ablauf darf t nur endlich oft schalten, falls t unendlich oft aktiviert ist:

3.6.2 Definition *Lokale Fairness*

Sei $SPEC = (S,OP,E)$ eine Spezifikation, $N = (P,T,F)$ ein über $SPEC$-beschriftetes Netz und A ein Ablauf von N. Für $p \in P$ heißt p *fair in* A g.d.w. für alle $t \in T$ mit $p \in {}^\bullet t$ und für alle $\beta : X \to T_{OP}$ gilt:
Wenn es für alle $i \in \mathbb{N}$ ein $j \geq i$ gibt mit $A(j)\ \overrightarrow{t,\beta}$, dann existiert für alle i auch ein $k \geq i$ mit $A(k)\ \overrightarrow{t,\beta}\ A(k+1)$.

Damit ergeben sich die weiteren Definitionen: Wir fassen alle Stellen, bei denen wir lokale Fairness fordern, zu einer Menge von "Fairness-Bedingungen" zusammen und definieren, was "faire Abläufe" sind.

3.6.3 Definition *Ablauf mit Fairness*

Sei $SPEC = (S,OP,E)$ eine Spezifikation, $N = (P,T,F)$ ein über $SPEC$ beschriftetes Netz und A ein Ablauf von N.

(i) Eine Menge von *Fairness-Bedingungen* für N ist eine Teilmenge $FAIR \subseteq P$.

(ii) Sei $FAIR$ eine Menge von Fairness-Bedingungen für N. A heißt *Ablauf von N mit Fairness FAIR* g.d.w.

p ist fair in A für alle $p \in FAIR$.

Jetzt können wir auch formalisieren, was die Bedeutung eines beschrifteten Netzes mit Fairness-Bedingungen ist:

3.6.4 Definition *Netz mit Fairness, Semantik*

Sei $SPEC = (S,OP,E)$ eine Spezifikation, $N = (P,T,F)$ ein über $SPEC$ beschriftetes Netz, $ins = (\varphi,M^\circ,\lambda)$ eine (S,OP)-Beschriftung von N und $FAIR$ die Fairness-Bedingungen für N.

(i) Ein Viertupel $\Omega = (N,ins,E,FAIR)$ heißt *SPEC-beschriftetes Netz mit Fairness FAIR*.

(ii) Sei $\Omega = (N,ins,E,FAIR)$ ein $SPEC$-beschriftetes Netz mit Fairness $FAIR$. Dann ist die *Semantik von* Ω definiert durch:

$$SEM(\Omega) \;=\; \{A \mid A \text{ ist Ablauf von } N \text{ mit Fairness } FAIR\}.$$

Die Fairness-Annahmen werden dabei als Nebenbedingungen beim Netz notiert.

4 Systemanalyse und -Spezifikation durch eine temporale Logik

Um die Korrektheit einer Systemmodellierung überprüfen zu können, benötigen wir eine Sprache zur Formulierung von Systemeigenschaften. Jede Aussage dieser Sprache soll am Systemmodell überprüft werden können, d.h. eine Aussage gilt genau dann, wenn das Modell die Eigenschaft hat, die durch die Aussage formuliert wird. Dabei sind vor allem die Eigenschaften von Interesse, die das dynamische Verhalten eines Systemmodells betreffen. Als Systemmodelle werden wir im weiteren ausschließlich beschriftete Netze betrachten, die 1-sicher sind und sich damit wie Prädikat/Ereignis-Netze verhalten (vgl. [REIS86], S. 132). In Abschnitt 4.1 wird eine Spezifikationssprache für beschriftete, 1-sichere Netze vorgestellt. Die Sprache argumentiert über das Verhalten eines dynamischen Systems, indem sie die Menge seiner Abläufe in zwei Teile gliedert. Aus der Sicht einer Systemspezifikation mit dieser Logik gibt es korrekte Abläufe und unerwünschte Abläufe. Ein beschriftetes, 1-sicheres Netz, mit dem man ausschließlich korrekte Abläufe im Sinne dieser Spezifikation konstruieren kann, gilt als korrekte Implementierung. Die Logik basiert u.a. auf den temporalen Operatoren *unless*, *ensures* und *leads-to*. Wir benutzen die Operatoren in Anlehnung an das von K.M. Chandy und J. Misra entwickelte Logikkalkül für UNITY-Programme, das eine weitreichende Ähnlichkeit mit dem in dieser Arbeit verwendeten Kalkül aufweist [CHAN88].

Aussagen über Eigenschaften sind aber erst dann wirklich nützlich, wenn wir aus ihnen neue Aussagen ableiten können, die wiederum Eigenschaften des Systemmodells sind. So können wir aus trivialen, leicht “ablesbaren” Netzeigenschaften Aussagen ableiten, die uns das Verhalten des Netzes auf einer sehr abstrakten und problemnahen Ebene beschreiben. Eine Sammlung von syntaktischen Ableitungsregeln findet sich in 4.2.2, 4.3.1 und 4.4.1.

4.1 Eine Sprache für Eigenschaften von beschrifteten Netzen

Die Formeln unserer Sprache sind Zeichenketten über den Elementen eines Alphabets, das aus den folgenden Teilen besteht: 1) einer Menge von Operationssymbolen (z.B.: $\neg$, “(”, *unless*), 2) allen Σ-Grundtermen der Signatur des beschrifteten Netzes und 3) den Stellen P des Netzes. Daraus folgt, daß jedes Netz seine eigene Sprache besitzt, da jedes Netz andere Stellen hat und über eine andere Signatur beschriftet wird. Wir legen zunächst die Menge der Symbole fest, aus denen die Sprachen aller Netze aufgebaut sind:

4.1.1 Definition *Zeichensatz der Sprachen der Eigenschaften*

Die Menge $\alpha = \{\neg, \vee, \text{“(”}, \text{“)”}, \text{“.”}, \mathit{unless}, \mathit{unless}_\kappa, \mathit{ensures}, \mathit{ensures}_\kappa, \mathit{ensures}^3, \mathit{ensures}^3_\kappa, \textbf{initially}, \mapsto, \mathit{fair}, \mathit{fair}_\kappa\}$ ist der *Zeichensatz der Sprachen der Eigenschaften*. Dabei sind $\mathit{ensures}^3$ und $\mathit{ensures}^3_\kappa$ dreistellige Relationssymbole.

Wenn a,b und c drei Prädikate sind, dann steht a *unless* b für die Eigenschaft “a gilt solange bis b gilt”, die Formel a *ensures* b steht für “a wird in einem Schritt zu b” und **initially** a bedeutet, daß a im Anfangszustand gilt. $a \mapsto b$ lesen wir als “a führt zu b” und ein $a\ \mathit{ensures}^b\ c$ bedeutet etwa “a wird in einem Schritt zu c, wenn b fair ist, d.h. b *fair*” . Dabei bedeutet ein Index κ, daß der Operator kompositional ist (davon später mehr).

Ein Teil unserer Sprache besteht aus Ausdrücken, die aussagenlogisch interpretiert werden können und deren Gültigkeit später anhand einzelner Markierungen bestimmt wird. Als Aussagevariablen benutzen wir Konstrukte der Form $u.p$, wobei u ein beliebiger Σ-Grundterm sein kann und p eine Stelle im Netz N ist (wir nehmen an, daß N über Σ beschriftet ist). Weitere Ausdrücke sind z.B. $(u.p \vee v.q)$ und $\neg u.p$.

4.1.2 Definition *Ausdrücke der Sprache der Eigenschaften*

Sei $\Omega = (N,ins,E,FAIR)$ ein *SPEC*-beschriftetes 1-sicheres Netz mit Fairness, $N = (P,T,F)$, $\Sigma = (S,OP)$ die Signatur von *SPEC* und $ins = (\varphi,M^\circ,\lambda)$ die Σ-Beschriftung von N. Seien darüberhinaus OP,P und α paarweise disjunkt. Die Menge der Ausdrükke $EXPR_\Omega$ der Sprache der Eigenschaften von Ω ist die kleinste Menge, für die gilt:

(i) $u.p \in EXPR_\Omega$ für alle $p \in P$ und $u \in T_{OP,\varphi(p)}$.

(ii) $(a \vee b) \in EXPR_\Omega$, wenn $a,b \in EXPR_\Omega$.

(iii) $\neg a \in EXPR_\Omega$, wenn $a \in EXPR_\Omega$.

Aus den Ausdrücken können kompliziertere Formeln zusammengebaut werden, deren Gültigkeit später anhand der Semantik eines Netzes (d.h. der Menge alle möglichen Abläufe) festgelegt wird. Dabei ist jeder Ausdruck bereits eine Formel und gehört daher zur Sprache der Eigenschaften.

4.1.3 Definition *Syntax der Sprache der Eigenschaften*

Sei Ω wie in der Annahme von 4.1.2. Die Sprache L_Ω der Eigenschaften von Ω ist die kleinste Menge, für die gilt:

(i) $a \in L_\Omega$, wenn $a \in EXPR_\Omega$.

(ii) $a\ \mathit{unless}_{(\kappa)}\ b \in L_\Omega$, wenn $a,b \in EXPR_\Omega$.

(iii) $a\ \mathit{ensures}_{(\kappa)}\ b$, $a\ \mathit{ensures}^r_{(\kappa)}\ b \in L_\Omega$, wenn $a,r,b \in EXPR_\Omega$.

(iv) $a \mapsto b$, $a\ \mathit{fair}_{(\kappa)}$, **initially** $a \in L_\Omega$, wenn $a,b \in EXPR_\Omega$.

Der Index (κ) steht dabei wahlweise für keinen Index oder den Index κ.

Um die Lesbarkeit unserer Formeln zu verbessern, führen wir einige Konventionen ein:

4.1.4 Schreibweise

Wir schreiben

(i) $(a \wedge b)$ statt $\neg(\neg a \vee \neg b)$.

(ii) $(a \Rightarrow b)$ statt $(\neg a \vee b)$.

(iii) $(a \Leftrightarrow b)$ statt $((a \Rightarrow b) \wedge (b \Rightarrow a))$.

(iv) $(a_1 \ or \ a_2 \ \ldots \ or \ a_n)$ und meinen, daß genau ein a_i gilt, für $i \in \{1,\ldots,n\}$.

(v) Für die Operatoren $\neg$, $\vee$, $\wedge$, $\Rightarrow$, $\Leftrightarrow$ gelten die üblichen Vorrangregeln. Wir schreiben Klammern nur dann, wenn die beabsichtigte Klammerung nicht offensichtlich ist.

Die Semantik eines Ausdrucks a wird durch eine Modellbeziehung[1] zwischen a und den Markierungen des Σ-beschrifteten Netzes $N = (P,T,F)$ definiert. Eine Markierung M von N heißt Modell von a, wenn a eine wahre Aussage über M macht. Wir sagen auch "a ist gültig in M". Den Wahrheitsbegriff stützen wir letztendlich auf den Ausdrücken der Form $u.p$ ab, wobei u ein Σ-Grundterm ist und $p \in P$. Dabei ist $u.p$ genau dann gültig in M, wenn der Term u in dem Mengen-Term $M(p)$ enthalten ist, d.h. das "Objekt" u ist in der Extension $M(p)$ des "Prädikats" p enthalten. Da wir uns nur auf 1-sichere Netze beziehen, kann ein Objekt genau einmal oder keinmal in $M(p)$ enthalten sein, was unserem intuitiven Verständnis eines Prädikats entspricht.

4.1.5 Definition *Semantik der Ausdrücke von L_Ω*

Sei $\Omega = (N,ins,E,FAIR)$ ein *SPEC*-beschriftetes 1-sicheres Netz mit $N = (P,T,F)$. Die Modellbeziehung zwischen einer Markierung M von N und einem Ausdruck aus $EXPR_\Omega$ ist definiert durch:

(i) $M \models u.p$ g.d.w. $MAKE(u) \leq_{\hat{E}} M(p)$.

(ii) $M \models \neg a$ g.d.w. nicht $M \models a$.

(iii) $M \models (a \vee b)$ g.d.w. $M \models a$ oder $M \models b$.

Um die Semantik der anderen Formeln in L_Ω einfach beschreiben zu können, führen wir eine Notation ein, die über alle Markierungen bzw. alle erreichbaren Markierungen von N argumentiert. Eine Klausel

$$\{a\} \ t \ \{b\}$$

drückt aus, daß in allen erreichbaren Markierungen, in denen der Ausdruck a gilt, entweder die Transition t nicht aktiviert ist, oder das Schalten von t zu einer Folgemarkierung führt, in der b gilt. Die Klausel

$$\{a\}_K \ t \ \{b\}$$

---

1) vgl. [Ebbi86], S. 42

formuliert den gleichen Sachverhalt bezogen auf alle Markierungen von N, d.h. auch die nicht erreichbaren Markierungen. Wenn wir ausdrücken wollen, daß t auch tatsächlich aktiviert sein muß (unter einer Belegung β), wenn a gilt, dann benutzen wir die Klauseln

$$\{a\}_{(\kappa)}\ \overrightarrow{t,\beta}\ \{b\}$$

Wir fassen also zusammen:

4.1.6 Schreibweise

Sei $\Omega = (N,ins,E,FAIR)$ ein *SPEC*-beschriftetes, 1-sicheres Netz mit Fairness, $N = (P,T,F)$, $\Sigma = (S,OP)$ die Signatur von *SPEC*, $ins = (\varphi,M^\circ,\lambda)$ die Σ-Beschriftung von N und X eine Familie von Σ-Variablen. Seien $a,b \in EXPR_\Omega$ und $t \in T$.

(i) Zur Formulierung von Sicherheitsannahmen schreiben wir
  (a) $\{a\}\ t\ \{b\}$, wenn für alle erreichbaren Markierungen $M \in [M^\circ\rangle$ von N gilt: wenn $M \models a$ und $M\ \overrightarrow{t,\beta}\ M'$ für irgendein β, dann $M' \models b$.
  (b) $\{a\}_\kappa\ t\ \{b\}$, wenn für alle Markierungen M von N gilt: wenn $M \models a$ und $M\ \overrightarrow{t,\beta}\ M'$ für irgendein β, dann $M' \models b$ (also auch für nicht erreichbare Markierungen M).

(ii) Sei $\beta : X \to T_{OP}$. Zur Formulierung von Lebendigkeitsannahmen schreiben wir
  (a) $\{a\}\ \overrightarrow{t,\beta}\ \{b\}$, wenn für alle erreichbaren Markierungen $M \in [M^\circ\rangle$ von N gilt: wenn $M \models a$, dann $M\ \overrightarrow{t,\beta}\ M'$ und $M' \models b$.
  (b) $\{a\}_\kappa\ \overrightarrow{t,\beta}\ \{b\}$, wenn für alle Markierungen M von N gilt: wenn $M \models a$, dann $M\ \overrightarrow{t,\beta}\ M'$ und $M' \models b$.

Jetzt können wir die Semantik unserer Sprache L_Ω angeben. Auch hier definieren wir die Gültigkeit durch eine Modellbeziehung. Wenn Ω ein beschriftetes Netz mit Fairness ist, dann sagen wir, daß Ω die Eigenschaft $f \in L_\Omega$ hat, wenn Ω ein Modell für f ist, d.h. f ist gültig in Ω, oder kurz $\Omega \models f$. Der Gültigkeitsbegriff wird diesmal auf den Abläufen von Ω bzw. auf alle Markierungen von Ω gestützt.

Einige der temporalen Operatoren besitzen zwei verschiedene Semantiken: durch einen Index κ legen wir fest, daß der Operator auf allen Markierungen gilt, während der gleiche Operator ohne κ nur über die erreichbaren Markierungen argumentiert. Der Unterschied der beiden Semantiken ist im allgemeinen unwesentlich. Nur wenn wir Systemmodelle durch Vereinigung komponieren wollen, müssen wir den Unterschied beachten: die Operatoren mit Index κ (*unless*$_\kappa$, *ensures*$_\kappa$ und *fair*$_\kappa$) sind kompositional, d.h. sie gelten u.U. nach der Komposition auch im zusammengesetzten System, während alle anderen Operatoren diese Eigenschaft nicht haben. Aus einem kompositionalen Operator kann der Index κ gestrichen werden, ohne die Gültigkeit der Formel zu verletzen; der umgekehrte Weg ist i.a. nicht möglich. Über die Komposition von beschrifteten Netzen und deren Eigenschaften bzgl. der Teilsysteme wird ab 5.1.1 berichtet.

4.1.7 Definition *Semantik von* L_Ω

Sei $\Omega = (N,ins,E,FAIR)$ ein *SPEC*-beschriftetes, 1-sicheres Netz mit Fairness, $N = (P,T,F)$, $\Sigma = (S,OP)$ die Signatur von *SPEC*, $ins = (\varphi,M^\circ,\lambda)$ die Σ-Beschriftung von N und X eine Familie von Σ-Variablen. Die Modellbeziehung zwischen Ω und einer Formel aus L_Ω ist definiert durch:

(i) $\Omega \models a$ g.d.w. $a \in EXPR_\Omega$ und für *alle* Markierungen M von N gilt $M \models a$ (a ist also eine aussagenlogische Tautologie).

(ii) $\Omega \models a \; unless_{(\kappa)} \; b$ g.d.w. für alle $t \in T$ gilt: $\{a \wedge \neg b\}_{(\kappa)} \; t \; \{a \vee b\}$.

(iii) $\Omega \models a \; ensures_{(\kappa)} \; b$ g.d.w. $\Omega \models a \; unless_{(\kappa)} \; b$ und es gibt ein $t \in T$ und ein $\beta : X \to T_{OP}$ mit $\{a \wedge \neg b\}_{(\kappa)} \; \overrightarrow{t,\beta} \; \{b\}$.

(iv) $\Omega \models a \; ensures^r_{(\kappa)} \; b$ g.d.w. $\Omega \models a \; unless_{(\kappa)} \; b$ und es gibt ein $t \in T$ und ein $\beta : X \to T_{OP}$ mit $\{a \wedge r \wedge \neg b\}_{(\kappa)} \; \overrightarrow{t,\beta} \; \{b\}$.

(v) $\Omega \models$ **initially** a g.d.w. $M^\circ \models a$.

(vi) $\Omega \models a \mapsto b$ g.d.w. für alle Abläufe $A \in SEM(\Omega)$ und alle $i \in \mathbb{N}$ mit $A(i) \models a$ existiert ein $j \geq i$ mit $A(j) \models b$.

(vii) $\Omega \models r \; fair_{(\kappa)}$ g.d.w. falls es ein $t \in T$ und ein $\beta : X \to T_{OP}$ und Prädikate $a,b \in EXPR_\Omega$ gibt, mit $\{a \wedge r \wedge \neg b\}_{(\kappa)} \; \overrightarrow{t,\beta} \; \{b\}$ und $\Omega \models a \; unless_{(\kappa)} \; b$, dann gilt für alle Abläufe $A \in SEM(\Omega)$:

wenn es für alle $i \in \mathbb{N}$ ein $j \geq i$ gibt mit $A(j) \models (a \wedge r \wedge \neg b)$, dann gibt es zu jedem i auch ein $k \geq i$ mit $A(k) \models b$.

Damit ist die Syntax und die Semantik unserer Sprache formal definiert. Um mit L_Ω praktisch arbeiten zu können, führen wir noch ein Konzept ein, das die Verwendung von Quantoren begründet:

4.1.8 Definition Ω-*Variablen*

Sei $\Omega = (N,ins,E,FAIR)$ ein *SPEC*-beschriftetes Netz mit Fairness, $N = (P,T,F)$, $\Sigma = (S,OP)$ die Signatur von *SPEC* und α der Zeichensatz von L_Ω. $X = (X_s)_{s \in S}$ heißt *Familie von* Ω-*Variablen*, falls X_s paarweise disjunkt zu OP, P und α ist für alle $s \in S$.

4.1.9 Schreibweise

Sei $\Omega = (N,ins,E,FAIR)$ ein *SPEC*-beschriftetes, 1-sicheres Netz, $\Sigma = (S,OP)$ die Signatur von *SPEC*, $a \in L_\Omega$, X eine Familie von Ω-Variablen und $x \in X_s$ für ein $s \in S$.

(i) a^u_x bezeichnet die Formel, in der alle Vorkommen von x durch u ersetzt wurden.

(ii) Wir schreiben $\langle \forall \; x \; :: \; a \rangle$ und meinen: für alle Grundterme $u \in T_{OP}$ der Signatur Σ gilt $\Omega \models a^u_x$.

(iii) Wir schreiben $\langle \exists \; x \; :: \; a \rangle$ und meinen: es gibt einen Grundterm $u \in T_{OP}$ der Signatur Σ mit $\Omega \models a^u_x$.

(iv) Mehrfache Quantifizierungen lösen wir von innen nach außen auf, und wir fassen sie textuell zusammen:

$\langle \forall\, x \;::\; \langle \forall\, y \;::\; a \rangle \rangle$ wird zu $\langle \forall\, x,y \;::\; a \rangle$

(v) Wenn Ω aus dem Kontext ersichtlich ist, dann schreiben wir a anstelle von $\Omega \models a$.

(vi) Nicht quantifizierte (freie) Variablen sind als all-quantifiziert zu betrachten, da wir grundsätzlich den universellen Abschluß einer Formel annehmen:
$x.p \;\; unless \;\; x.q$ steht für $\langle \forall\, x \;::\; x.p \;\; unless \;\; x.q \rangle$

L_Ω und die Prädikatenlogik der 1. Stufe

Sowohl die Syntax als auch die Semantik unserer Sprache L_Ω beziehen sich auf ein konkretes Netz Ω. Oft ist es jedoch notwendig, auch darüberhinaus logisch argumentieren zu können, d.h. insbesondere unabhängig von bestimmten Netzen. Wir werden deshalb (wann immer es nötig ist) neben L_Ω auch noch allgemeine Formulierungen aus der Prädikatenlogik der 1. Stufe benutzen. Dieser Teil unserer Argumentationssprache wird nicht formal definiert, sondern als grundlegendes Werkzeug der mathematischen Beweisführung betrachtet. Dabei gilt: entweder können die Formulierungen, die Aussagen in L_Ω mit denen aus der Prädikatenlogik kombinieren, auf L_Ω alleine abgestützt werden (die Prädikatenlogik wird also nur benutzt, um Formeln kurz zu halten), oder die Bedeutung ist intuitiv klar. So soll z.B. die Formel

$$x.p \;\; unless \;\; false$$

ausdrücken, daß aus der Stelle p keine Marken abgezogen werden können, d.h. falls $x.p$ jemals gilt, dann gilt es solange, bis "*false*" gilt, also für immer. "false" steht dann in L_Ω für einen beliebigen unerfüllbaren Ausdruck.

In den folgenden Abschnitten geben wir eine informelle Beschreibung der einzelnen Operatoren und zeigen ihren Zusammenhang mit Systemeigenschaften auf.

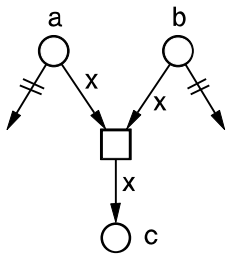
4.2 Unless

Im folgenden betrachten wir ein beschriftetes, 1-sicheres Netz $\Omega = (N,ins,E,FAIR)$. Zwischen den Operatoren mit Index κ und denen ohne Index unterscheiden wir nur dann, wenn wir über Eigenheiten sprechen, die der jeweils andere Operator nicht besitzt. Weisen wir nicht gesondert darauf hin, so gelten alle Angaben analog auch für Operatoren mit Index κ. Anstelle von Ausdrücken aus L_Ω sprechen wir allgemein über Prädikate a, b.

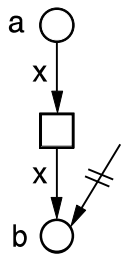
Durch die Relation *unless* können Sicherheitseigenschaften ausgedrückt werden. Die Formel $a \;\; unless \;\; b$ sprechen wir als "a gilt solange, bis b gilt", und ihre Bedeutung ist:

$$a \;\; unless \;\; b \quad \text{g.d.w.} \quad \forall\, t \in T \;\; \text{gilt} \;\; \{a \wedge \neg b\} \;\; t \;\; \{a \vee b\}$$

Mit anderen Worten: für alle erreichbare Markierungen M, in denen $(a \wedge \neg b)$ gültig ist und für alle Transitionen, die M in eine Folgemarkierung M' überführen, muß $(a \vee b)$ in



x.a $\wedge$ *x.b* *unless* *x.c*
x.a *unless* *x.a* $\wedge$ *x.b*
x.b *unless* *x.b* $\wedge$ *x.a*
x.a *unless* $\neg$*x.a* $\wedge$ *x.c*
x.b *unless* $\neg$*x.b* $\wedge$ *x.c*
x.a $\wedge$ $\neg$*x.b* *unless* *x.a* $\wedge$ *x.b*



$\neg$*x.b* *unless* $\neg$*x.b* $\wedge$ *x.a*

Abb. 4.1. Ablesen von Unless-Eigenschaften

der Folgemarkierung M' gültig sein. Falls a also in irgendeiner erreichbaren Markierung gelten sollte, dann

- gilt entweder b niemals und a gilt für immer (d.h. in allen Folgemarkierungen)
- oder b gilt irgendwann und a gilt mindestens solange bis b gilt (b kann dabei bereits von Anfang an mit a gelten)

Unless-Eigenschaften werden entweder direkt am Netz abgelesen oder aus bereits abgelesenen Eigenschaften hergeleitet. Abbildung 4.1 zeigt zwei Netzauschnitte zusammen mit einigen Eigenschaften, die daraus abgelesen werden können. Durchgestrichene Pfeile sollen dabei andeuten, daß in der Nachbarschaft keine anderen Pfeile existieren dürfen.

Spezialformen von Unless

Mit *unless* lassen sich auch zwei besonders anschauliche Klassen von Programmeigenschaften ausdrücken, nämlich Stabilität und Invarianz. Um Eigenschaften dieser beiden Klassen hervorzuheben, geben wir ihnen eigene Namen:

4.2.1 Schreibweise

Wir schreiben
(i) **stable** *a* statt *a unless false* .
(ii) **invariant** *a* statt (**initially** *a* und **stable** *a*) .
(iii) **constant** *a* statt (**stable** *a* und **stable** $\neg a$) .

Ein stabiles Prädikat *a* bleibt in allen Folgemarkierungen "wahr", wenn es in einer Markierung wahr wird. Natürlich kann es auch sein, daß *a* niemals wahr wird. Eine Invariante ist in allen Markierungen von allen Abläufen wahr. Beide Klassen von Eigenschaften werden in der hierarchischen Spezifikation des parallelen Garbage Collectors häufig benutzt. Durch **constant** werden Prädikate ausgezeichnet, die sich nicht ändern und von denen wir nicht wissen, ob sie wahr oder falsch sind. Diese Eigenschaft werden wir nur bei der Programmkomposition benutzen, um auszudrücken, daß das eine Teilsystem bestimmte Prädikate des anderen Teilsystems unverändert läßt.

S-Invarianten und Invarianten in $L\Omega$

Zwischen den S-Invarianten von Ω und dem Operator **invariant** von $L\Omega$ gibt es einen sehr nützlichen Zusammenhang: da wir wissen, daß die Menge der Objekte auf den Stellen, die an einer S-Invariante beteiligt sind, konstant ist und nur durch die Startmarkierung festgelegt wird, können wir Prädikate herleiten, die während eines ganzen Ablaufs stabil bleiben. Zusammen mit der Startmarkierung von Ω können wir invariante Eigenschaften ableiten.

Ableitungsregeln für Unless

Um aus abgelesenen Eigenschaften neue Eigenschaften zu gewinnen, die nicht direkt ablesbar sind, benutzen wir Regeln der Form

$$\frac{a_1,\ldots,a_n}{b}$$

wodurch ausgedrückt wird, daß die Eigenschaft *b* gilt, wenn die Eigenschaften a_1, ..., a_n gelten. Wir nennen a_1, ..., a_n die *Prämisse* der Regel und *b* ist die *Behauptung*. Bei Regeln ohne Prämisse (d.h. $n = 0$), notieren wir einfach nur die Behauptung *b* ohne den Ableitungsstrich. Die nachfolgend aufgeführten Ableitungsregeln mit *unless* werden in der hierarchischen Spezifikation des parallelen Garbage Collectors oft verwendet:

4.2.2 Theorem[2] *Ableitungsregeln mit unless*

(i) Reflexivität und Antireflexivität:

$$p \;\; unless \;\; p,$$
$$p \;\; unless \;\; \neg p$$

(ii) Abschwächung der Konsequenz:

$$\frac{p \;\; unless \;\; q \;\; , \;\; q \;\Rightarrow\; r}{p \;\; unless \;\; r}$$

2) vgl. [CHAN88], S.58

(iii) Konjunktion und Disjunktion:

$$\frac{\begin{array}{c} p \;\; \mathit{unless} \;\; q, \\ p' \;\; \mathit{unless} \;\; q' \end{array}}{\begin{array}{ll} (p \wedge p') \;\; \mathit{unless} \;\; (p \wedge q') \vee (p' \wedge q) \vee (q \wedge q'), & \{\text{Konjunktion}\} \\ (p \vee p') \;\; \mathit{unless} \;\; (\neg p \wedge q') \vee (\neg p' \wedge q) \vee (q \wedge q') & \{\text{Disjunktion}\} \end{array}}$$

(iv) Einfache Konjunktion und einfache Disjunktion:

$$\frac{\begin{array}{c} p \;\; \mathit{unless} \;\; q, \\ p' \;\; \mathit{unless} \;\; q' \end{array}}{\begin{array}{ll} p \wedge p' \;\; \mathit{unless} \;\; q \vee q', & \{\text{einfache Konjunktion}\} \\ p \vee p' \;\; \mathit{unless} \;\; q \vee q' & \{\text{einfache Disjunktion}\} \end{array}}$$

(v) Löschungstheorem:

$$\frac{\begin{array}{c} p \;\; \mathit{unless} \;\; q, \\ q \;\; \mathit{unless} \;\; r \end{array}}{p \vee q \;\; \mathit{unless} \;\; r}$$

(vi) Implikationstheorem:

$$\frac{p \Rightarrow q}{p \;\; \mathit{unless} \;\; q}$$

Beweis:

(i) Reflexivität und Antireflexivität:

Für alle Transitionen t und alle Prädikate p gilt
(a) $\{\mathit{false}\} \;\; t \;\; \{p\}$.
(b) $\{p\} \;\; t \;\; \{\mathit{true}\}$.
Die Behauptungen folgen jeweils durch Anwendung der Definition von *unless*. □

(ii) Abschwächung der Konsequenz:

$\{p \wedge \neg q\} \;\; t \;\; \{p \vee q\}$
für alle t, aus der Prämisse p *unless* q
$\{p \wedge \neg q\} \;\; t \;\; \{p \vee r\}$
Abschwächung der Nachbedingung durch die Prämisse $q \Rightarrow r$
$\{p \wedge \neg r\} \;\; t \;\; \{p \vee r\}$
Verstärkung der Vorbedingung durch die Prämisse $\neg r \Rightarrow \neg q$ □

(iii) Konjunktion und Disjunktion:

Aus der Prämisse der Regel folgt, daß für alle Transitionen t gilt
$\{p \wedge \neg q\} \;\; t \;\; \{p \vee q\}$,
$\{p' \wedge \neg q'\} \;\; t \;\; \{p' \vee q'\}$.

Durch Konjunktion bzw. Disjunktion der Vorbedingungen und der Nachbedingungen erhalten wir

$$\{p \wedge \neg q \wedge p' \wedge \neg q'\} \;\; t \;\; \{(p \vee q) \wedge (p' \vee q')\}, \tag{1}$$

$$\{(p \wedge \neg q) \vee (p' \wedge \neg q')\} \;\; t \;\; \{p \vee q \vee p' \vee q'\}. \tag{2}$$

Die Behauptung der Konjunktionsregel folgt aus (1), da die Vorbedingung der Behauptung die Vorbedingung von (1) impliziert und die Nachbedingung von (1) impliziert die Nachbedingung der Behauptung. Analoges gilt für die Behauptung der Disjunktionsregel und (2). □

(iv) Einfache Konjunktion bzw. Disjunktion:

Folgt aus der Regel für Konjunktion bzw. Disjunktion durch geeignete Abschwächung der Konsequenz. □

(v) Löschungstheorem:

p *unless* q
, Prämisse
q *unless* r
, Prämisse
$p \vee q$ *unless* $(\neg p \wedge r) \vee (q \wedge r)$
, Disjunktion der beiden letzten Formeln
$p \vee q$ *unless* r
, Abschwächung der Konsequenz □

(vi) Implikationstheorem:

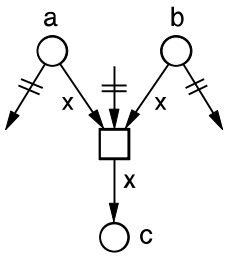
p *unless* p
, Reflexivität von *unless*
p *unless* q
, Abschwächung der Konsequenz durch die Prämisse $p \Rightarrow q$ □

4.3 Ensures

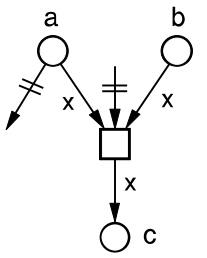
Der *ensures*-Operator beschreibt die grundlegendste Art, wie über die Lebendigkeit in einem beschrifteten, 1-sicheren Netz gesprochen werden kann. Die Formel *a ensures b* sprechen wir als "*a* wird in einem Schritt zu *b*", und ihre Bedeutung ist:

a *ensures* b g.d.w. a *unless* b und es gibt ein t und ein β mit $\{a \wedge \neg b\} \;\; \overrightarrow{t,\beta} \;\; \{b\}$

Anders ausgedrückt: wenn *a* irgendwann gilt, dann bleibt *a* solange gültig, bis *b* gilt (durch *a unless b*), und nach endlicher Zeit wird *b* gelten. Daß *b* irgendwann gelten wird, kann folgendermaßen begründet werden: In allen (erreichbaren) Markierungen, in denen $a \wedge \neg b$ gilt, ist die Transition *t* unter der Belegung β aktiviert. Durch den Unless-Anteil wird sichergestellt, daß *t* weiterhin unter β aktiviert bleibt. Da es jedoch keine persistent



$x.a \wedge x.b \quad ensures \quad x.c$



$x.a \quad ensures^{x.b} \quad x.c$

Abb. 4.2. Ablesen von Ensures-Eigenschaften

aktivierten Transitionen gibt, wird entweder t unter β irgendwann schalten (dann gilt b), oder b wird auf einem anderen Weg gültig gemacht.

Formeln mit *ensures* werden nur dazu benutzt, um Lebendigkeitseigenschaften aus einem Netz abzulesen. Sobald wir Lebendigkeit auf einer abstrakteren Ebene formulieren wollen, benutzen wir den Leads-to Operator ($\mapsto$). Abbildung 4.2 zeigt wieder zwei Netzausschnitte und die daraus ablesbaren Eigenschaften. Die dreistellige Form von *ensures* ($x.a \; ensures^{x.b} \; x.c$) wird nur im Zusammenhang mit lokalen Fairness-Eigenschaften benötigt (vgl. Fairness-Theorem in 4.5.1). Hier einige Ableitungsregeln für *ensures*:

4.3.1 Theorem[3] *Ableitungsregeln mit ensures*

(i) Abschwächung der Konsequenz:

$$\frac{p \quad ensures \quad q \;,\; q \Rightarrow r}{p \quad ensures \quad r}$$

(ii) Konjunktion:

$$\frac{\begin{array}{c} p \quad unless \quad q \;, \\ p' \quad ensures \quad q' \end{array}}{p \wedge p' \quad ensures \quad (p \wedge q') \vee (p' \wedge q) \vee (q \wedge q')}$$

(iii) Konjunktion der negierten Nachbedingung auf die Vorbedingung:

$$\frac{p \quad ensures \quad q \vee r}{p \wedge \neg q \quad ensures \quad q \vee r}$$

3) vgl. [CHAN88] S.62

Beweis:

(i) Abschwächung der Konsequenz

Wie bereits gezeigt, gilt die Regel für *unless*. Deshalb reicht es aus zu zeigen, daß der *ensures*-Anteil gilt:

$\{p \wedge \neg q\} \quad \overrightarrow{t,\beta} \quad \{q\}$

für ein t, β (aus der Prämisse).

$\{p \wedge \neg q\} \quad \overrightarrow{t,\beta} \quad \{r\}$

Abschwächung der Nachbedingung durch die Prämisse $q \Rightarrow r$.

$\{p \wedge \neg r\} \quad \overrightarrow{t,\beta} \quad \{r\}$

Verstärkung der Vorbedingung durch die Prämisse $\neg r \Rightarrow \neg q$. □

(ii) Konjunktion:

Analog zum Beweis der Konjunktion von *unless*. □

(iii) Konjunktion der negierten Nachbedingung auf die Vorbedingung:

$p \quad unless \quad q \vee r$

, *unless*-Anteil aus der Prämisse

$\neg q \quad unless \quad q$

, Antireflexivität von *unless*

$p \wedge \neg q \quad unless \quad q \vee r$

, einfache Konjunktion der beiden letzten Formeln

$p \quad ensures \quad q \vee r$

, Prämisse

$p \wedge \neg q \quad ensures \quad (p \wedge \neg q \wedge r) \vee (p \wedge (q \vee r)) \vee (q \vee r)$

, Konjunktion der beiden letzen Formeln

$p \wedge \neg q \quad ensures \quad q \vee r$

, Abschwächung der Konsequenz □

4.4 Leads-to

Die meisten Lebendigkeitseigenschaften werden durch Formeln mit $\mapsto$ augedrückt. Der Leads-to Operator ist reflexiv, transitiv und nicht kompositional, d.h. Leads-to Eigenschaften von Teilsystemen gelten i.a. nicht im Gesamtsystem, insbesondere gibt es keine Operatorversion mit Index κ. Eine Formel $a \mapsto b$ sprechen wir als "a führt zu b" und sie bedeutet: für alle Abläufe A eines Netzes und für alle Markierungen $A(i)$ gilt: wenn a von $A(i)$ erfüllt wird, dann existiert ein $j \geq i$ und $A(j)$ erfüllt b. Durch die Abstützung von $\mapsto$ auf den Abläufen eines beschrifteten Netzes können auch solche Lebendigkeitseigenschaften formuliert werden, die sich nur in Situationen mit lokaler Fairness ergeben. Diese Eigenschaften können nicht durch *ensures* ausgedrückt werden (siehe Fairness-Theorem 4.5.1).

4.4.1 Theorem[4] *Ableitungsregeln mit Leads-to*

(i) Implikationstheorem:

$$\frac{p \;\Rightarrow\; q}{p \;\mapsto\; q}$$

(ii) Disjunktion:

$$\frac{p \;\mapsto\; q \;,\; p' \;\mapsto\; q'}{p \;\vee\; p' \;\mapsto\; q \;\vee\; q'}$$

(iii) Generierung:

$$\frac{p \quad \textit{ensures} \quad q}{p \;\mapsto\; q}$$

(iv) Reflexivität:

$$p \;\mapsto\; p$$

(v) Transitivität:

$$\frac{p \;\mapsto\; q \,,\, q \;\mapsto\; r}{p \;\mapsto\; r}$$

(vi) Abschwächung der Konsequenz:

$$\frac{p \;\mapsto\; q \;,\; q \;\Rightarrow\; r}{p \;\mapsto\; r}$$

(vii) Löschungstheorem:

$$\frac{p \;\mapsto\; q \;\vee\; b \;,\; b \;\mapsto\; r}{p \;\mapsto\; q \;\vee\; r}$$

(viii) PSP-Theorem (Progress-Savety-Progress):

$$\frac{p \;\mapsto\; q \;,\; r \quad \textit{unless} \quad b}{p \;\wedge\; r \;\mapsto\; (q \;\wedge\; r) \;\vee\; b}$$

(ix) Vervollständigungstheorem:

für jede endliche Menge von Prädikaten $p_i, q_i,\ 0 \le i < N$:

$$\frac{\langle \forall\, i \;::\; p_i \;\mapsto\; q_i \;\vee\; r\rangle,\quad \langle \forall\, i \;::\; q_i \quad \textit{unless} \quad r\rangle}{\langle \forall\, i \;::\; p_i\rangle \;\mapsto\; \langle \forall\, i \;::\; q_i\rangle \;\vee\; r}$$

---

4) vgl. [CHAN88], S. 64

Beweis:

(i) Implikationstheorem:

Wegen $p \Rightarrow q$, gilt für alle Abläufe A und alle $i \in \mathbb{N}$: wenn $A(i) \models p$, dann $A(i) \models q$. Also $p \mapsto q$.

(ii) Disjunktion:

Wegen $p \mapsto q$ gilt für alle Abläufe A und $i \in \mathbb{N}$: wenn $A(i) \models p$, dann existiert ein $j \geq i$ mit $A(j) \models q$.
Wegen $p' \mapsto q'$ gilt für alle Abläufe A und $i \in \mathbb{N}$: wenn $A(i) \models p'$, dann existiert ein $k \geq i$ mit $A(j) \models q'$.
Also gilt für alle Abläufe A und $i \in \mathbb{N}$: wenn $A(i) \models (p \vee p')$, dann existiert entweder ein $j \geq i$ mit $A(j) \models q$ oder ein $k \geq i$ mit $A(k) \models q'$.
Daher $p \vee p' \mapsto q \vee q'$. □

(iii) Generierung:

Aus der Prämisse p *ensures* q folgt

$$\{p \wedge \neg q\} \quad \overrightarrow{t,\beta} \quad \{q\}$$

für ein t, β. Daraus folgt, daß für jeden Ablauf A und $i \in \mathbb{N}$ gilt: wenn $A(i) \models (p \wedge \neg q)$, dann gibt es $j > i$ und es gilt nicht $A(j) \models (p \wedge \neg q)$ nach Definition 3.6.1 (iii). Wegen p *unless* q muß daher gelten $A(j) \models q$.
Also gibt es für jeden Ablauf A und jedes $i \in \mathbb{N}$ mit $A(i) \models (p \wedge \neg q)$ ein $j > i$ mit $A(j) \models q$. Damit gilt $p \wedge \neg q \mapsto q$.
Zusammen mit $p \wedge q \mapsto q$ aus dem Implikationstheorem ergibt sich durch Disjunktion der beiden Formeln: $p \mapsto q$. □

(iv) Reflexivität:

$p \Rightarrow p$
, trivial
$p \mapsto p$
, aus der letzten Formel und dem Implikationstheorem □

(v) Transitivität:

Wegen $p \mapsto q$ gilt für alle Abläufe A und $i \in \mathbb{N}$: wenn $A(i) \models p$, dann existiert ein $j \geq i$ mit $A(j) \models q$.
Wegen $q \mapsto r$ gilt für alle Abläufe A und $j \in \mathbb{N}$: wenn $A(j) \models q$, dann existiert ein $k \geq j$ mit $A(j) \models r$.
Also gilt für alle Abläufe A und $i \in \mathbb{N}$: wenn $A(i) \models p$, dann existiert ein $k \geq i$ mit $A(j) \models r$.
Daher $p \mapsto r$. □

(vi) Abschwächung der Konsequenz:

$p \mapsto q$
, Prämisse
$q \mapsto r$
, aus der Prämisse $q \Rightarrow r$ und dem Implikationstheorem
$p \mapsto r$
, Transitivität von $\mapsto$ auf den letzten beiden Formeln □

(vii) Löschungstheorem:

$b \mapsto r$
, Prämisse
$q \mapsto q$
, Reflexivität von $\mapsto$
$q \vee b \mapsto q \vee r$
, Disjunktion der beiden letzten Formeln
$p \mapsto q \vee b$
, Prämisse
$p \mapsto q \vee r$
, Transitivität von $\mapsto$ auf den letzten beiden Formeln □

(viii) PSP-Theorem:

Aus $p \mapsto q$ folgt, daß für alle Abläufe A und $i \in \mathbb{N}$ gilt: wenn $A(i) \models p$, dann gibt es ein $j \geq i$ und $A(j) \models q$.
Darüberhinaus gilt für alle Transitionen t

$\{r \wedge \neg b\}\ t\ \{r \vee b\}$

wegen *r unless b*. Daher gilt: wenn $A(i) \models (p \wedge r \wedge \neg b)$, dann gibt es ein $j \geq i$ und $A(j) \models (q \wedge ((r \wedge \neg b) \vee r \vee b))$, d.h. $A(j) \models (q \wedge r) \vee b$. Also:
$p \wedge r \wedge \neg b \mapsto (q \wedge r) \vee b$.
Zusammen mit $p \wedge r \wedge b \mapsto (q \wedge r) \vee b$ wegen der Reflexivität von $\mapsto$, ergibt sich durch Disjunktion der beiden Formeln: $p \wedge r \mapsto (q \wedge r) \vee b$. □

(ix) Vervollständigungstheorem:

Wir nehmen zuerst die Gültigkeit einer einfacheren Regel an, die wir später beweisen:

$$\frac{\begin{array}{c} p \mapsto q,\ p' \mapsto q', \\ q \ \mathit{unless}\ r,\ q' \ \mathit{unless}\ r \end{array}}{p \wedge p' \mapsto (q \wedge q') \vee r} \qquad (1)$$

Aus dieser Regel können wir ableiten

$$\frac{\begin{array}{c} p \mapsto q \vee r,\ p' \mapsto q' \vee r, \\ q \ \mathit{unless}\ r,\ q' \ \mathit{unless}\ r \end{array}}{p \wedge p' \mapsto (q \wedge q') \vee r} \qquad (2)$$

Das ist ein Spezialfall des Vervollständigungstheorems mit $N = 2$. Der allgemeine Fall folgt durch Induktion über N.

Beweis: (2) aus (1)

$q \;\; unless \;\; r \,, \;\; q' \;\; unless \;\; r$
, Prämisse in (2)
$q \vee r \;\; unless \;\; r \,, \;\; q' \vee r \;\; unless \;\; r$
, einfache Disjunktion mit $r \; unless \; r$
$p \wedge p' \;\mapsto\; ((q \vee r) \wedge (q' \vee r)) \vee r$
, aus den letzten beiden Formeln und der Prämisse $p \mapsto q \vee r,\; p' \mapsto q' \vee r$ von (2) angewandt auf (1)
$p \wedge p' \;\mapsto\; (q \wedge q') \vee r$
, aus der letzen Formel durch Abschwächung der Konsequenz

Beweis: (1)

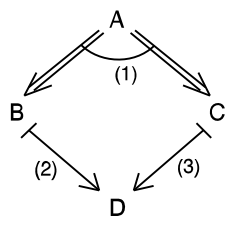
Aus $p \mapsto q$ folgt, daß für alle Abläufe A und alle $i \in \mathbb{N}$ gilt: wenn $A(i) \models p$, dann gibt es ein $j \geq i$ und $A(j) \models q$.
Analog gilt für $p' \mapsto q'$: wenn $A(i) \models p'$, dann gibt es ein $k \geq i$ und $A(k) \models q'$.
Wegen der Prämisse $q \; unless \; r$, $q' \; unless \; r$ gilt
$\{q \wedge \neg r\} \;\; t \;\; \{q \vee r\}$ und
$\{q' \wedge \neg r\} \;\; t \;\; \{q' \vee r\}$
für alle Transitionen t, daher:
wenn $A(i) \models p \wedge p'$, dann gibt es entweder ein $j \geq i$ und $A(j) \models q \wedge q'$, oder es gibt ein $k \geq i$ und $A(k) \models r$.
Also $p \wedge p' \;\mapsto\; (q \wedge q') \vee r$. □

Proof Lattices

Bei der Herleitung von Lebendigkeitseigenschaften mit $\mapsto$ benutzen wir gelegentlich eine graphische Darstellungsform. Angenommen, die folgenden Formeln gelten:

(1) $A \;\Rightarrow\; B \vee C$
(2) $B \mapsto D$
(3) $C \mapsto D$

Die Ableitung der Eigenschaft $A \mapsto D$ aus (1), (2) und (3) schreiben wir folgendermaßen:



Im oberen Teil wenden wir das Implikationstheorem auf $\Rightarrow$ an und können $A \mapsto B \vee C$ ablesen. Danach wenden wir zweimal das Löschungstheorem an:

a) $A \mapsto B \vee C$ und $B \mapsto D$ ergibt $A \mapsto D \vee C$
b) $A \mapsto D \vee C$ und $C \mapsto D$ ergibt $A \mapsto D$

Die theoretische Grundlage bildet das Konzept der Proof Lattices von L. Lamport und S. Owicki [OWIC82]:

4.4.2 Definition *Proof Lattice*

Sei Ω ein beschriftetes, 1-sicheres Netz mit Fairness. Ein *Proof Lattice für* Ω ist ein endlicher gerichteter azyklischer Graph, in dem gilt:

(i) Jeder Knoten ist durch einen Ausdruck aus $EXPR_\Omega$ beschriftet.
(ii) Jede Kante ist entweder durch $\Rightarrow$ oder $\mapsto$ beschriftet.
(iii) Es gibt genau einen *Startknoten*, der keine ankommenden Kanten hat.
(iv) Es gibt genau einen *Zielknoten*, der keine abgehenden Kanten hat.
(v) Alle anderen Knoten haben mindestens eine ankommende und eine abgehende Kante.
(vi) Wenn ein Knoten durch A beschriftet ist und seine abgehenden Kanten auf Knoten zeigen, die mit $A_1,\ldots,A_k$ beschriftet sind, dann ist

$$A \mapsto (A_1 \vee A_2 \ldots \vee A_k)$$

eine Eigenschaft von Ω.

Das folgende Lemma stellt den Zusammenhang zwischen Proof Lattices und den Ableitungsregeln von L_Ω her:

4.4.3 Lemma

Sei Ω ein beschriftetes, 1-sicheres Netz mit Fairness. Falls es einen Proof Lattice für Ω gibt, in dem der Startknoten mit A beschriftet ist und der Zielknoten mit B, dann gilt $A \mapsto B$ in Ω.

(Beweis analog zu [OWIC82] S. 467).

4.5 Fair

In Definition 3.6.2 wurden lokale Fairness-Bedingungen für beschriftete Netze eingeführt. Eine Stelle p heißt fair genau dann, wenn alle Transitionen, die im Nachbereich von p liegen, unendlich oft schalten dürfen, falls sie unendlich oft aktiviert sind. Durch den Fair-Operator in L_Ω können wir über diese besondere Art der Lebendigkeit argumentieren. Die Formel *r fair* sprechen wir als "*r* ist fair" und sie bedeutet: alle Transitionen, die unter Markierungen aktiviert sind, in denen $(a \wedge r \wedge \neg b)$ gilt mit *a unless b* (für feste

Prädikate a und b), können unendlich oft schalten, wenn sie unendlich oft aktiviert sind (in allen Abläufen von Ω). Die Semantik des Operators entspricht dabei weitgehend der Definition der lokalen Fairness-Bedingungen im Netz, allerdings erfolgt die Lokalisierung der fair zu behandelnden Transitionen durch beliebige Prädikate, während die Fairness-Bedingungen durch Stellen notiert werden. Daher erfüllt ein Netz mit der Fairness-Bedingung p trivialerweise eine Formel $u.p\ fair$.

4.5.1 Theorem *Fairness*

$$\frac{\begin{array}{c} p \ \ ensures^{r} \ \ q\ , \ \ r\ fair, \\ p \wedge r \ \ unless \ \ q \vee q', \\ q' \ \mapsto \ r \vee q \end{array}}{p \wedge r \ \mapsto \ q}$$

Beweis:

(i) Wir rekapitulieren: $r\ fair$, d.h. wenn es t, β gibt mit $\{p \wedge r \wedge \neg q\} \ \overrightarrow{t,\beta} \ \{q\}$ und $p\ unless\ q$ dann gilt:
wenn es für jedes $i \in \mathbb{N}$ ein $j \geq i$ gibt mit $A(j) \models (p \wedge r \wedge \neg q)$, dann gibt es zu jedem i auch ein $k \geq i$ und $A(k) \models q$.

(ii) Wir nehmen an, es gibt einen Ablauf A und ein $i \in \mathbb{N}$ mit $A(i) \models (p \wedge r \wedge \neg q)$ zu dem es kein $j \geq i$ gibt mit $A(j) \models q$.

(a) Dann gibt es jedoch ein $k > i$ und $A(k) \models q'$, wegen $p\ ensures^{r}\ q$ und $p \wedge r\ unless\ q \vee q'$: da p laut Annahme nicht zu q werden darf, aber es auch keine persistent aktivierten Transitionen gibt, muß r irgendwann zu q' werden.

(b) Wegen $q' \mapsto r \vee q$ und $p\ unless\ q$ (aus der Prämisse $p\ ensures^{r}\ q$) gibt es dann ein $j \geq k$ mit $A(j) \models (p \wedge r \wedge \neg q)$. Hier gilt wieder unsere Annahme und wir können erneut (a) und (b) anwenden.

(c) Also gibt es zu jedem $i \in \mathbb{N}$ ein $j \geq i$ mit $A(j) \models (p \wedge r \wedge \neg q)$, aber es gibt kein $k \geq i$ mit $A(k) \models q$ wegen unserer Annahme. Das ist jedoch ein Widerspruch zur Prämisse $r\ fair$, dessen Voraussetzung wegen $p\ ensures^{r}\ q$ erfüllt ist: es gibt ein t, β mit $\{p \wedge r \wedge \neg q\} \ \overrightarrow{t,\beta} \ \{q\}$ und $p\ unless\ q$.

(iii) Die Annahme in (ii) muß also falsch sein: es gibt keinen Ablauf A und kein $i \in \mathbb{N}$ mit $A(i) \models (p \wedge r \wedge \neg q)$ zu dem es kein $j \geq i$ gibt mit $A(j) \models q$. Es gilt daher $p \wedge r \mapsto q$. □

5 Systemstrukturierung durch Komposition

Am Beispiel des parallelen Garbage Collectors wird sich zeigen, daß es nötig sein kann, die Eigenschaften eines Systems auf den Eigenschaften der Teilsysteme abzustützen. Auch kann ein modularer Entwurf Wesentliches zum Verständnis des Gesamtsystems beitragen. In unserem Fallbeispiel sind der Collector und der Mutator zwei weitgehend unabhängige Komponenten, deren dynamisches Verhalten getrennt voneinander spezifiziert werden kann. Natürlich ist es unerläßlich, daß die eine Spezifikation auf die jeweils andere Komponente Rücksicht nimmt. Schließlich sollen ja beide Komponenten miteinander kooperieren.

Im folgenden beschreiben wir, wie zwei beschriftete, 1-sichere Netze durch Vereinigung komponiert werden können und welcher Zusammenhang zwischen den Eigenschaften der Komposition und den Eigenschaften der Komponenten besteht.

5.1 Netzkomposition durch Vereinigung

Zur Vereinfachung nehmen wir an, daß zwei 1-sichere Netze komponiert werden sollen, die beide über der gleichen algebraischen Spezifikation beschriftet sind. Damit ist gewährleistet, daß beide Netze auf dem gleichen abstrakten Datentyp operieren. Wichtig ist darüberhinaus, daß die Transitionsmengen der beiden Netze disjunkt sind, während die Stellenmengen identisch sein müssen, d.h. die beiden Netze $N_1 = (P,T_1,F_1)$ und $N_2 = (P,T_2,F_2)$ manipulieren die gleichen Prädikate, aber auf verschiedene Art und Weise. Die Stellen, die in der einen Komponente isoliert sind, repräsentieren die privaten Prädikate der anderen Komponente. Die beschriebenen Vereinfachungen bzgl. der algebraischen Spezifikation und der Stellenmenge sind nicht notwendig für die Vereinigung von Netzen, aber sie vereinfachen den Formalismus. Darüberhinaus fordern wir, daß das komponierte Gesamtsystem wieder 1-sicher sein muß.

5.1.1 Definition *Komposition durch Vereinigung*

Seien $\Omega_i = (N_i,ins_i,E,FAIR_i)$ über $SPEC = (S,OP,E)$ beschriftete 1-sichere Netze mit Fairness, $N_i = (P,T_i,F_i)$, $ins_i = (\varphi,M^\circ,\lambda_i)$ für $i \in \{1,2\}$. $\Omega = (N,ins,E,FAIR)$ mit $N = (P,T,F)$ und $ins = (\varphi,M^\circ,\lambda)$ heißt die *Vereinigung* von Ω_1 und Ω_2 g.d.w.

(i) $T = T_1 \dot\cup T_2$, $F = F_1 \dot\cup F_2$, wobei $\dot\cup$ die disjunkte Vereinigung bezeichnet.

(ii) Für alle $f \in F$ gilt

$$\lambda(f) = \begin{cases} \lambda_1(f) & \text{falls } f \in F_1 \\ \lambda_2(f) & \text{sonst} \end{cases}$$

(iii) $FAIR = FAIR_1 \cup FAIR_2$.

(iv) Ω ist 1-sicher.
Wir schreiben dann $\Omega = \Omega_1 [\!] \Omega_2$.

5.2 Eigenschaften komponierter Netze

Nützlich ist die Komposition von Netzen vor allem deshalb, weil Eigenschaften, die wir in den Teilsystemen ablesen, u.U. auch im Gesamtsystem gelten. Für den Umgang mit Eigenschaften der Komponenten benutzen wir die folgende Notation:

5.2.1 Schreibweise

Seien Ω_1, Ω_2 zwei beschriftete, 1-sichere Netze mit $\Omega = \Omega_1 [\!] \Omega_2$. Für $A \in L_\Omega$ schreiben wir
(i) A in Ω_1 bzw. A in Ω_2 und meinen $\Omega_1 \models A$ bzw. $\Omega_2 \models A$.
(ii) A und meinen A in Ω.

Das Vereinigungstheorem faßt die Regeln zusammen, die wir bei der Ableitung von Eigenschaften der Komposition aus den Eigenschaften der Komponenten benutzen können:

5.2.2 Theorem *Vereinigung*

Sei $\Omega = \Omega_1 [\!] \Omega_2$ die Vereinigung zweier 1-sicherer, beschrifteter Netze mit Fairness.
Dann gelten die folgenden Ableitungsregeln:
(i)

$$\frac{p \;\; unless_\kappa \;\; q \;\; \text{in} \;\; \Omega_1 \,, \;\; p \;\; unless_\kappa \;\; q \;\; \text{in} \;\; \Omega_2}{p \;\; unless_\kappa \;\; q \;\; \text{in} \;\; \Omega_1 [\!] \Omega_2}$$

(ii)

$$\frac{p \;\; ensures_\kappa \;\; q \;\; \text{in} \;\; \Omega_1 \,, \;\; p \;\; unless_\kappa \;\; q \;\; \text{in} \;\; \Omega_2}{p \;\; ensures_\kappa \;\; q \;\; \text{in} \;\; \Omega_1 [\!] \Omega_2}$$

(iii)

$$\frac{p \;\; ensures^r_\kappa \;\; q \;\; \text{in} \;\; \Omega_1 \,, \;\; p \;\; unless_\kappa \;\; q \;\; in \;\; \Omega_2}{p \;\; ensures^r_\kappa \;\; q \;\; in \;\; \Omega_1 [\!] \Omega_2}$$

(iv)

$$\frac{r \;\; fair_\kappa \;\; q \;\; \text{in} \;\; \Omega_1}{r \;\; fair_\kappa \;\; q \;\; \text{in} \;\; \Omega_1 [\!] \Omega_2}$$

Beweis:

(i) p *unless*$_\kappa$ q in Ω_1 und p *unless*$_\kappa$ q in Ω_2, daraus folgt:
für alle t in Ω_1: $\{p \wedge \neg q\}_\kappa \; t \; \{p \vee q\}$ und
für alle t in Ω_2: $\{p \wedge \neg q\}_\kappa \; t \; \{p \vee q\}$.
Daher:
für alle t in $\Omega_1 [\!] \Omega_2$: $\{p \wedge \neg q\}_\kappa \; t \; \{p \vee q\}$. □

(ii) p *ensures*$_\kappa$ q in Ω_1 und p *unless*$_\kappa$ q in Ω_2, daraus folgt:
für alle t in Ω_1: $\{p \wedge \neg q\}_\kappa \; t \; \{p \vee q\}$ und
es gibt ein t und ein β in Ω_1 mit $\{p \wedge \neg q\}_\kappa \; \vec{t,\beta} \; \{q\}$ und
für alle t in Ω_2: $\{p \wedge \neg q\}_\kappa \; t \; \{p \vee q\}$.
Daher:
p *unless*$_\kappa$ q in $\Omega_1 [\!] \Omega_2$ und es gibt ein t und ein β in $\Omega_1 [\!] \Omega_2$ mit
$\{p \wedge \neg q\}_\kappa \; \vec{t,\beta} \; \{q\}$, also p *ensures*$_\kappa$ q in Ω. □

(iii) Analog zum Beweis von (ii). □

(iv) Folgt aus der Vereinigung der Fairness-Bedingungen in Definition 5.1.1 (iii). □

Dabei ist anzumerken, daß diese Regeln grundsätzlich nur für kompositionale Operatoren (d.h. solche mit Index κ) gelten. Weitere nützliche Ableitungsregeln können aus dem Vereinigunstheorem durch Abschwächung der Konsequenz bei *unless* gefolgert werden:

5.2.3 Korollar

Sei $\Omega = \Omega_1 [\!] \Omega_2$ die Vereinigung zweier 1-sicherer, beschrifteter Netze. Dann gilt
(i)

$$\frac{p \;\; \textit{unless}_\kappa \;\; q \;\; \text{in} \;\; \Omega_1 \,, \;\; \mathbf{stable}_\kappa \;\; p \;\; \text{in} \;\; \Omega_2}{p \;\; \textit{unless}_\kappa \;\; q \;\; \text{in} \;\; \Omega_1 [\!] \Omega_2}$$

(ii)

$$\frac{\mathbf{invariant}_\kappa \;\; p \;\; \text{in} \;\; \Omega_1 \,, \;\; \mathbf{stable}_\kappa \;\; p \;\; \text{in} \;\; \Omega_2}{\mathbf{invariant}_\kappa \;\; p \;\; \text{in} \;\; \Omega_1 [\!] \Omega_2}$$

(iii)

$$\frac{p \;\; \textit{ensures}_\kappa \;\; q \;\; \text{in} \;\; \Omega_1 \,, \;\; \mathbf{stable}_\kappa \;\; p \;\; \text{in} \;\; \Omega_2}{p \;\; \textit{ensures}_\kappa \;\; q \;\; \text{in} \;\; \Omega_1 [\!] \Omega_2}$$

Abbildung 5.1 soll das Vereinigungstheorem anhand eines Beispiels verdeutlichen. Teilsystem M hat die Eigenschaften a *ensures*$_\kappa$ b und $\mathbf{stable}_\kappa$ b, während im Teilsystem C die Eigenschaften b *ensures*$_\kappa$ a und $\mathbf{stable}_\kappa$ a gelten. Ohne erneut Eigenschaften aus $M [\!] C$

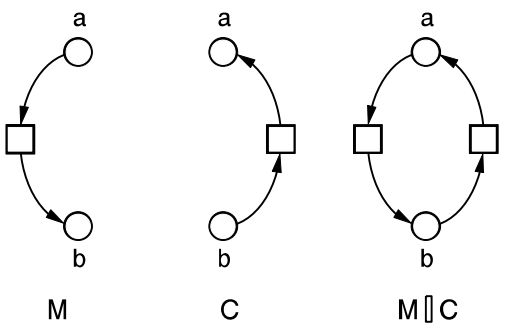


Abb. 5.1. Die Netze M und C werden komponiert

ablesen zu müssen, können wir ableiten, daß a *ensures*$_\kappa$ b und b *ensures*$_\kappa$ a Eigenschaften der Komposition $M \,[\!]\, C$ sind.

5.3 Das Substitutions-Axiom

Formeln der Art **invariant** a haben eine besonders praktische Eigenschaft: da a in dem betrachteten Netz in allen erreichbaren Markierungen gilt, kann a mit *true* gleichgesetzt werden bzw. überall, wo *true* steht, darf *true* durch a substituiert werden. Auf diese Weise können Informationen über den Startzustand des Systems in alle Formeln unserer Sprache $L\Omega$ gelangen. So lassen sich Eigenschaften ableiten, die ohne das Wissen über die Startmarkierung nicht vefügbar wären[1]. Als Konsequenz des Substitutions-Axioms ergeben sich z.B. die folgenden Ableitungsregeln:

$$\frac{\textbf{invariant}\ \ a \vee b\,,\ \ a \vee b \ \mapsto\ c}{\mathit{true} \ \mapsto\ c}$$

$$\frac{a\ \ \mathit{unless}\ \ b\,,\ \ \textbf{invariant}\ \neg b}{\textbf{stable}\ \ a}$$

Eine Einschränkung des Substitutions-Axioms ergibt sich jedoch hinsichtlich der kompositionalen Operatoren mit Index κ: Da *unless*$_\kappa$, *ensures*$_\kappa$ und *fair*$_\kappa$ über alle Markierungen argumentieren und nicht nur über die erreichbaren Markierungen, sind Eigenschaften, die durch Anwendung des Substitutions-Axioms entstanden sind, nicht mehr kompositional. Insbesondere verlieren dabei die Operatoren den Index κ. Die eben erwähnte Regel lautet für Operatoren mit Index κ daher wie folgt:

$$\frac{a\ \ \mathit{unless}_\kappa\ \ b\,,\ \ \textbf{invariant}_\kappa\ \neg b}{\textbf{stable}\ \ a}$$

1) vgl. [CHAN88], S. 49

Ableitbar ist also nur **stable** a und nicht etwa $\mathbf{stable}_\kappa\, a$. Diese Einschränkung ist jedoch nur dann zu beachten, wenn Teilsysteme durch Vereinigung ihrer Transitionsmengen zu einem Gesamtsystem komponiert werden. In diesem Fall darf das Substitutions-Axiom nur auf Eigenschaften des Gesamtsystems angewendet werden.

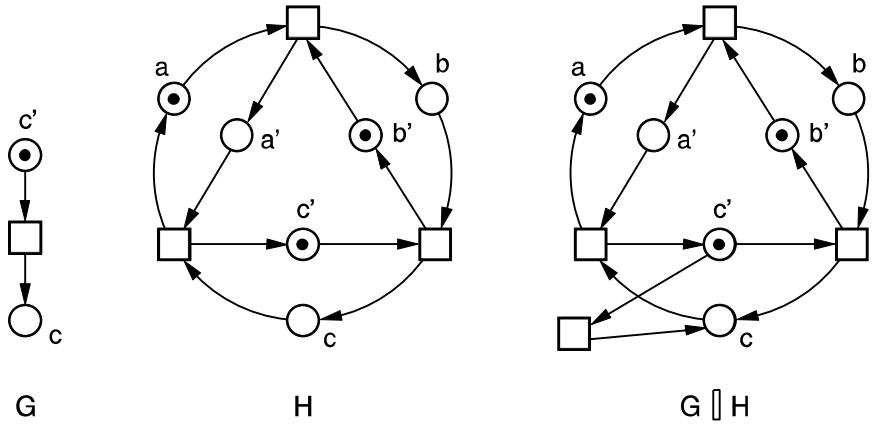


Abb. 5.2. Komposition der Netze G und H

Dieser Sachverhalt soll anhand eines Beispiels veranschaulicht werden. Abbildung 5.2 zeigt die beiden Netze G und H sowie ihre Komposition $G \,[\!]\, H$. Über den Stellen a,b und c liegt in H eine S-Invariante, aus der wir ablesen

$$\mathbf{invariant}_\kappa \quad a \;\; or \;\; b \;\; or \;\; c \quad \text{in } H.$$

Zusammen mit dem Substitutions-Axiom können wir (fälschlicherweise mit Index κ) herleiten

$$\mathbf{invariant}_\kappa \quad a \Rightarrow \neg b \quad \text{in } H.$$

Da a und b in G stabil sind, gilt auf jeden Fall

$$\mathbf{invariant}_\kappa \quad a \Rightarrow \neg b \quad \text{in } G.$$

Aus den beiden letzten Formeln und dem Vereinigungstheorem können wir dann ableiten

$$\mathbf{invariant}_\kappa \quad a \Rightarrow \neg b \quad \text{in } G \,[\!]\, H.$$

Diese Formel ist jedoch keine Eigenschaft von $G \,[\!]\, H$, was wir anhand eines kurzen Markenspiels nachweisen können.

TEIL 3

Ein paralleler Garbage Collector als Petrinetz

6 Problemanalyse

6.1 Aufgabenstellung

Gegeben ist ein Mutator, der seine Datenstruktur gemäß einer uns unbekannten Programmvorschrift manipuliert. Gegeben sind auch n Speicherzellen, aus denen die Datenstruktur des Mutators aufgebaut werden kann. Am Anfang sind $n - 1$ Speicherzellen als frei markiert, während eine der Zellen den Grundstock der Datenstruktur bildet. Wir werden sie im folgenden als *Wurzel* bezeichnen. Verändert wird die Datenstruktur, indem der Mutator einen neuen Zeiger von einer Speicherzelle aus auf eine andere richtet, oder bereits bestehende Zeiger löscht. Neue Zeiger werden entweder auf Zellen gerichtet, die bereits in der Datenstruktur enthalten sind, oder auf Zellen, die bisher als frei markiert waren. In diesem Fall vergrößert sich die Datenstruktur um eine Speicherzelle. Sie verkleinert sich, falls durch das Löschen eines Zeigers eine oder mehrere Speicherzellen nicht mehr von der Wurzel aus durch eine Folge von Zeigern erreicht werden kann. Speicherzellen mit dieser Eigenschaft bezeichnen wir als Müll. Unsere Aufgabe besteht darin, einen zum Mutator nebenläufigen Garbage Collector zu entwerfen, der die zu Müll gewordenen Speicherzellen erkennt und sie wieder als frei markiert.

6.2 Abstraktionen

Zum Zwecke der formalen Darstellung müssen wir von der bereits erwähnten Datenstruktur des Mutators abstrahieren und eine gleichwertige mathematische Beschreibung finden. Durch einen gerichteten Graphen mit Knoten und Kanten wird von der Datenstruktur, den Speicherzellen und den Zeigern abstrahiert. Dabei übernimmt der Graph die Funktion der Datenstruktur, die Knoten entsprechen den Speicherzellen, und die Kanten ersetzen die Zeiger.

6.2.1 Definition *Gerichteter Graph, Pfad, Zyklus*

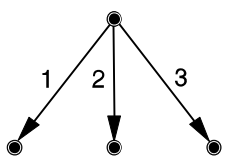
(i) Eine Paar $G = (V,E)$ heißt *gerichteter Graph* g.d.w.
  (a) V ist eine endliche Menge. Die Elemente von V nennen wir *Knoten*.
  (b) $E \subseteq V \times V$ ist die Menge der *gerichteten Kanten* von a nach b für alle $(a,b) \in E$.

(ii) Eine Folge $(v_1,\ldots,v_n)$ mit $n \geq 2$ heißt *Pfad* von v_1 nach v_n g.d.w. $(v_i,v_{i+1}) \in E$ für alle $1 \leq i \leq n-1$.

(iii) Ein Pfad $(v_1,\ldots,v_n)$ heißt *Zyklus* g.d.w. $v_1 = v_n$.

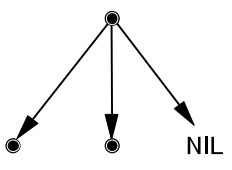
Auf den ersten Blick scheint die Darstellung des Speichergraphen wenig auf die Anforderungen der Praxis einzugehen. Tatsächlich aber ist die Modellierung der Datenstruktur eines Prozesses als gerichteter Graph eine gültige und berechtigte Abstraktion:

- *Geordnete Söhne*: In allen realen Programmen sind die Zeiger einer Speicherzelle geordnet. Es gibt also einen ersten Nachfolgeknoten, einen zweiten Nachfolgeknoten usw.. Diese Eigenschaft ist jedoch ausschließlich für ein problemgemäßes Verhalten des Mutators wichtig. Für den Collector und seine Zusammenarbeit mit dem Mutator ist die Reihenfolge der abgehenden Kanten ohne Bedeutung.



- *Kantenduplikate*: Eine Speicherzelle kann mehrere Zeiger besitzen, die auf die gleiche Speicherzelle gerichtet sind. Auch diese Eigenschaft beeinflußt nicht die Funktion unseres Collectors, da für ihn nur wichtig ist, daß eine Verbindung zwischen zwei Knoten existiert, nicht aber wieviele Verbindungen.

- *NIL-Zeiger*: Meistens ist die Anzahl der Zeiger, die eine Speicherzelle enthält, festgelegt. Die Zeiger, die unbenutzt sind, werden mit der Markierung NIL versehen. Die in unserem Modell benutzte Abstraktion läßt eine variable Anzahl von Zeigern zu und stellt eine Verallgemeinerung der realen Gegebenheiten dar. NIL-Zeiger werden dadurch überflüssig.



6.3 Beschreibung des Mutators

Wir haben uns zur Aufgabe gemacht, einen funktionsfähigen parallelen Garbage Collector zu konstruieren. Anhand des Konsistenzproblems (vgl. Abschnitt 1.4) kann gezeigt werden, daß es unmöglich ist, einen zum Mutator nebenläufigen Collector zu haben, ohne daß der Mutator den Collector gelegentlich unterstützt [DIJK76]. In welcher Form diese Hilfe erfolgen muß, oder wie stark wir den Mutator dabei einschränken dürfen, kann nicht von vorne herein formal spezifiziert werden. Später, wenn wir den Collector implementiert und seine Korrektheit nachgewiesen haben, werden die Anforderungen, die an den Mutator zu stellen sind, formal vorliegen. Es gilt dann zu prüfen, ob diese "Kooperationsbedingungen" annehmbar sind.

Damit ergibt sich die folgende Methodik: wir implementieren einen parallelen Garbage Collector als Petrinetz ohne dabei den Mutator zu implementieren. Dann geben wir eine formale Spezifikation des Collectors an, die anschließend solange (hierarchisch) verfeinert wird, bis die Formeln der verfeinerten Spezifikation direkt an der Implementierung "abgelesen" werden können. Während der Verfeinerung der Collector-Spezifikation entstehen gleichzeitig die Kooperationsbedingungen, die wir an den Mutator stellen müssen, damit unser Collector seine formale Spezifikation erfüllen kann. Jedes Petrinetz, das diese Kooperationsbedingungen erfüllt, ist in unserem Sinne ein Mutator und kann mit dem Collector-Petrinetz vereinigt werden (vgl. 5.1.1). Durch diese Verfahrensweise ist es uns möglich einen nebenläufigen Garbage Collector zu implementieren und dessen Korrektheit nachzuweisen, ohne über einen konkreten Mutator argumentieren zu müssen.

6.4 Beschreibung des Collectors

Die Aufgabe der automatischen Wiedergewinnung von Speicherzellen obliegt dem Collector und sie besteht darin, jeden Knoten des Speichergraphen, der nicht mehr von der Wurzel aus durch einen Pfad erreichbar ist, als solchen zu erkennen und ihn als "frei" zu markieren. Die derart markierten Knoten können vom Mutator wieder in seine Datenstruktur aufgenommen werden.

Vom Collector erwarten wir, daß er durch eine Betrachtung des Speichergraphen für jeden Knoten entscheidet, ob er noch erreichbar ist oder nicht. Wir erwarten darüber hinaus, daß er diese Entscheidung nach absehbarer Zeit trifft. Wir fordern sogar, daß er sie immer wieder trifft. Andererseits dulden wir auch eine kleine Nachlässigkeit: Die Entscheidung muß nur dann richtig sein, wenn sich der Collector bei einem Knoten für "unerreichbar" entscheidet. Wir fordern also *Sicherheit*: Der Collector darf keinen Knoten freigeben, der noch nicht Müll ist. Und wir fordern *Lebendigkeit*: Jeder Knoten, der Müll ist, muß nach endlicher Zeit erkannt werden. Daß der Collector gelegentlich Knoten

übersieht, die eigentlich schon längst unerreichbar geworden sind, sehen wir ihm nach, solange er diese Knoten nur irgendwann erkennt.

Um entscheiden zu können, ob ein Knoten von der Wurzel aus erreichbar ist, muß aus der *Verbindungsrelation E* des Graphen die *Erreichbarkeitsrelation R* ($= E^*$) — also die reflexive transitive Hülle von E — berechnet werden (R steht für engl. "Reachable"). Dabei ergibt sich folgendes Problem: Die Verbindungsrelation E wird sich während der Berechnung von R verändern, da Mutator und Collector ja parallel arbeiten sollen. Doch hier hilft unsere Nachlässigkeit bei der informellen Spezifikation: Der Collector darf gelegentlich Knoten übersehen, die bereits Müll sind, d.h. er muß R nicht exakt berechnen. Mit "nicht exakt" ist gemeint, daß die Näherung von R, die der Collector berechnet und die Grundlage seiner Entscheidungen sein wird, mindestens R enthalten muß, aber auch größer sein kann. Es ist also erlaubt, daß die Näherung auch die Erreichbarkeit eines Knoten ausweist, die nicht mehr gegeben ist. Um der Freizügigkeit Grenzen zu setzen, fordern wir jedoch, daß jeder Knoten, der Müll geworden ist, nach spätestens einem kompletten Collector-Zyklus erkannt wird. Mit anderen Worten: Knoten, die zu Beginn eines Collector-Zyklus bereits Müll sind, müßen im aktuellen Zyklus erkannt werden; Knoten, die erst während eines Collector-Zyklus unerreichbar werden, dürfen vom Collector im aktuellen Zyklus übersehen werden.

Voraussetzung für eine — in unserem Sinne — korrekte Näherung von R ist, daß der Mutator selbständig in der aktuellen Näherung des Collectors Eintragungen vornimmt, wenn er einen Knoten oder einen Knotenbaum im Graphen "umhängt". Dann ist nämlich die Gefahr gegeben, daß der Collector, der ja zur Berechnung der Näherung von R verschiedene Stellen des Graphen zu verschiedenen Zeitpunkten beobachtet, zu dem Schluß kommt, daß der entsprechende Knotenbaum — entgegen der Realität — Müll ist. Das trifft z.B. dann ein, wenn der Collector die Stelle, an die der Knotenbaum neu angehängt werden soll, schon besichtigt hat, die Stelle(n), an denen er nur noch kurze Zeit hängt, aber noch nicht. Der Mutator stellt zuerst die neue Verbindung her und kappt dann die alten, bevor der Collector sie besichtigen konnte. Zur Veranschaulichung: Sie versuchen einen Gegenstand hinter Ihrem Rücken zu verstecken. Nach der Aufforderung, Ihre Hände zu zeigen, bieten Sie zuerst Ihre rechte Hand an (leer!) und dann Ihre linke Hand (auch leer!). So könnte auch der Mutator immer einen Weg finden, um den Collector zu täuschen (vgl. Abschnitt 1.4).

6.4.1 Ein nebenläufiger Graph-Markierer für den Collector

Die Abläufe, die im Collector stattfinden müssen, können jetzt beschrieben werden. Ein zyklisches System wird abwechselnd zwei Arbeiten durchführen:

- Inkrementelle, d.h. zum Mutator nebenläufige Berechnung einer Näherung der Erreichbarkeitsrelation R ($= E^*$) mit Unterstützung durch den Mutator in folgenden Fällen: a) Der Mutator vergrößert R durch Einbinden eines bisher freien Knotens in seine Datenstruktur. b) Der Mutator hängt einen Knotenbaum in der oben beschriebenen Weise um.

- Freigabe aller Knoten, die laut der berechneten Näherung von R nicht mehr in der Datenstruktur des Mutators enthalten sind.

Die Punkte 1 und 2 beschreiben die beiden Phasen, aus denen der Collectorablauf besteht. Sie werden als Markierungs- bzw. Bereinigungsphase bezeichnet, um die besondere Art ihrer jeweiligen Implementierung zu charakterisieren.

Die *Markierungsphase* wird durch einen *nebenläufigen Graph-Markierer* implementiert. Er berechnet eine Näherung der Erreichbarkeitsrelation in endlicher Zeit, und sein Speicheraufwand ist direkt proportional zur Anzahl der Knoten. Jeder Knoten wird dabei als Markierungsprozeß aufgefaßt, der sich anfänglich ruhig verhält und darauf wartet, eine Aktivierungsnachricht zu erhalten. Trifft sie ein, so wird der betreffende Knotenprozeß aktiviert und versendet seinerseits Aktivierungsnachrichten an alle seine Söhne. Anschließend markiert er den eigenen Knoten als “erreichbar” und terminiert. Der Prozeß eines bereits als “erreichbar” markierten Knoten wird durch eine erneute Aktivierungsnachricht nicht mehr aktiviert, um zu gewährleisten, daß die Markierungsphase auch dann zu einem Abschluß kommt, wenn der Graph Zyklen enthält. Gestartet wird der Graph-Markierer, indem eine Aktivierungsnachricht an die Wurzel des Graphen geschickt wird. Sind alle Knotenprozesse terminiert und keine Nachrichten mehr unterwegs, dann ist die Markierungsphase beendet.

In der *Bereinigungsphase* können jetzt alle Knoten, die nicht als “erreichbar” markiert wurden, freigegeben werden. Die Markierungen der anderen Knoten werden gelöscht, um die nächste Markierungsphase vorzubereiten.

Die Unterstützung des Mutators, die die Punkte a) und b) fordern, wird folgendermaßen an den Graph-Markierer angeschlossen: Jedesmal, wenn der Mutator eine neue Kante errichtet, schickt er an den Knoten, zu dem die neue Kante zeigt, eine Aktivierungsnachricht, um sicherzustellen, daß der Knoten selbst und der gegebenenfalls daran hängende Knotenbaum in der aktuellen Näherung sicher als erreichbar gelten.

7 Implementierung

Bevor wir ein Petrinetz konstruieren, das uns den gewünschten nebenläufigen Collector modelliert, versuchen wir uns erst an einer sequentiellen Version. Das bedeutet, wir können zu Beginn davon ausgehen, daß der Mutator stillsteht während der Collector sich den Graphen ansieht. Diese Vorgehensweise hat den Vorteil, daß wir uns in der Anfangsphase die Arbeit etwas leichter machen, und es wird sich herausstellen, daß bei geschickter Wahl des sequentiellen Collectors der parallele Collector nur einen Schritt weit davon entfernt liegt.

7.1 Ein sequentieller Collector

Die algebraische Semantik von Collector und Mutator wird durch einen abstrakten Datentyp $\mathbf{graph}^{(n)}$ festgelegt. Dabei bezeichnet der Parameter n die Anzahl der Knoten im Graphen. Das Netz unseres parallelen Garbage Collectors wird über der Signatur der Multimengen-Erweiterung von $\mathbf{graph}^{(n)}$ beschriftet werden.

7.1.1 Definition *Abstrakter Datentyp* ***graph***

$\mathbf{graph}^{(n)}$ =

sorts: *vertice*
edge

opns: $v_1,\ldots,v_n : \rightarrow vertice$
$r : \rightarrow vertice$
$EDGE : vertice\ vertice \rightarrow edge$
$PR_1 : edge \rightarrow vertice$
$PR_2 : edge \rightarrow vertice$

eqns: $x,y \in vertice$
$PR_1(EDGE(x,y)) = x$
$PR_2(EDGE(x,y)) = y$
$r = v_1$

Die Semantik dieser algebraischen Spezifikation ist die Menge aller **graph**-Algebren G, die isomorph zur Quotienten-Term-Algebra $T_{\mathbf{graph}^{(n)}}$ sind, d.h. $G \in ADT(\mathbf{graph}^{(n)})$.

Gehen wir im weiteren davon aus, wir hätten eine Algebra G, deren Trägermengen gemäß $\mathbf{graph}^{(n)}$ Knoten und gerichtete Kanten sind. Die Menge der Knoten ist fest (nennen wir sie V, für engl. “Vertices”), und die Menge der Kanten wird vom Mutator verändert (die nennen wir E für engl. “Edges”). Sobald der Collector mit seiner Arbeit

beginnt und sich den Graphen ansehen will, muß jede weitere Veränderung des Graphen unterbleiben. Wir haben im letzten Kapitel davon gesprochen, daß der Collector im allgemeinen Fall eine Näherung für die Erreichbarkeitsrelation R berechnen muß (R ist die reflexive, transitive Hülle von E), und ich behaupte, daß diese Näherung im sequentiellen Fall — den wir jetzt betrachten — immer exakt ist und R berechnet. Somit wird jeder Knoten erkannt, der nicht mehr von der Wurzel aus erreichbar ist.

Bei der Berechnung von R soll der Collector sich allerdings etwas einschränken. Wir wollen Platz sparen und auf keinen Fall die Relation R komplett aufbewahren müssen. Versuchen wir also, R auf den Informationsgehalt zu reduzieren, den wir mindestens brauchen. Der Collector muß nicht im allgemeinen wissen, welcher Knoten von welchem anderen Knoten aus erreichbar ist, sondern es reicht aus, zu wissen, welcher Knoten von der Wurzel aus erreichbar ist. Das bedeutet, wir können den Speicherbedarf auf die Quadratwurzel des ursprünglichen Platzbedarfs von R reduzieren. Wenn jeder Knoten mit einer zusätzlichen Markierung versehen wird, kann der Collector seine Aufgabe bereits erfüllen. Aus traditionellen Gründen beschreiben wir die Knotenmarkierung durch Farben: “weiß” sind am Anfang alle Knoten, bis auf die, die “frei” sind. “schwarz” wird ein Knoten, wenn der Collector erkannt hat, daß er von der Wurzel aus durch einen Pfad erreichbar ist. Wenn der Collector die Markierungsphase beendet, dann sind alle erreichbaren Knoten “schwarz” und die Müllknoten sind immer noch “weiß”. Wir werden bei der Beschriftung der Petrinetze die englischen Begriffe *white*, *black* und *free* benutzen, um darauf hinzuweisen, daß es sich um Symbole handelt. Darüberhinaus führen wir die folgenden Abkürzungen ein:

7.1.2 Schreibweise

Wir schreiben

(i) V statt $v_1 + v_2 + \ldots + v_n$.

(ii) $V \setminus r$ statt $V - r$.

(iii) (x,y) statt $EDGE(x,y)$.

(iv) F statt $\sum_{j=1}^{n} \sum_{k=1}^{n} (v_j,v_k)$.

(v) $f(u)$ statt $(u,v_1) + (u,v_2) + \ldots + (u,v_n)$.

Die Abkürzung $f(x)$ liefert zu einem Knoten x die Menge aller möglicherweise von x abgehenden Kanten.

Die Markierungsphase

Um die Farbe schwarz korrekt an die Knoten anbringen zu können, muß der Graph traversiert werden, d.h. ausgehend von der Wurzel wird jeder Knoten, der von dort aus erreichbar ist, vom Collector aufgesucht. Damit er dabei nicht die Übersicht verliert und insbesondere keinen Knoten vergißt oder etwa mehrmals besucht, benötigt er eine angemessene Buchhaltung:

- Der Collector merkt sich, an welchen Stellen im Graphen er gerade arbeitet. Das können Knoten sein oder auch die einzelnen abgehenden Kanten der Knoten. Letzteres hat den Vorteil, daß die Operationen des Collectors kleiner sind, da nicht der ganze Knoten auf einmal besucht wird, sondern jede Kante getrennt.
- Trifft er auf einen Knoten, der bereits schwarz ist, dann braucht er diesen Zweig im Graphen nicht weiter zu verfolgen, da er ihn bereits vorher schon besucht haben muß.

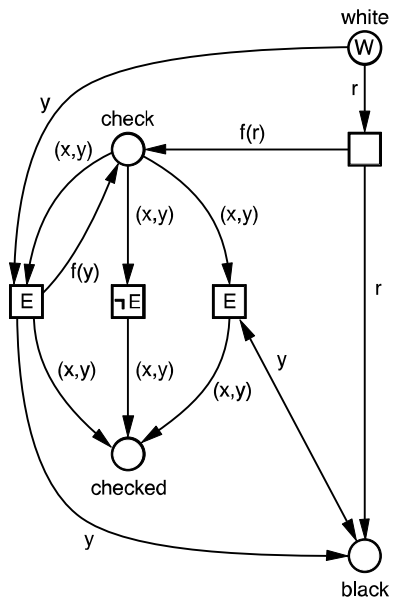


Abb. 7.1. Der Graph-Markierer im Collector

In Abbildung 7.1 wird die Markierungsphase als Petrinetz modelliert. Die Stellen *black* und *white* repräsentieren die vorhin besprochene Knotenmarkierung und in *check* merkt sich der Collector die abgehenden Kanten der Knoten, die er gerade besucht hat. *check* heißt die Stelle deshalb, weil er erst noch die Existenz der Kanten in *check* prüfen muß. Hat er also einen Knoten x besucht und schwarz markiert (weil x vorher weiß war), dann werden alle möglichen Kanten, die von x abgehen können (also genau $f(x)$), zur Überprüfung nach *check* gebracht, unabhängig davon, ob die Kanten im Graphen gerade existieren, oder nicht. Von dort aus übernehmen drei Transitionen die weitere Prüfung:

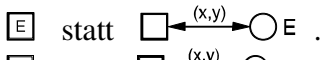
- Falls eine Kante nicht existiert, wird sie ignoriert und nach *checked* gebracht (das sind die überprüften Kanten). Diesem Fall widmet sich die mittlere Transition mit der Inschrift $\neg E$.

- Das gleiche geschieht mit einer Kante, die zwar existiert, aber auf einen schwarzen Knoten gerichtet ist. Das wird von der rechten Transition mit der Inschrift *E* und dem Doppelpfeil nach *black* erledigt.
- Die eigentliche Markierungsarbeit übernimmt die linke Transition mit der Inschrift *E*. Wenn nämlich die Kante existiert und der Knoten, auf den sie zeigt noch weiß ist, dann wird der Knoten schwarz markiert, und alle seine abgehenden Kanten werden nach *check* zur Überprüfung gebracht.

Die Inschriften der drei Transitionen (*E* bzw. $\neg E$) sollen bedeuten, daß die jeweilige Transition nur dann schalten kann, wenn die gerade anstehende Kante (x,y) existiert (d.h. sie befindet sich in *E*) bzw. nicht existiert ($(x,y) \notin E$). Wir benutzen die Inschriften *E* und $\neg E$ als Abkürzung für Schlingen zu den Stellen E bzw $\neg E$:

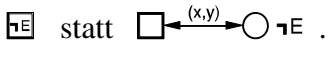
7.1.3 Schreibweise

Wir schreiben

(i) 

statt

.

(ii) 

statt

.

Noch ein Wort zur Startmarkierung in Abbildung 7.1: die Menge *W* von Knoten, die sich dort anfänglich in der Stelle *white* befindet, bezeichnet alle Knoten, die nicht frei sind. Insbesondere muß der Wurzelknoten dort enthalten sein (die Wurzel nennen wir *r* für engl. “root”). Um den Collector zu starten, muß die Wurzel schwarz markiert werden und alle Kanten, die möglicherweise von ihr wegführen, nach *check* gebracht werden. Der Collector hat die Markierungsphase beendet, wenn es keine Kanten mehr gibt, die überprüft werden müßen, d.h. *check* ist leer. In diesem Fall sind alle Knoten, die von der Wurzel aus durch einen Pfad erreichbar sind, schwarz markiert.

Wer im Zusammenhang mit dem Netz in Abbildung 7.1 darüber besorgt ist, daß es keine Transition gibt, die schalten könnte, falls eine existierende Kante in *check* liegt, die auf einen Knoten in *free* zeigt, der sei auf den Korrektheitsbeweis im nächsten Teil dieser Arbeit verwiesen. Dort wird gezeigt, daß existierende Kanten, die auf freie Knoten zeigen, nie geprüft werden.

Die Bereinigungsphase

Der Collector hat gerade die Markierungsphase beendet, und alle von der Wurzel aus erreichbaren Knoten sind schwarz. Alle anderen Knoten sind entweder frei (diese Markierung hat der Collector nicht angerührt), oder sie sollten es gleich werden, denn sie sind während der Markierungsphase weiß geblieben und sind daher Müll. Die Aufgabe des Collectors besteht jetzt darin, alle weißen Knoten als frei zu markieren und — in Erwartung der nächsten Markierungsphase — die schwarzen Knoten wieder weiß zu machen.

Diesen Vorgang modelliert das Petrinetz in Abbildung 7.2. Dort gelten am Anfang alle Knoten als “zu bereinigen” (wir nennen das zugehörige Prädikat *sweep*, was im Eng-

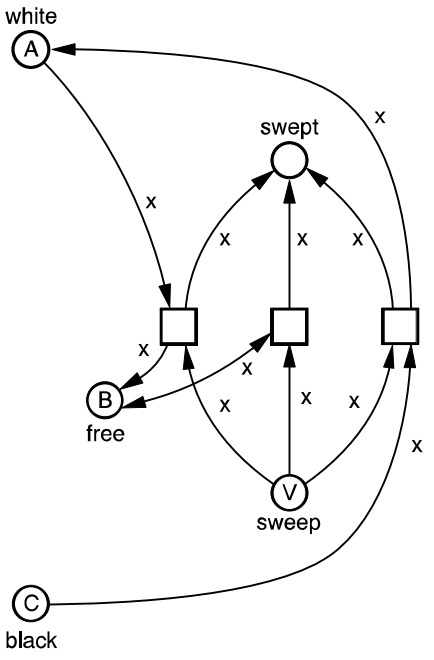


Abb. 7.2. Die Bereinigungsphase im Collector

lischen für “bereinigen” steht). Unbereinigte Knoten sind solche, deren Markierungen noch zu säubern sind. Im einzelnen heißt das:

- ein schwarzer Knoten wird weiß (dafür ist die rechte Transition zuständig)
- ein weißer Knoten wird frei (durch die linke Transition)
- und freie Knoten bleiben frei (durch die Transition in der Mitte).

Die Bereinigungsphase ist beendet, wenn die Markierungen aller Knoten bereinigt sind, d.h. alle Knoten befinden sich in der Stelle *swept* (steht für “bereinigt” im Englischen).

Als nächstes versuchen wir aus den beiden Teilnetzen der Abbildungen 7.1 und 7.2 einen kompletten Garbage Collector aufzubauen. Dabei ergibt sich jedoch die folgende Schwierigkeit: um zur Bereinigungsphase zu gelangen, muß erst die Terminierung der Markierungsphase festgestellt werden.

Terminierung der Markierungsphase

Wie bereits erwähnt wurde, ist die Markierungsphase dann beendet, wenn es keine Kanten mehr gibt, die überprüft werden müssen, d.h. *check* ist leer. Leider kann im Kalkül der Petrinetze nicht direkt abgeprüft werden, ob eine Stelle leer ist. Einfach dagegen ist es, zu prüfen, ob eine Stelle voll ist. Zum Glück existiert zwischen den beiden Sachverhalten ein einfacher Zusammenhang: eine Stelle ist genau dann leer, wenn

ihr Komplement voll ist. Was wir also benötigen, ist ein zu *check* komplementäres Prädikat, das von allen Transitionen in Abbildung 7.1 respektiert, d.h. neben *check* mitgeführt wird. Um an dieser Stelle aber nicht mehr von den einzelnen Kanten sprechen zu müssen, sondern über den ganzen Knoten, führen wir ein Prädikat ein, dessen Grundmenge Knoten sind und welches komplementär zu *check* und *checked* im folgenden Sinne ist:

- entweder ist ein Knoten x im Komplement (das wir *quiet* nennen werden, für "ruhig" im Englischen),
- oder jede der möglicherweise von x abgehenden Kanten ($= f(x)$) befindet sich entweder in *check* oder *checked*.

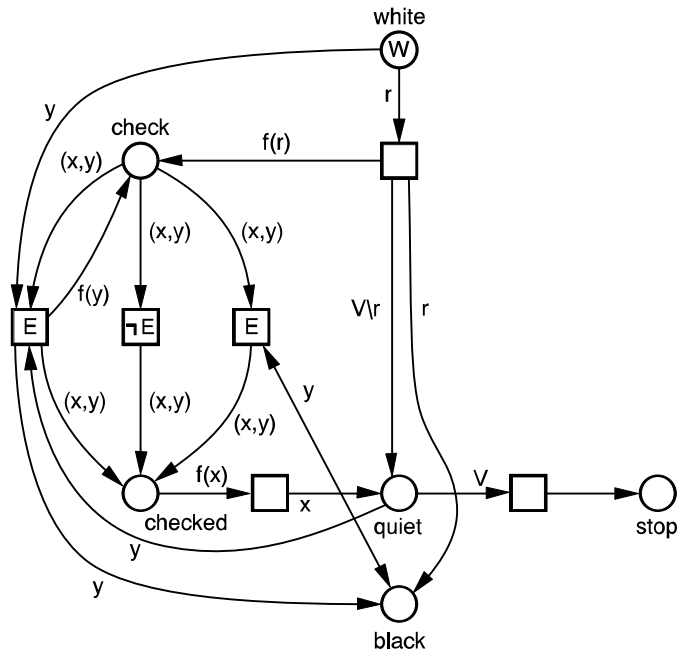


Abb. 7.3. Markierer mit Terminierungserkennung

Jetzt können wir die "Terminiertheit" der Markierungsphase einfach formulieren: sind alle Knoten in *quiet*, dann sind keine Kanten mehr in *check*, d.h. die Markierungsphase ist beendet. Abbildung 7.3 zeigt nochmals die Modellierung der Markierungsphase, jetzt aber mit der Möglichkeit, die Terminierung zu erkennen. Am Anfang werden alle Knoten — bis auf die Wurzel — nach *quiet* gebracht. Jedesmal, wenn der Collector einen neuen Knoten besucht, wird dieser aus dem Prädikat *quiet* entnommen. Im Gegenzug wird ein Knoten wieder "ruhig", wenn alle seine potentiellen Kanten überprüft wurden, d.h. $f(x)$ befindet sich in *checked*. Sobald sich alle Knoten in *quiet* befinden — insbesondere auch die Wurzel — dann ist die Markierungsphase beendet. Abbildung 7.4 zeigt einen Collector, der die Markierungs- und Bereinigungsphase hintereinander ausführt, und zwar

immer wieder. Daß dabei eigentlich gar keine Zeit mehr bleibt, in der der Mutator den Graphen verändern darf, stört uns zwar noch, aber zu Unrecht, da der Mutator fast die ganze Zeit über den Graphen hätte verändern dürfen, wenn Collector und Mutator nebenläufig arbeiten.

7.2 Ein zum Mutator nebenläufiger Collector

Abbildung 7.4 zeigt einen Garbage Collector (N_{COL}), der kontinuierlich markiert und dann bereinigt und dann markiert ... . Während der Collector wie beschrieben arbeitet, darf der Mutator den Graphen verändern, sofern er sich an einige Regeln hält:

- (MS1) Er darf nur dann eine Kante erzeugen, wenn der Zielknoten, auf den die Kante gerichtet werden soll, entweder als "frei" markiert ist, oder von der Wurzel aus erreichbar ist.
- (MS2) Wird diese neue Kante erzeugt, dann muß der Zielknoten danach schwarz sein. Entweder er war bereits schwarz, oder der Mutator muß ihn schwarz machen.
- (MS3) Falls der Zielknoten *y* der neuen Kante vorher weiß war, dann muß der Mutator auch noch alle Kanten, die möglicherweise von *y* wegzeigen, zur Überprüfung nach *check* bringen.
- (MS4) Eine neue Kante kann nur dann erzeugt werden, wenn der Zielknoten vorher "ruhig" oder "schwarz" ist.

Darüberhinaus sind vom Mutator noch einige Bedingungen zu respektieren, die für die Veränderung der Prädikate im Collector gelten. Dazu gehört z.B. die Annahme, daß kein Knoten zugleich frei und schwarz sein darf, und daß sich die Stelle *E* fair verhalten muß (vgl. 3.6.2). Die Forderungen (MS1-MS4) stellen für den Mutator eine in dieser Reihenfolge zunehmende Einschränkung dar. Während (MS1) nur die grundlegende Eigenschaft eines jeden Mutators beschreibt (auch die eines Mutators mit sequentiellem Garbage Collector), formulieren (MS2) und (MS3) die wesentliche Hilfestellung, ohne die ein paralleler Garbage Collector nicht arbeiten könnte (siehe Abschnitt 6.4.1: "Ein nebenläufiger Graph-Markierer ..."). (MS4) dagegen drückt eine Einschränkung des Mutators aus, die in dieser Arbeit gefordert wird, um den Korrektheitsbeweis zu vereinfachen: Der Mutator darf während der Bereinigungsphase des Collectors nur Kanten zu schwarzen Knoten erzeugen. Bedenkt man jedoch, daß die Bereinigungsphase in einem einzigen Schritt ausgeführt werden kann — da alle beteiligten Ereignisse unabhängig voneinander sind — dann scheint diese Einschränkung nicht schwerwiegend zu sein.

Als Nebenbedingung für das Netz in Abbildung 7.4 fordern wir noch, daß freie Knoten keine abgehenden Kanten haben. Das ist eine kleine Problemreduzierung, die das Petrinetz übersichtlicher werden läßt, aber das Problem nicht trivialisiert. Ohne diese Annahme müßte der Collector die abgehenden Kanten eines Knoten löschen, bevor er

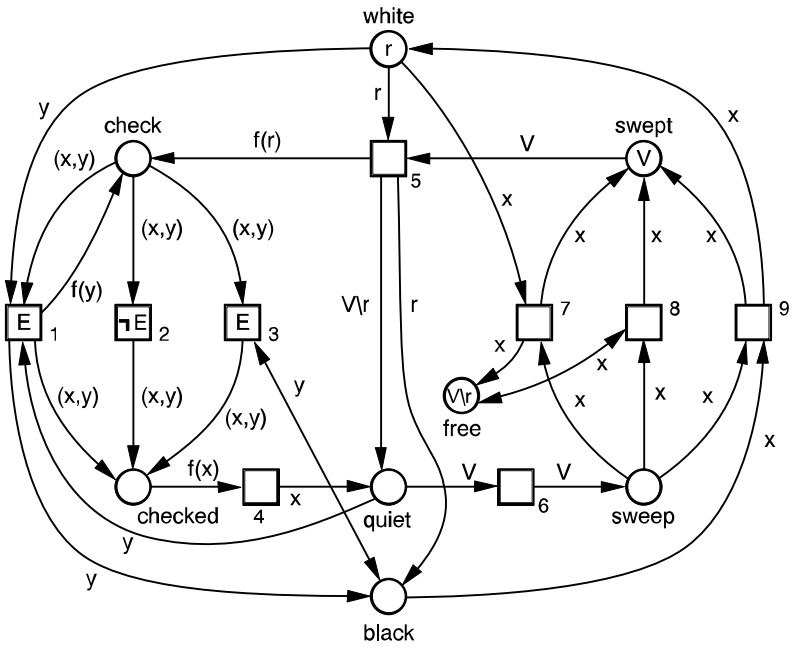


Abb. 7.4. Der Collector N_{COL}

den Knoten als frei markiert. Das würde in dem Netz in Abbildung 7.4 ein paar Stellen und Transitionen mehr kosten, aber zu keinen neuen Erkenntnissen führen.

7.3 Analyse durch S-Invarianten

Bevor wir formal die Korrektheit unserer Implementierung nachweisen, wollen wir anhand klassischer Analysemethoden einige Eigenschaften unseres Netzes darstellen. Als besonders geeignet erweisen sich in diesem Zusammenhang die sog. S-Invarianten, die im zweiten Teil dieser Arbeit bereits formalisiert wurden (vgl. 3.5.5). Eine S-Invariante ist ein Vektor über Multimengen-Terme, der für jede Stelle eine Gewichtung angibt, unter der die Summe aller Stelleninhalte für alle erreichbaren Markierungen konstant bleibt. Zum Nachweis von S-Invarianten werden bekannte Methoden aus der Algebra benutzt: aus dem Petrinetz wird die sog. Inzidenzmatrix abgelesen, die als homogenes lineares Gleichungssystem aufgefasst werden kann. Wenn dabei die transponierte Inzidenzmatrix als Grundlage dient, dann sind die Lösungen dieses Gleichungssystems gerade die S-Invarianten. Abbildung 7.5 zeigt die Inzidenzmatrix, die Startmarkierung $M°$ und die beiden S-Invarianten I_1 und I_2 unseres Garbage Collectors aus Abbildung 7.4.

Um Eigenschaften in unserer Sprache L_{COL} formulieren zu können, müssen wir zeigen, daß N_{COL} 1-sicher ist:

N_{COL}	t_1	t_2 / t_3	t_4	t_5	t_6	t_7	t_8	t_9	M°	I_1	I_2
check	$f(y) - (x,y)$	$-(x,y)$		$f(r)$							s
checked	(x,y)	(x,y)	$-f(x)$								s
quiet	$-y$		x	$V-r$	$-V$						$f(z)$
sweep					V	$-x$	$-x$	$-x$			$f(z)$
swept				$-V$		x	x	x	V		$f(z)$
free						x			$V-r$	z	
white	$-y$			$-r$		$-x$		x	r	z	
black	y			r				$-x$		z	

Abb. 7.5. Inzidenzmatrix zu N_{COL}

7.3.1 Lemma

N_{COL} ist 1-sicher.

Beweis:

N_{COL} ist offensichtlich von nicht-negativen S-Invarianten überdeckt (I_1, I_2 in Abb. - 7.5), d.h. für jede Stelle p existiert eine nicht-negative S-Invariante i mit $i(p) >\hat{\tilde{E}} \tilde{O}$. Für jede Stelle p und der zugehörigen nicht-negativen S-Invariante i existiert ein Mengen-Term u, für den gilt:

$$u \cdot i(p) \equiv\hat{\tilde{E}} M^\circ \cdot i.$$

Die Mengen-Terme lauten: V für *free*, *white*, *black*, *quiet*, *sweep*, und *swept*; F für *check* und *checked*. Darüberhinaus sind $M^\circ \cdot I_1$ und $M^\circ \cdot I_2$ Mengen-Terme. Die Behauptung folgt durch das Theorem 3.5.10. □

7.3.2 Schreibweise

Für $p \in \{check, checked, E\}$ schreiben wir $p[x,y]$ statt $EDGE(x,y).p$.

Aus den S-Invarianten I_1 und I_2 können wir damit die ersten beiden Eigenschaften von N_{COL} folgern:

7.3.3 Lemma

In N_{COL} sind die beiden folgenden Eigenschaften gültig:

(i) **invariant**$_\kappa$ *x.free or x.black or x.white*

(ii) **invariant**$_\kappa$ *x.sweep or x.swept or x.quiet or check*$[x,y]$ *or checked*$[x,y]$

Beweis:

(i) N_{COL} ist 1-sicher und die S-Invariante I_1 überdeckt nur die Stellen *free*, *black* und *white*. Zusammen mit der Startmarkierung $M°$ folgt daher, daß jedes Konstantensymbol $v_1,\ldots,v_n$ der Sorte *vertice* genau einmal in einer der drei Stellen vorkommen muß.

(ii) analog zu (i) mit der S-Invariante I_2; die Behauptung folgt aus der Tatsache, daß u ein Mengenterm ist, wenn $f(u)$ ein Mengenterm ist und es gilt $f(V) = F$. □

TEIL 4

Hierarchische Spezifikation und Korrektheitsbeweis

8 Beweisstruktur

Das Gesamtsystem besteht aus zwei Komponenten, nämlich dem Mutator und dem Collector. Während die Eigenschaften des Garbage Collectors bereits von Anfang an feststehen, und daher auch bereits zu Beginn formalisiert werden können, muß der Mutator nur die Kooperationsbedingungen erfüllen, die nötig sind, um dem parallelen Garbage Collector die Arbeit zu ermöglichen. Die Kooperationsbedingungen werden im Laufe der Spezifikationsverfeinerung herausgearbeitet. Eine Aufstellung dieser Forderungen kann daher erst am Ende der hierarchischen Spezifikation angegeben werden.

Der in den nächsten Kapiteln fomulierte Beweis besitzt verschiedene Strukturmerkmale, deren Kenntnis dem Leser eine große Hilfe sein kann. Die *horizontale Struktur* ergibt sich durch die schrittweise Verfeinerung von Spezifikationen, in dessen Verlauf nacheinander verschiedene Aspekte der Korrektheit behandelt werden. Die *vertikale Struktur* wird durch drei verschiedene Sorten von Formeln eingebracht, die sich in einer bestimmten Weise aufeinander abstützen: Anforderungen, abgeleitete Eigenschaften und Eigenschaften. Diese Strukturierung soll zum einen die Lesbarkeit erhöhen und zum anderen dafür sorgen, daß keine Zyklen in der Argumentation entstehen.

8.1 Horizontale Beweisstruktur

Ausgehend von der Spezifikation des Garbage Collectors (*SPEC1 Collector*), führt eine Kette von 5 Systemspezifikationen (*SPEC1-5 System*) zu den einzelnen Spezifikationen der beiden Teilsysteme (*SPEC Mutator* bzw. *SPEC2 Collector*). Anschließend wird die Petrinetz-Implementierung des parallelen Garbage Collectors anhand von *SPEC2 Collector* verifiziert. Die Spezifikation des Mutators bleibt dagegen “unimplementiert” und stellt gleichsam eine Art Schnittstelle zu einem beliebigen Mutator her, sofern dieser nur *SPEC Mutator* erfüllt. Jedes Petrinetz, das zu dieser Schnittstelle paßt, ist damit ein geeigneter Mutator, mit dem der bereits implementierte Garbage Collector korrekt — im Sinne von *SPEC1 Collector* — zusammenarbeiten kann.

In Abbildung 8.1 wird die horizontale Beweisstruktur graphisch dargestellt. Um dem Attribut “horizontal” gerecht zu werden, müßte die Spezifikationsverfeinerung von links

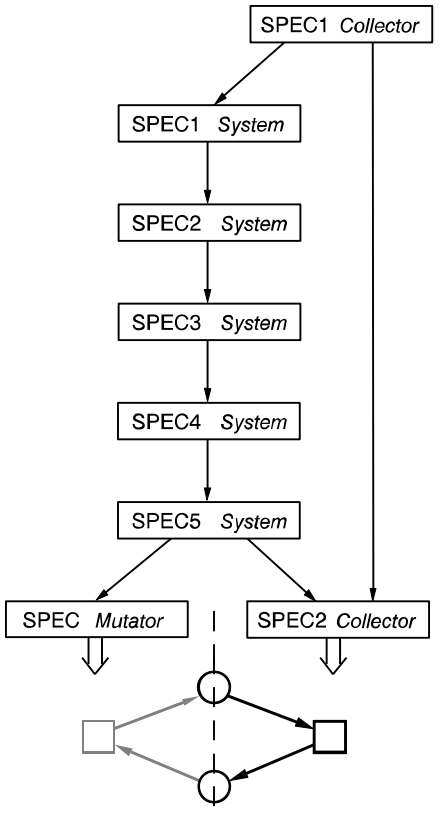


Abb. 8.1. Horizontale Beweisstruktur

nach rechts verlaufen anstelle von oben nach unten, was hier jedoch aus Platzgründen unterbleiben mußte. Die Verbindungspfeile geben dabei das Verhältnis der einzelnen Spezifikationen zueinander an. Eine Spezifikation *SPEC B* entsteht aus einer Spezifikation *SPEC A*, wenn in der Abbildung ein Pfeil von *SPEC A* nach *SPEC B* zeigt. Diese "Pfeilrelation" hat jedoch im Bild nicht immer die gleiche Bedeutung. Ein genauer semantischer Zusammenhang zwischen den einzelnen Spezifikationen wird durch die Halbordnung $\sqsubseteq$ ausgedrückt. Eine Spezifikation *SPEC B* erfüllt *SPEC A*, d.h. *SPEC A* $\sqsubseteq$ *SPEC B*, genau dann, wenn alle Formeln von *SPEC A* aus *SPEC B* herleitbar sind. In der hierarchischen Spezifikation des parallelen Garbage Collectors ergeben sich die folgenden Zusammenhänge:

- *SPEC*1 *Collector* $\sqsubseteq$ *SPEC*2 *Collector*
- *SPEC*1 *System* $\sqsubseteq$ *SPEC*2 *System* $\sqsubseteq$ *SPEC*3 *System* $\sqsubseteq$ *SPEC*4 *System* $\sqsubseteq$ *SPEC*5 *System*
- *SPEC*5*System* $\sqsubseteq$ (*SPEC Mutator* $[\!]$ *SPEC*2 *Collector*)

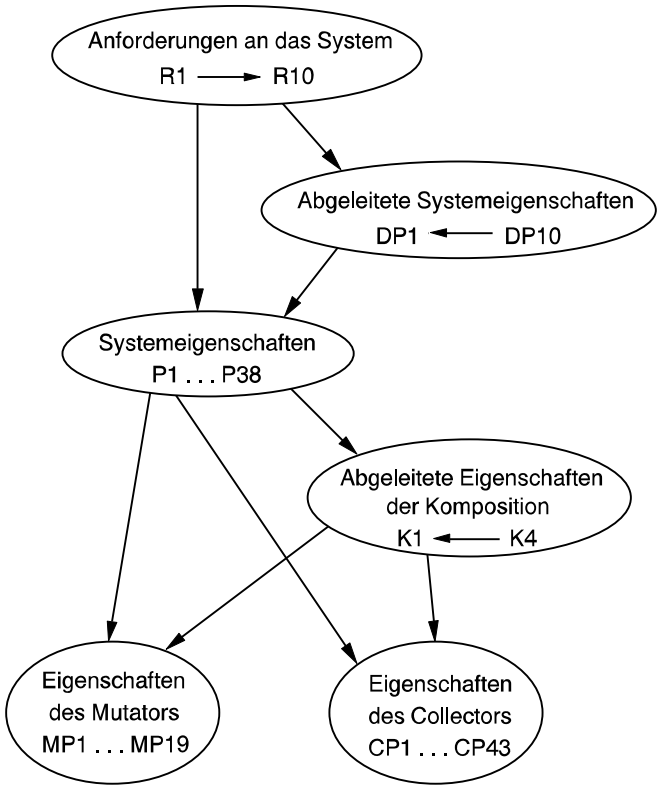


Abb. 8.2. Vertikale Beweisstruktur

Der Operator ∥ bezeichnet dabei die Vereinigung von Spezifikationen zweier Teilsysteme gemäß dem Vereinigungstheorem (vgl. 5.2.2). Die noch offene "Pfeilbeziehung" zwischen *SPEC2 Collector* und *SPEC1 System* in Abbildung 8.1 kann nur informell beschrieben werden: Alle Anforderungen an den Garbage Collector, die bei einer Betrachtung des Gesamtsystems noch sichtbar sind, werden nach *SPEC1 System* übernommen. Alle anderen werden direkt nach *SPEC2 Collector* übertragen.

8.2 Vertikale Beweisstruktur

Zur Verhinderung von Argumentationszyklen können verschiedene Methoden benutzt werden, z.B.:

- In jedem Beweis einer Formel werden nur Formeln benutzt, die später bewiesen werden (oder nur solche, die bereits bewiesen wurden, je nach Beweisrichtung).
- Jede Formel erhält eine Anschrift, aus der ersichtlich ist, aus welchen Formeln sie hergeleitet wurde bzw. aus welchen nicht.

In unserem hierarchischen Beweis wird die zuletzt genannte Methode verwendet. Sie bietet größere Freiheit bei der Argumentation und ist wirksamer bei der Erkennung von Zyklen. Die dazu nötige Information steckt bei geeigneter Reglementierung bereits in der Formelnummer. Abbildung 8.2 zeigt die im Beweis verwendete Reglementierung. Durch Pfeile wird angegeben, wie sich die einzelnen Formeln gegenseitig abstützen können. So kann sich z.B. eine Anforderung R1 auf alle Anforderungen stützen, die eine größere Nummer besitzen, auf alle abgeleiteten Systemeigenschaften und direkt auf Systemeigenschaften. Abgeleitete Systemeigenschaften sind Formeln, die beim Beweis von Anforderungen an verschiedenen Stellen benutzt werden und die deshalb eine eigene Strukturebene erhalten. Das gleiche gilt für die abgeleiteten Eigenschaften der Komposition in Bezug auf die Eigenschaften der Komponenten. Innerhalb der Ebenen für abgeleitete Eigenschaften stützen sich Formeln auf solche mit einer kleineren Nummer. So könnte sich z.B. DP7 auf DP4 und DP1 stützen, nicht aber auf DP10.

Eine Ausnahme dieser Regeln ergibt sich aus dem Zusammenhang zwischen Sicherheits- und Lebendigkeitsanforderungen: Da Sicherheitsanforderungen nicht durch Lebendigkeitsanforderungen bewiesen werden, dürfen zum Beweis einer Lebendigkeitsanforderung auch Sicherheitsanforderungen herangezogen werden, die eine kleinere Formelnummer besitzen.

In der Spezifikationsverfeinerung im nächsten Kapitel kommt es gelegentlich vor, daß Formeln unter zwei Nummern geführt werden, je nachdem, in welchem Kontext sie auftreten. Eine solche Situation tritt immer dann auf, wenn eine Systemanforderung sich wie eine abgeleitete Eigenschaft verhält. Anders ausgedrückt: eine Formel wird zuerst als Anforderung notiert und später als abgeleitete Eigenschaft bewiesen, da sich herausstellt, daß die Formel sich nur auf Eigenschaften stützt und noch öfters benötigt wird.

9 Formale Spezifikation des dynamischen Verhaltens

Im folgenden Abschnitt geben wir die Spezifikation unseres Collectors an. Wir betrachten dabei den Collector als eine Komponente des Gesamtsystems, insbesondere werden wir Eigenschaften vom Collector fordern, die im Gesamtsystem nicht mehr sichtbar sind. Es handelt sich dabei um die Forderungen, daß der Graph i.a. vom Collector nicht verändert werden darf, und daß die "frei"-Markierungen der Knoten vom Collector nicht gelöscht werden dürfen. In dem darauffolgenden Abschnitt spezifizieren wir das Verhalten des Gesamtsystems unter Berücksichtigung der Collector-Spezifikation, d.h. alle Anforderungen an den Collector, die im Gesamtsystem später als Eigenschaften sichtbar werden sollen, übernehmen wir aus der Collector-Spezifikation.

9.1 Spezifikation des parallelen Garbage Collectors

Um das Verhalten eines Garbage Collectors zu beschreiben, reichen im wesentlichen zwei Anforderungen aus:

- *Lebendigkeit:* jede Speicherzelle, die nicht mehr von der Wurzel aus durch einen Pfad erreichbar ist, soll nach endlicher Zeit erkannt und als "frei" markiert werden.
- *Sicherheit:* unter keinen Umständen dürfen Speicherzellen als "frei" markiert werden, wenn sie noch von der Wurzel aus erreichbar sind.

Die Arbeitsgrundlage des Collectors bilden vorerst zwei dynamische Prädikate. Das Prädikat E repräsentiert die aktuelle Menge der existierenden Kanten, während das Prädikat *free* für jeden Knoten bestimmt, ob er frei ist oder nicht. Darüberhinaus benutzen wir abgeleitete Prädikate, wie z.B. R, welches die transitive, reflexive Hülle von E bezeichnet:

9.1.1 Schreibweise

(i) $E[x,y] \equiv$ es gibt eine Kante von x nach y im Graphen.
(ii) $R[x,y] \equiv (x = y) \vee$ es gibt einen Pfad von x nach y im Graphen.
(iii) $x.food \equiv R[r,x]$
(iv) $x.garbage \equiv \neg x.food$

Am Anfang gibt es noch keine Kanten, d.h. $E = \emptyset$, und alle Knoten — bis auf die Wurzel — sind frei. Knoten, die von der Wurzel aus durch einen Pfad erreichbar sind, bezeichnen wir als "eßbar" (oder engl. *food*), alle anderen als "Abfall" (oder engl. *garbage*). Abfallknoten sind entweder in *free* oder nicht in *free*. Auf keinen Fall darf sich ein *food*-Knoten in *free* befinden, und jeder *garbage*-Knoten soll irgendwann nach *free* kommen. Darüberhinaus darf der Collector weder "frei"-Markierungen löschen, noch irgendwelche

Kanten erzeugen. Abgehende Kanten darf er nur dann löschen, wenn der Knoten danach als frei markiert ist. *SPEC1 Collector* faßt diese Vorstellungen zusammen:

9.1.2 Spezifikation *SPEC1 Collector*

Anforderungen

invariant $\neg(x.free \wedge x.food)$ (R1)

$x.garbage \mapsto x.free$ (R2)

Eigenschaften

stable $x.free$ (CP1)

stable $\neg E[x,y]$ (CP2)

stable $E[x,y] \vee x.free$ (CP3)

Die Anforderungen (R1) und (R2) beschreiben das wesentliche Verhalten eines Garbage Collectors, wie wir es auch von unserem Netz N_{COL} erwarten. (CP1-CP3) sind Grundannahmen, ohne die der Collector keinen Sinn hätte. Da sie einerseits trivial sind und andererseits im Gesamtsystem nicht mehr sichtbar sind, werden sie nicht in die Spezifikation des Gesamtsystems aufgenommen, sondern direkt vom Netz gefordert (siehe *SPEC2 Collector* in Abschnitt 11.2.1).

9.2 Spezifikation des Gesamtsystems

Wie bereits erwähnt wurde, ist es unmöglich, einen zum Mutator nebenläufigen Garbage Collector zu entwickeln, der ohne eine Kooperation durch den Mutator auskommt. Aus diesem Grund ist es erforderlich, beim Korrektheitsbeweis des Collectors das gesamte System zu betrachten, d.h. die Komposition von *Collector* und *Mutator* (= *System*). In die Startspezifikation des Gesamtsystems werden die Collectoranforderungen aus *SPEC1 Collector* übernommen. Sie werden im nächsten Kapitel im Zuge der Spezifikationsverfeinerung durch einfachere Formeln ersetzt, die letztendlich direkt an der Implementierung N_{COL} abzulesen sind, oder sich in der Spezifikation des Mutators wiederfinden lassen.

9.2.1 Spezifikation *SPEC1 System*

Anforderungen

invariant $\neg(x.free \wedge x.food)$ (R1)

$x.garbage \mapsto x.free$ (R2)

Eigenschaften

$\neg E[x,y]$ *unless* $\neg E[x,y] \wedge (y.food \vee y.free)$ (P1)

invariant $x.free \Rightarrow \neg E[x,y]$ (P2)

9.2.2 Theorem

Die Anforderungen in *SPEC1 Collector* können aus *SPEC1 System* hergeleitet werden.

Beweis:

(R1) und (R2) sind in *SPEC1 System* enthalten. □

Zur Vereinfachung der im nächsten Kapitel beschriebenen Spezifikationsverfeinerung wird angenommen, daß ein Knoten, der als frei markiert ist, keine abgehenden Kanten aufweist (siehe P2). Zusammen mit der wesentlichen Eigenschaft des Mutators, nämlich Kanten nur auf solche Knoten zu richten, die entweder frei sind, oder von der Wurzel aus erreichbar sind (siehe P1), können bereits 4 grundlegende Eigenschaften des Gesamtsystems abgeleitet werden, auf die später oft Bezug genommen wird:

- Alle Knoten, die von einem eßbaren Knoten aus erreicht werden können, sind selbst eßbar (siehe DP1).
- Alle Knoten, die von einem freien Knoten aus erreicht werden können, sind selbst frei (siehe DP2).
- Zu einem Müllknoten, der nicht als frei markiert ist, kann kein neuer Pfad erzeugt werden (siehe DP3).
- Nur freie Knoten können wieder eßbar gemacht werden (siehe DP4).

9.2.3 Theorem

Die folgenden Eigenschaften können aus (P1) und (P2) abgeleitet werden:

$x.food \wedge R[x,y] \Rightarrow y.food$ (DP1)

invariant $x.free \wedge R[x,y] \Rightarrow y.free$ (DP2)

$\neg R[x,y]$ *unless* $\neg R[x,y] \wedge (y.food \vee y.free)$ (DP3)

$x.garbage$ *unless* $x.garbage \wedge x.free$ (DP4)

Beweis: (DP1) $x.food \wedge R[x,y] \Rightarrow y.food$

$x.food \Rightarrow R[r,x]$
, aus der Definition von *food*

$x.food \wedge R[x,y] \Rightarrow R[r,x] \wedge R[x,y]$
, auf beiden Seiten Konjunktion mit $R[x,y]$

$x.food \wedge R[x,y] \Rightarrow R[r,y]$
, Zusammenfassung der rechten Seite durch die Transitivität von R

$x.food \wedge R[x,y] \Rightarrow y.food$
, rechte Seite durch *food* ersetzen □

Beweis: (DP2) **invariant** $x.free \wedge R[x,y] \Rightarrow y.free$

invariant $z.free \Rightarrow \neg E[z,y]$
, (P2)

invariant $z.free \wedge R[z,x] \Rightarrow \neg E[z,y] \wedge ((x = z) \vee \langle \exists\, y :: E[z,y] \rangle)$
, dazu links $R[z,x]$ und rechts aus der Definition von R

invariant $z.free \wedge R[z,x] \Rightarrow (x = z)$
, Vereinfachung der rechten Seite durch Widerspruch bzgl. $E[z,y]$

invariant $z.free \wedge R[z,x] \Rightarrow x.free$ □

Beweis: (DP3) $\neg R[x,y]$ *unless* $\neg R[x,y] \wedge (y.food \vee y.free)$

Definiere $R^{(k)}[x,y]$ für $k \geq 0$ folgendermaßen:

$R^{(k)}[x,y] \equiv (x = y) \vee$ es gibt einen Pfad von x nach y mit höchstens k Kanten

oder formal:

$R^{(0)}[x,y] \equiv (x = y)$ und

$R^{(k+1)}[x,y] \equiv \langle \exists\, z :: E[x,z] \wedge R^{(k)}[z,y] \rangle \vee R^{(k)}[x,y]$

Durch Induktion über k zeigen wir

$\neg R^{(k)}[x,y]$ *unless* $\neg R^{(k)}[x,y] \wedge (y.food \vee y.free)$

ist gültig für alle k, woraus die Behauptung folgt.

Induktionsanfang:

$\neg R^{(0)}[x,y]$ *unless* $\neg R^{(0)}[x,y] \wedge (y.food \vee y.free)$
, folgt aus der Definition, da $R^{(0)}$ konstant ist

Induktionsschritt:

$\neg R^{(k)}[x,y]$ *unless* $\neg R^{(k)}[x,y] \wedge (y.food \vee y.free)$
, Hypothese für ein k (1)

$\neg E[x,z]$ *unless* $\neg E[x,z] \wedge (z.food \vee z.free)$
, (P1)

$R^{(k)}[z,y] \wedge z.food \Rightarrow y.food$
, aus (DP1) und $R^{(k)}[z,y] \Rightarrow R[z,y]$

invariant $R^{(k)}[z,y] \wedge z.free \Rightarrow y.free$
, aus (DP2) und $R^{(k)}[z,y] \Rightarrow R[z,y]$

$(\neg E[x,z] \vee \neg R^{(k)}[z,y])$ *unless* $(\neg E[x,z] \vee \neg R^{(k)}[z,y]) \wedge (y.food \vee y.free)$
, Disjunktion von (1) und (P1), anschließend Abschwächung der Konsequenz durch die beiden vorhergehenden Formeln

$\langle \forall\ z :: \neg E[x,z] \vee \neg R^{(k)}[z,y] \rangle$ *unless*
$\langle \forall\ z :: \neg E[x,z] \vee \neg R^{(k)}[z,y] \rangle \wedge (y.food \vee y.free)$
, Konjunktion über alle z

$\langle \forall\ z :: \neg E[x,z] \vee \neg R^{(k)}[z,y] \rangle \wedge \neg R^{(k)}[x,y]$ *unless*
$\langle \forall\ z :: \neg E[x,z] \vee \neg R^{(k)}[z,y] \rangle \wedge \neg R^{(k)}[x,y] \wedge (y.food \vee y.free)$
, Konjunktion mit der Hypothese (1)

$\neg R^{(k+1)}[x,y]$ *unless* $\neg R^{(k+1)}[x,y]) \wedge (y.food \vee y.free)$
, Anwendung der Definition von $R^{(k+1)}$

Induktionsschluß:

$\neg R^{(n)}[x,y]$ *unless* $\neg R^{(n)}[x,y]) \wedge (y.food \vee y.free)$
, gültig für alle n, insbesondere $R[x,y] = R^{(n)}[x,y]$, wobei n die Länge des Pfades zwischen x und y ist. □

Beweis: (DP4) $x.garbage$ *unless* $x.garbage \wedge x.free$

$\neg R[r,y]$ *unless* $\neg R[r,y] \wedge (y.food \vee y.free)$
, ersetze x durch r in (DP3)

$y.garbage$ *unless* $y.garbage \wedge y.free$
, Anwendung von $y.garbage \Leftrightarrow \neg R[r,y] \Leftrightarrow \neg y.food$ □

10 Verfeinerung der Systemspezifikation

In den folgenden 4 Spezifikationen (*SPEC2 System* - *SPEC5 System*) werden die beiden zentralen Anforderungen an den Collector durch Systemeigenschaften bewiesen:

- Jeder Abfallknoten wird irgendwann als frei markiert (siehe R2).
- Kein Knoten ist zugleich frei und eßbar (siehe R1).

Die einzelnen Spezifikationen teilen den Beweis in 4 sich aufeinander abstützende Aspekte der Korrektheit auf. Obwohl keiner der 4 Teile einen eigenständigen Beweis darstellt, liegt jeder der 4 Verfeinerungen ein eigenes Thema zugrunde:

- *SPEC2 System* befasst sich mit dem zentralen Lebendigkeitsaspekt eines parallelen Garbage Collectors. Die Anforderung $x.garbage \mapsto x.free$ wird auf den nie endenden Collector-Zyklus abgestützt, der aus Markierungsphase und Bereinigungsphase besteht. Die Formel $true \mapsto x.sweep$ verkörpert diesen Zyklus in Form eines immer wieder gültig werdenden Prädikats. Sie ist eine der neuen Anforderungen, die an *SPEC3 System* übergeben werden.
- In *SPEC3 System* wird der Collectorzyklus aufgebaut. Die Terminierung der beiden Phasen wird auf einfachere Anforderungen abgestützt, z.B.: in der Markierungsphase wird jeder Knoten immer wieder “ruhig”, oder $marking \mapsto x.quiet$.
- *SPEC4 System* kümmert sich um den zentralen Sicherheitsaspekt: kein Knoten ist zugleich frei und eßbar, oder $\neg(x.free \wedge x.food)$. Gleichzeitig wird ein wichtiger Sicherheitsbaustein für die Terminierung der Markierungsphase geliefert: wenn alle Knoten ruhig sind, dann sind alle eßbaren Knoten schwarz, oder $over \Rightarrow T$
- In *SPEC5 System* schließlich werden Lebendigkeitsanforderungen, die in *SPEC2-SPEC4* entstanden sind, durch elementare Eigenschaften der Implementierung N_{COL} abgestützt.

10.1 Nachweis der Lebendigkeit

Ein Knoten, der Müll geworden ist, wird nach endlicher Zeit als frei markiert. Entweder er wird noch im aktuellen Collector-Zyklus erkannt oder spätestens im darauffolgenden. Das Prädikat *blackreach* dient als Kriterium für diesen Sachverhalt. Ein Knoten x wird als *blackreach* bezeichnet, falls es einen schwarzen Knoten gibt, von dem aus x durch einen Pfad erreichbar ist. Müllknoten mit dieser Eigenschaft entstehen immer dann, wenn ein Knoten während der Markierungsphase zu Müll wird, der Collector ihn aber bereits vorher besucht hat und ihn daher als erreichbar, d.h. schwarz, markiert hat. Ein solcher

Müllknoten wird vom Collector im aktuellen Zyklus nicht erkannt. Der Grund: am Ende der Markierungsphase des Collectors sind alle Knoten, die vorher nur *blackreach* waren, inzwischen selbst schwarz markiert. Sie werden daher vom Collector als eßbar betrachtet und in der darauffolgenden Bereinigungsphase nicht als frei markiert.

Ein Müllknoten, den der Collector aus diesem Grund nicht erkannte, wird dann jedoch sicher im nächsten Zyklus erkannt. Wir stützen diese Behauptung auf zwei Annahmen:

- Am Ende der Bereinigungsphase gibt es keine schwarzen Knoten mehr (siehe R3). Es gibt dann auch keine Knoten mehr, die *blackreach* sind.
- Müllknoten, die nicht *blackreach* sind, bleiben es solange, bis sie frei sind (siehe R4).

Zusammen mit der Annahme, daß der Collector immer wieder in die Bereinigungsphase eintritt ($true \mapsto x.sweep$, $x.sweep \mapsto clean$), kann daraus gefolgert werden, daß jeder Müllknoten nach endlicher Zeit erkannt wird. Eine besondere Rolle beim Beweis dieser Anforderung spielt der Proof Lattice 1 in Abbildung 10.1. Für eine Erläuterung des Formalismus von Proof Lattices sei auf Definition 4.4.2 verwiesen.

10.1.1 Schreibweise

(i) $x.blackreach \equiv \langle \exists\, z :: z.black \wedge R[z,x] \rangle$.

(ii) $clean \equiv \langle \forall\, x :: x.swept \rangle$.

10.1.2 Spezifikation *SPEC2 System*

Anforderungen

(R1)

invariant $clean \Rightarrow \neg x.black$ (R3)

$\neg x.blackreach$ *unless* $\neg x.blackreach \wedge (x.free \vee x.food)$ (R4)

$true \mapsto x.sweep$ (R5)

$x.sweep \mapsto clean$ (R6)

$x.white \wedge x.sweep \mapsto x.free \wedge x.swept$ (R7)

Eigenschaften

(P1,P2)

invariant $x.free \text{ or } x.black \text{ or } x.white$ (P3)

$x.black$ *unless* $x.black \wedge x.sweep$ (P4)

10.1.3 Theorem

SPEC1 System kann aus *SPEC2 System* hergeleitet werden, d.h. (R2) ist aus *SPEC2 System* herleitbar.

Beweis: (R2) *x.garbage* $\mapsto$ *x.free*

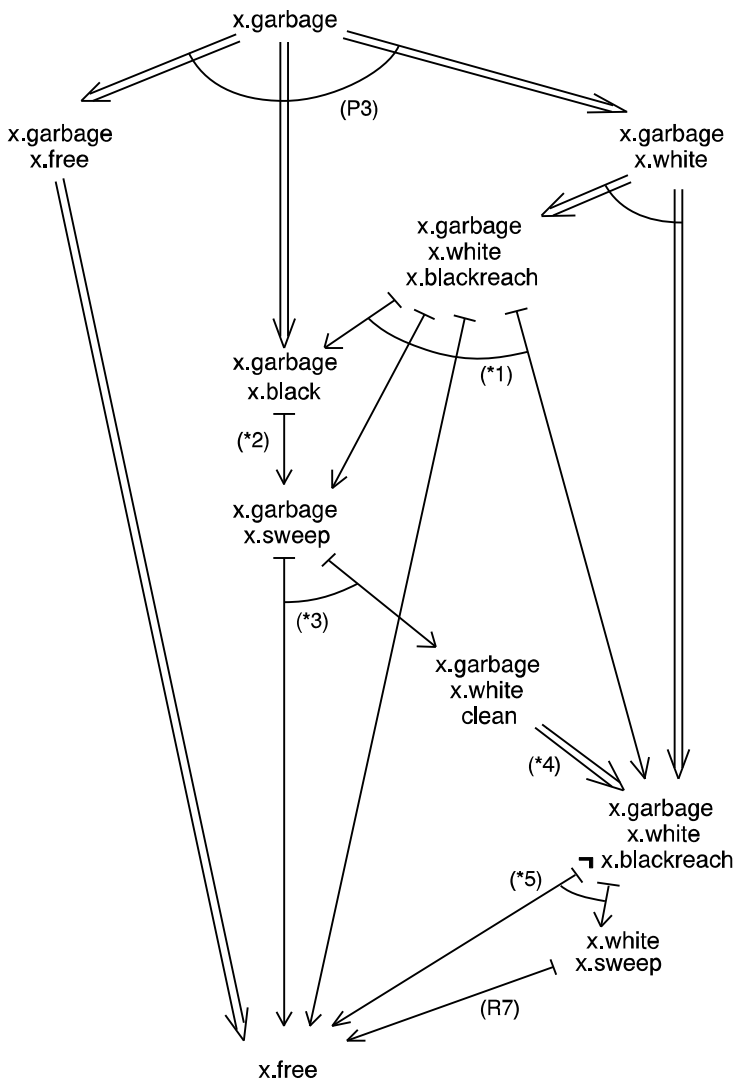


Abb. 10.1. Proof Lattice 1

siehe Proof Lattice 1 in Abbildung 10.1 □

Beweis: (*1) $x.garbage \wedge x.white \wedge x.blackreach \mapsto x.free \vee (x.garbage \wedge (x.black \vee x.sweep \vee (x.white \wedge \neg x.blackreach)))$

$x.white$ *unless* $x.black \vee x.free$
, aus (P3)

$x.blackreach \;\; unless \;\; \neg x.blackreach$
, Antireflexivität von *unless*

$x.white \wedge x.blackreach \;\; unless$
$x.black \vee x.free \vee (x.white \wedge \neg x.blackreach)$
, Konjunktion der vohergehenden Formeln und Abschwächung der Konsequenz

$x.garbage \;\; unless \;\; x.garbage \wedge x.free$
, (DP4)

$x.garbage \wedge x.white \wedge x.blackreach \;\; unless \;\; x.garbage \wedge$
$(x.black \vee x.free \vee (x.white \wedge \neg x.blackreach))$
, Konjunktion der letzten beiden Formeln und Abschwächung der Konsequenz

$true \mapsto x.sweep$
, (R5)

$x.garbage \wedge x.white \wedge x.blackreach \mapsto x.free \vee$
$(x.garbage \wedge (x.black \vee x.sweep \vee (x.white \wedge \neg x.blackreach)))$
, PSP-Theorem auf den letzten beiden Formeln und Abschwächung der Konsequenz □

Beweis: (*2) $x.garbage \wedge x.black \mapsto x.garbage \wedge x.sweep$

$x.black \;\; unless \;\; x.black \wedge x.sweep$
, (P4)

$x.garbage \;\; unless \;\; x.garbage \wedge x.free$
, (DP4)

$x.garbage \wedge x.black \;\; unless \;\; x.garbage \wedge x.black \wedge x.sweep$
, Konjunktion und Verkürzung der rechten Seite durch Widerspruch mit $x.black \Rightarrow \neg x.free$ aus (P3)

$true \mapsto x.sweep$
, (R5)

$x.garbage \wedge x.black \mapsto x.garbage \wedge x.sweep$
, PSP-Theorem und Abschwächung der rechten Seite □

Beweis: (*3) $x.garbage \wedge x.sweep \mapsto x.free \vee (x.garbage \wedge x.white \wedge clean)$

$x.sweep \mapsto clean$
, (R6)

$x.garbage$ *unless* $x.garbage \wedge x.free$
, (DP4)

$x.garbage \wedge x.sweep \mapsto (x.garbage \wedge clean) \vee (x.garbage \wedge x.free)$
, Anwendung des PSP-Theorems auf die letzten beiden Formeln

$x.garbage \wedge x.sweep \mapsto x.free \vee (x.garbage \wedge x.white \wedge clean)$
, Abschwächung der Konsequenz durch $clean \Rightarrow x.free \vee x.white$ aus (R3) und (P3) □

Beweis: (*4) **invariant** $clean \Rightarrow \neg x.blackreach$

invariant $clean \Rightarrow \langle \forall\, y :: \neg y.black \rangle$
, aus (R3) unter Ausnutzung der universellen Allquantifizierung

invariant $clean \Rightarrow \neg x.blackreach$
, aus dem Vorhergehenden und der Definition von *blackreach*: mindestens einen schwarzen Knoten müßte es geben □

Beweis: (*5) $x.garbage \wedge x.white \wedge \neg x.blackreach \mapsto x.free \vee (x.white \wedge x.sweep)$

$\neg x.blackreach$ *unless* $\neg x.blackreach \wedge (x.food \vee x.free)$
, (R4)

$x.garbage$ *unless* $x.garbage \wedge x.free$
, (DP4)

$x.garbage \wedge \neg x.blackreach$ *unless* $x.garbage \wedge \neg x.blackreach \wedge x.free$
, Konjunktion der letzten beiden Formeln und Verkürzung der rechten Seite durch Widerspruch mit $x.garbage \Leftrightarrow \neg x.food$

$x.white$ *unless* $x.black \vee x.free$
, aus (P3) abgeleitet

$x.garbage \wedge x.white \wedge \neg x.blackreach$ *unless* $x.free$
, Konjunktion der letzten beiden Formeln mit Abschwächung der Konsequenz sowie Verkürzung durch Widerspruch mit $x.black \Rightarrow x.blackreach$

$true \mapsto x.sweep$
, (R5)

$x.garbage \wedge x.white \wedge \neg x.blackreach \mapsto x.free \vee (x.white \wedge x.sweep)$
, PSP-Theorem auf den letzten beiden Formeln und Abschwächung der Konsequenz □

10.2 Der Collector-Zyklus als Grundlage der Lebendigkeit

In der folgenden Spezifikation werden die restlichen Prädikate aus der Implementierung N_{COL} eingeführt: *check*, *checked* und *quiet*. Mit ihrer Hilfe können die beiden Phasen des Collector-Zyklus vollständig beschrieben werden:

- Der Collector befindet sich in der Markierungsphase (*marking*), wenn für irgendeinen Knoten oder dessen abgehende Kanten eines der Prädikate *check*, *checked* oder *quiet* gilt.
- Er befindet sich in der Bereinigungsphase (*sweeping*), wenn für einen der Knoten eines der Prädikate *sweep* oder *swept* gilt.

Daß daraus ein sinnvoller Zyklus entsteht, auf den wir unsere Lebendigkeitsanforderungen aus der letzten Spezifikation stützen können, dafür sorgen zwei Annahmen:

- Zu jedem Zeitpunkt befindet sich der Collector nur in einer der beiden Phasen (siehe P5).
- Für einen Knoten kann zu keinem Zeitpunkt mehr als eines der Prädikate *sweep* oder *swept* gelten (siehe P9). Analoges gilt auch für die Prädikate *check*, *checked* und *quiet*; diese Eigenschaft wird jedoch erst in der nächsten Spezifikation ausgenützt.

Schließlich bleibt noch zu zeigen, daß die beiden Phasen tatsächlich terminieren und an ihrem Ende jeweils in die andere Phase münden. Die graphische Herleitung dieser Eigenschaft zeigt Proof Lattice 2 in Abbildung 10.2. Er stützt sich dabei u.a. auf drei neue Anforderungen:

- Während der Markierungsphase wird jeder Knoten immer wieder “ruhig” (*marking* $\mapsto$ *x.quiet*, siehe R10). Diese Forderung beinhaltet nicht, daß Knoten in *quiet* auch in *quiet* bleiben.
- Wenn alle Knoten “ruhig” sind, dann beginnt die Bereinigungsphase (*over* $\mapsto$ *x.sweep*, siehe R8).
- Innerhalb der Bereinigungsphase wird jeder Knoten nach endlicher Zeit bereinigt (*x.sweep* $\mapsto$ *x.swept*, siehe R9).

10.2.1 Schreibweise

(i) $over \equiv \langle \forall\, x :: x.quiet \rangle$.
(ii) $marking \equiv \langle \exists\, x,y :: x.quiet \vee check[x,y] \vee checked[x,y] \rangle$.
(iii) $sweeping \equiv \langle \exists\, x :: x.sweep \vee x.swept \rangle$.

10.2.2 Spezifikation *SPEC3 System*

Anforderungen

(R1,R7)

$over \mapsto x.sweep$ (R8)

$x.sweep \mapsto x.swept$ (R9)

$marking \mapsto x.quiet$ (R10)

Eigenschaften

(P1-P4)

invariant $marking \ or \ sweeping$ (P5)

$x.swept \ \ unless \ \ clean$ (P6)

$x.quiet \ \ unless \ \ x.black \vee x.sweep$ (P7)

$x.quiet \ \ unless \ \ (x.quiet \wedge x.white) \vee over$ (P8)

invariant $sweeping \Rightarrow x.sweep \ or \ x.swept$ (P9)

$\neg check[x,y] \ \ unless \ \ check[x,y] \wedge x.black$ (P10)

$\neg x.black \ \ unless \ \ \neg x.black \wedge$
$\quad (x.free \vee x.food \vee \langle \exists \, z :: check[z,x] \wedge E[z,x] \rangle)$ (P11)

$clean \wedge r.white \ \ ensures \ \ marking \vee r.black$ (P12)

$\neg x.swept \ \ unless \ \ x.swept \wedge (x.white \vee x.free)$ (P13)

$marking \ \ unless \ \ marking \wedge over$ (P14)

$\neg x.black \ \ unless \ \ marking$ (P15)

10.2.3 Theorem

Aus (P1-P15) können die folgenden Eigenschaften abgeleitet werden:

invariant $clean \Rightarrow \neg x.black$ (DP5)

invariant $check[x,y] \Rightarrow x.black$ (DP6)

$\neg x.blackreach \ \ unless \ \ \neg x.blackreach \wedge (x.free \vee x.food)$ (DP7)

10.2.4 Theorem

SPEC2 System kann aus *SPEC3 System* hergeleitet werden, d.h. (R3-R6) sind aus *SPEC3 System* herleitbar.

Beweis: (DP5,R3) **invariant** $clean \Rightarrow \neg x.black$

$\neg x.black \ \ unless \ \ marking$
, (P15)

$\neg x.swept \ \ unless \ \ x.swept \wedge (x.white \vee x.free)$
, (P13)

$\neg x.swept \;\vee\; \neg x.black \;\; unless \;\; false$
, Disjunktion der beiden letzten Formeln und Verkürzung der rechten Seite durch Widerspruch mit $x.black \Rightarrow (\neg x.white \wedge \neg x.free)$ aus (P3) und $x.swept \Rightarrow sweeping \Rightarrow \neg marking$ aus (P5)

initially $\neg x.swept \;\vee\; \neg x.black$
, folgt aus der Startmarkierung von N_{COL}

invariant $clean \;\Rightarrow\; \neg x.black$
, aus den letzten beiden Formeln kann die Invariante $\neg x.swept \vee \neg x.black$ abgeleitet werden, die zusammen mit der Definition von *clean* die Behauptung ergibt □

Beweis: (DP6) **invariant** $check[x,y] \;\Rightarrow\; x.black$

$x.black \;\; unless \;\; x.black \wedge x.sweep$
, (P4)

$\neg check[x,y] \;\; unless \;\; check[x,y] \wedge x.black$
, (P10)

$\neg check[x,y] \;\vee\; x.black \;\; unless \;\; false$
, Disjunktion der beiden vorhergehenden Formeln und Verkürzung der rechten Seite durch Widerspruch mit $x.sweep \Rightarrow sweeping \Rightarrow \neg marking \Rightarrow \neg check[x,y]$ aus (P5)

initially $\neg check[x,y] \;\vee\; x.black$
, aus der Startmarkierung von N_{COL}

invariant $check[x,y] \;\Rightarrow\; x.black$
, aus den letzten beiden Formeln □

Beweis: (DP7,R4) $\neg x.blackreach \;\; unless \;\; \neg x.blackreach \wedge (x.free \vee x.food)$

invariant $check[z,x] \;\Rightarrow\; z.black$
, (DP6)

$\neg x.black \;\; unless \;\; \neg x.black \wedge$
$(x.free \vee x.food \vee \langle \exists\, z :: check[z,x] \wedge E[z,x] \rangle)$
, (P11)

$\neg x.black \;\; unless \;\; \neg x.black \wedge$
$(x.free \vee x.food \vee \langle \exists\, z :: z.black \wedge R[z,x] \rangle)$
, Abschwächung der Konsequenz der letzten Formel durch (1) sowie $E[z,x] \Rightarrow R[z,x]$

$\neg z.black \;\; unless \;\; \neg z.black \wedge (z.free \vee z.food \vee z.blackreach)$
, aus der letzten Formel und der Definition von *blackreach*

$\neg R[z,x] \;\; unless \;\; \neg R[z,x] \wedge (x.food \vee x.free)$
, (DP3)

$\neg z.black \vee \neg R[z,x] \;\; unless$
$(\neg z.black \vee \neg R[z,x]) \wedge (x.free \vee x.food \vee x.blackreach)$
, Disjunktion und Abschwächung der Konsequenz durch $z.blackreach \wedge R[z,x] \Rightarrow x.blackreach$ (aus der Definition von *blackreach*) sowie (DP1): $R[z,x] \wedge z.food \Rightarrow x.food$ und (DP2): $R[z,x] \wedge z.free \Rightarrow x.free$

$\langle \forall z :: \neg z.black \vee \neg R[z,x] \rangle \;\; unless$
$\langle \forall z :: \neg z.black \vee \neg R[z,x] \rangle \wedge (x.free \vee x.food \vee x.blackreach)$
, Konjunktion der letzten Formel über alle z

$\neg x.blackreach \;\; unless \;\; \neg x.blackreach \wedge (x.food \vee x.free)$
, Anwendung der Definition von *blackreach* □

Beweis: (R5) $true \mapsto x.sweep$

siehe Proof Lattice 2 in Abbildung 10.2 □

Beweis: (R6) $x.sweep \mapsto clean$

siehe Proof Lattice 2 in Abbildung 10.2 □

Beweis: (*1) $sweeping \mapsto clean$

$x.swept \mapsto x.swept$
, Reflexivität von $\mapsto$

$x.sweep \mapsto x.swept$
, (R9)

$x.sweep \vee x.swept \mapsto x.swept$
, einfache Disjunktion der beiden letzten Formeln

$sweeping \mapsto x.sweep \vee x.swept$
, aus (P9) und Anwendung des Implikationstheorems

$sweeping \mapsto x.swept \vee clean$
, aus den beiden letzten Formeln und der Transitivität von $\mapsto$ sowie anschließender Abschwächung der Konsequenz um den Term *clean*

$x.swept \;\; unless \;\; clean$
, (P6)

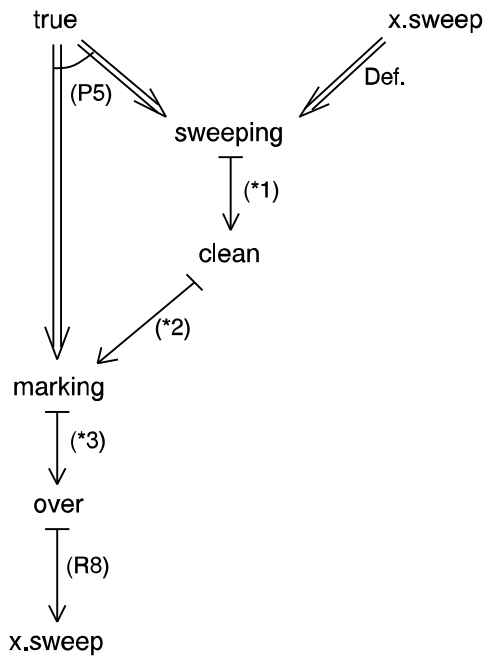


Abb. 10.2. Proof Lattice 2

sweeping $\mapsto \langle \forall\, x :: x.swept \rangle \vee clean$

, Anwendung des Vervollständigungstheorems auf die beiden vohergehenden Formeln; durch Einsetzen der Definition von *clean* ergibt sich die Behauptung □

Beweis: (*2) *clean* $\mapsto$ *marking*

invariant $\neg r.free$

, ersetze *x* durch *r* in (R1) und benutze **invariant** *r.food* aus der Definition von *food*

invariant $clean \Rightarrow x.white \vee x.free$

, aus (R3) und $\neg x.black \Rightarrow x.white \vee x.free$ aus (P3)

invariant $clean \Rightarrow r.white$

, ersetze in der letzten Formel *x* durch *r* und benutze die vorletzte Formel zur Verkürzung der Konsequenz

$clean \mapsto clean \wedge r.white \wedge \neg r.black$

, aus der letzten Formel durch Verstärkung der Konsequenz mit *clean* und $x.white \Rightarrow \neg x.black$ aus (P3), sowie Anwendung des Implikationstheorems (1)

$clean \wedge r.white \mapsto marking \vee r.black$
, aus (P12) und der Definition von $\mapsto$

$\neg r.black \ \ unless \ \ marking$
, ersetze x durch r in (P15)

$clean \wedge r.white \wedge \neg r.black \mapsto marking$
, PSP-Theorem auf den beiden letzten Formeln

$clean \mapsto marking$
, aus (1) und der vorhergehenden Formel durch die Transitivität von $\mapsto$ □

Beweis: (*3) $marking \mapsto over$

Definiere die Mengen c und d folgendermaßen:

$c = \{x \mid x.black\}$

$d = \{x \mid x.quiet\}$

Wir zeigen, daß für alle $C \subseteq c$ und alle $D \subseteq d$ die folgende Formel gilt:

$$marking \wedge c = C \wedge d = D \wedge d \neq V \mapsto marking \wedge (c \supset C \vee (c = C \wedge d \supset D) \vee d = V) \quad (1)$$

Durch Induktion über die lexikographische Metrik $(|c|,|d|)$, die nach oben durch $(|V|,|V|)$ begrenzt wird, können wir aus (1) das folgende ableiten:

$marking \mapsto marking \wedge (d = V)$

woraus die Behauptung folgt, da $over \Leftrightarrow (d = V)$.

$x \notin C \ \ unless \ \ false$
, da C konstant ist

$\neg x.black \wedge c = C \ \ unless \ \ x.black \vee c \neq C$
, Antireflexivität von *unless*

$x \notin C \wedge \neg x.black \wedge c = C \ \ unless \ \ x \notin C \wedge (x.black \vee c \neq C)$
, Konjunktion der beiden letzten Formeln

$\neg x.black \wedge c = C \ \ unless \ \ c \neq C$
, Vereinfachung beider Seiten durch $x.black \Leftrightarrow x \in c$ (2)

$x.black \ \ unless \ \ x.black \wedge \neg marking$
, aus (P4) durch Abschwächung der Konsequenz mit $x.sweep \Rightarrow sweeping \Rightarrow \neg marking$ aus (P5)

$marking \ \ unless \ \ marking \wedge (d = V)$
, aus (P14) und $over \Leftrightarrow (d = V)$

$marking \wedge x.black \quad unless \quad marking \wedge (d = V)$
, Konjunktion der beiden letzten Formeln und Abschwächung der Konsequenz

$marking \wedge c = C \quad unless \quad marking \wedge (c \supset C \vee d = V)$
, aus der letzten Formel und der Definition von c (3)

$x.quiet \quad unless \quad x.black \vee x.sweep$
, (P7)

$marking \quad unless \quad marking \wedge (d = V)$
, aus (P14) und $over \Leftrightarrow (d = V)$

$marking \wedge x.quiet \ unless \ x.black \vee (d = V)$
, Konjunktion, anschließend Abschwächung der Konsequenz und Vereinfachung durch Widerspruch mit $x.sweep \Rightarrow sweeping \Rightarrow \neg marking$ aus (P5)

$x.quiet \quad unless \quad x.black \vee (d = V)$
, Vereinfachung der linken Seite durch $x.quiet \Rightarrow marking$

$x.quiet \wedge c = C \wedge \neg x.black \quad unless \quad c \neq C \vee d = V$
, Konjunktion der letzten Formel mit (2) und Abschwächung der Konsequenz (4)

$x.quiet \quad unless \quad (x.quiet \wedge x.white) \vee (d = V)$
, aus (P8) und $over \Leftrightarrow (d = V)$

$x.black \quad unless \quad x.black \wedge x.sweep$
, (P4)

$x.quiet \wedge x.black \quad unless \quad (d = V)$
, Konjunktion der beiden letzten Formeln und Vereinfachung durch Widerspruch mit $x.quiet \Rightarrow marking \Rightarrow \neg sweeping \Rightarrow \neg x.sweep$ aus (P5), sowie $x.black \Rightarrow \neg x.white$ aus (P3)

$c = C \quad unless \quad c \neq C$
, Antireflexivität von *unless*

$x.quiet \wedge c = C \wedge x.black \quad unless \quad c \neq C \vee d = V$
, einfache Konjunktion der letzten beiden Formeln

$x \in d \wedge c = C \quad unless \quad c \neq C \vee d = V$
, einfache Disjunktion der letzten Formel mit (4) und Ersetzung von $x.quiet$ durch $x \in d$

$c = C \wedge d \supseteq D \quad unless \quad c \neq C \vee d = V$
, einfache Konjunktion der letzten Formel über alle $x \in D$

$d = D \quad unless \quad d \neq D$
, Antireflexivität von *unless*

$c = C \ \wedge \ d = D \quad unless \quad c \neq C \ \vee \ d \supset D \ \vee \ d = V$
, Konjunktion der beiden letzten Formeln und Abschwächung der Konsequenz

$marking \ \wedge \ c = C \quad unless \quad marking \ \wedge \ (c \supset C \ \vee \ d = V)$
, (3)

$marking \ \wedge \ c = C \ \wedge \ d = D \quad unless$
$\quad marking \ \wedge \ (c \supset C \ \vee \ (c = C \ \wedge \ d \supset D) \ \vee \ d = V)$
, Konjunktion der letzten beiden Formeln und Abschwächung der Konsequenz (5)

$marking \ \mapsto \ x.quiet \ \mapsto \ x \in d$
, ersetze $x.quiet$ durch $x \in d$ in (R10)

$d = D \ \wedge \ d \neq V \ \wedge \ x \notin D \ \wedge \ marking \ \mapsto \ x \in d$
, Verstärkung der linken Seite

$x \notin D \quad unless \quad false$
, D ist konstant

$d = D \ \wedge \ d \neq V \ \wedge \ x \notin D \ \wedge \ marking \ \mapsto \ d \neq D$
, PSP-Theorem auf den beiden letzten Formeln und Abschwächung der Konsequenz durch $x \in d \wedge x \notin D \Rightarrow d \neq D$

$d = D \ \wedge \ d \neq V \ \wedge \ \langle \exists \, x \ :: \ x \notin D \rangle \ \wedge \ marking \ \mapsto \ d \neq D$
, Disjunktion der letzten Formel über alle $x \notin D$

$d = D \ \wedge \ d \neq V \ \wedge \ marking \ \mapsto \ d \neq D$
, Kürzung der linken Seite durch $d = D \wedge d \neq V \ \Rightarrow \ \langle \exists \, x :: x \notin D \rangle$

$marking \ \wedge \ c = C \ \wedge \ d = D \ \wedge \ d \neq V \ \mapsto$
$\quad marking \ \wedge \ (c \supset C \ \vee \ (c = C \ \wedge \ d \supset D) \ \vee \ d = V)$
, PSP-Theorem zwischen der letzten Formel und (5) □

10.3 Nachweis der Sicherheit

In *SPEC4 System* beschäftigen wir uns mit dem wesentlichen Sicherheitsaspekt eines Garbage Collectors. Es wurde gefordert, daß zu keinem Zeitpunkt ein Knoten zugleich eßbar und frei sein darf ($\neg(x.free \wedge \ x.food)$, siehe R1). Wir werden zeigen, daß diese Forderung erfüllt ist, wenn unsere Implementierung u.a. die folgenden Eigenschaften aufweist:

- Am Ende der Markierungsphase sind alle eßbaren Knoten entweder schwarz oder frei, aber auf keinen Fall weiß (*over* $\Rightarrow T$, siehe DP8).
- In der Bereinigungsphase werden ausschließlich weiße Knoten als frei markiert (siehe P22).

Daraus kann gefolgert werden, daß nur solche Knoten als frei markiert werden, die wirklich Müll sind. Das Prädikat T ist dabei der Indikator für eine korrekt abgeschlossene Markierungsphase. T gilt g.d.w.

- der Wurzelknoten ist nicht weiß
- schwarze oder freie Knoten erreichen nur Knoten, die ebenfalls nicht weiß sind.

Insbesondere sind dann alle eßbaren Knoten, d.h. solche, die von der Wurzel aus durch einen Pfad erreichbar sind, schwarz oder frei.

Eine der Eigenschaften, die die Sicherheit des Collectors gewährleistet, nämlich *over* $\Rightarrow T$, ist gleichzeitig auch noch ein wichtiger Baustein beim Beweis der Terminierung der Markierungsphase: mit ihr wird die Anforderung *over* $\mapsto$ *x.sweep* abgestützt (siehe Beweis von R8), die im wesentlichen davon abhängt, daß die Markierungsphase wirklich beendet ist, wenn alle Knoten einmal ruhig sind, d.h. es gilt *over* und damit T. Vor allem wird dann kein Knoten mehr aus *quiet* entfernt, bis die Bereinigungsphase beginnt. Im einzelnen heißt das:

- Der Mutator richtet keine Kanten mehr auf weiße Knoten, da alle Knoten, zu denen er Kanten errichten darf, entweder schwarz oder frei sind.
- Die Markierungsphase des Collectors ist beendet, da keine Kanten mehr überprüft werden müssen, d.h. *check* ist leer.

10.3.1 Schreibweise

$$T \equiv \neg r.white \wedge \langle \forall\, x,y :: \neg x.white \wedge E[x,y] \Rightarrow \neg y.white \rangle$$

10.3.2 Spezifikation *SPEC4 System*

Anforderungen
(R7,R9,R10)

Eigenschaften
(P1-P15)

$\neg marking \;\; unless \;\; marking \wedge r.black \wedge (x.quiet \vee x.black)$ (P16)

invariant $marking \Rightarrow x.quiet \;\; or \;\; check[x,y] \;\; or \;\; checked[x,y]$ (P17)

$x.quiet \;\; unless \;\; \langle \forall\, z :: z.sweep \rangle \vee$
$\quad (x.quiet \wedge x.white \wedge (\langle \exists\, y :: check[y,x] \rangle \vee x.food))$ (P18)

$check[x,y] \;\; unless \;\; checked[x,y] \wedge (\neg E[x,y] \vee y.black)$ (P19)

$\neg x.sweep \;\; unless \;\; \neg x.sweep \wedge over$ (P20)

$\neg x.white \;\; unless \;\; x.white \wedge x.swept$ (P21)
$\neg x.free \;\; unless \;\; \neg x.free \wedge x.white \wedge x.sweep$ (P22)
$x.sweep \;\; unless \;\; x.swept$ (P23)
$\neg E[x,y] \;\; unless \;\; E[x,y] \wedge y.black$ (P24)
$x.white \;\; unless \;\; (x.free \wedge x.swept) \vee (x.black \wedge check[x,y])$ (P25)
$x.garbage \;\; unless \;\; x.food \wedge \neg x.free$ (P26)
$over \wedge \langle \forall\, x :: \neg x.sweep \rangle \;\; ensures \;\; x.sweep \vee \langle \exists\, z,y :: check[z,y] \rangle$ (P27)

10.3.3 Theorem

Aus (P1-P27) können die folgenden Eigenschaften abgeleitet werden:
invariant $over \Rightarrow T$ (DP8)
invariant $\neg(x.free \wedge x.food)$ (DP9)

10.3.4 Theorem

SPEC3 System kann aus *SPEC4 System* hergeleitet werden, d.h. (R1) und (R8) sind aus *SPEC4 System* herleitbar.

Beweis: (DP8) $over \Rightarrow T$

$\neg y.white \;\; unless \;\; y.white \wedge y.swept$
, (P21)

$check[x,y] \;\; unless \;\; checked[x,y] \wedge (\neg E[x,y] \vee y.black)$
, (P19)

$\neg y.white \vee check[x,y] \;\; unless \;\; y.white \wedge (\neg E[x,y] \vee y.swept)$
, Disjunktion mit Abschwächung der Konsequenz sowie Verkürzung durch Widerspruch mit $y.white \Rightarrow \neg y.black$ aus (P3)

$\neg E[x,y] \;\; unless \;\; y.black$
, aus (P24) durch Abschwächung der Konsequenz

$\neg E[x,y] \vee \neg y.white \vee check[x,y] \;\; unless \;\; E[x,y] \wedge y.swept$
, Disjunktion der vorhergehenden Formeln und Vereinfachung der Konsequenz durch Widerspruch mit $y.white \Rightarrow \neg y.black$ aus (P3) (1)

$x.free \;\; unless \;\; \neg E[x,y]$
, aus (P2) und dem Implikationstheorem

$x.white \;\; unless \;\; (x.black \wedge check[x,y]) \vee x.free$
, aus (P25) und Abschwächung der Konsequenz

$\neg x.black \;\; unless \;\; (x.black \wedge check[x,y]) \vee \neg E[x,y]$
, Disjunktion der beiden letzten Formeln und Ersetzen von $x.white \vee x.free$ durch $\neg x.black$ aus (P3), sowie Abschwächung der Konsequenz

$\neg x.black \vee \neg E[x,y] \vee \neg y.white \vee check[x,y] \;\; unless \;\; x.black \wedge y.swept$
, Disjunktion der letzten Formel mit (1) und Abschwächung der Konsequenz (2)

$x.sweep \;\; unless \;\; x.swept$
, (P23)

$x.swept \;\; unless \;\; clean$
, (P6)

$x.swept \vee x.sweep \;\; unless \;\; clean$
, Löschungstheorem auf den beiden letzten Formeln

$\langle \exists\, x :: x.swept \vee x.sweep \rangle \;\; unless \;\; clean$
, einfache Disjunktion der letzten Formel über alle x

$sweeping \;\; unless \;\; \neg x.black$
, Einsetzen der Definition von *sweeping* auf der rechten Seite und Abschwächung der Konsequenz durch $clean \Rightarrow \neg x.black$ (DP5)

$sweeping \vee \neg x.black \vee \neg E[x,y] \vee \neg y.white \vee check[x,y] \;\; unless \;\; false$
, Disjunktion der letzten Formel mit (2) und Vereinfachung der rechten Seite durch Widerspruch mit $y.swept \Rightarrow sweeping$

initially $sweeping \vee \neg x.black \vee \neg E[x,y] \vee \neg y.white \vee check[x,y]$
, aus der Startmarkierung von N_{COL}

invariant $x.black \wedge E[x,y] \wedge y.white \Rightarrow check[x,y] \vee sweeping$
, aus den letzten beiden Formeln und der Definition einer Invariante (3)

$\neg marking \;\; unless \;\; marking \wedge r.black$
, aus (P16) durch Abschwächung der Konsequenz

$r.black \;\; unless \;\; r.black \wedge r.sweep$
, (P4) mit x durch r ersetzt

$\neg marking \vee r.black \;\; unless \;\; false$
, Disjunktion und Vereinfachung der Konsequenz durch Widerspruch mit $r.sweep \Rightarrow sweeping \Rightarrow \neg marking$ aus (P5)

initially $\neg marking \vee r.black$
, aus der Startmarkierung von N_{COL}

invariant $marking \Rightarrow \neg r.white$
, aus der Definition einer Invariante und $r.black \Rightarrow \neg r.white$ aus (P3) (4)

invariant $x.quiet \Rightarrow \neg check[x,y] \wedge \neg sweeping$
, mit $x.quiet \Rightarrow marking \Rightarrow \neg sweeping$ aus (P5), sowie $marking \wedge x.quiet \Rightarrow \neg check[x,y]$ aus (P17)

invariant $x.quiet \Rightarrow \neg(x.black \wedge E[x,y] \wedge y.white)$
, aus (3) durch Widerspruch mit der vorhergehenden Formel

invariant $x.quiet \Rightarrow (\neg x.white \wedge E[x,y] \Rightarrow \neg y.white \vee x.free)$
, aus der vorhergehenden Formel durch Umformung der rechten Seite mit $\neg x.black \Rightarrow x.free \vee x.white$ aus (P3)

invariant $x.quiet \Rightarrow \langle \forall\, y :: \neg x.white \wedge E[x,y] \Rightarrow \neg y.white \rangle$
, Widerspruch mit $E[x,y] \Rightarrow \neg x.free$ aus (P2); Ausnutzung der universellen Allquantifizierung bei y

invariant $over \Rightarrow T$
, Konjunktion der letzten Formel über alle x und Anwendung der Definition von *over*. Zusätzlich gilt $over \Rightarrow marking \Rightarrow \neg r.white$ aus (4) □

Beweis: (R8) $over \mapsto x.sweep$

$x.quiet \;\; unless \;\; \langle \forall\, z :: z.sweep \rangle \vee$
$(x.quiet \wedge x.white \wedge (\langle \exists\, y :: check[y,x] \rangle \vee x.food))$
, (P18)

$over \;\; unless \;\; \langle \forall\, z :: z.sweep \rangle \vee$
$(over \wedge \langle \exists\, x,y :: x.white \wedge (check[y,x] \vee x.food) \rangle)$
, Konjunktion der letzten Formel über alle x und Abschwächung der Konsequenz

$T \wedge x.food \Rightarrow \neg x.white$
, aus der Definition von T und *food*

$over \;\; unless \;\; \langle \forall\, z :: z.sweep \rangle$
, Vereinfachung der vorletzten Formel durch Widerspruch mit $over \Rightarrow y.quiet \Rightarrow \neg check[y,x]$ aus (P17), sowie mit der letzten Formel und $over \Rightarrow T$ (DP8) (1)

$over \wedge \langle \forall\, x :: \neg x.sweep \rangle \;\; ensures \;\; x.sweep \vee \langle \exists\, z,y :: check[z,y] \rangle$
, (P27)

$over \mapsto x.sweep \vee \langle \exists\, z,y :: check[z,y] \rangle$
, aus der letzten Formel mit der Definition von $\mapsto$ und $over \Rightarrow marking \Rightarrow \neg sweeping \Rightarrow \langle \forall x :: \neg x.sweep \rangle$ aus (P5)

$over \mapsto \langle \forall\ z\ ::\ z.sweep \rangle$
, PSP-Theorem mit (1) und Vereinfachung der Konsequenz durch Widerspruch mit $over \Rightarrow marking \Rightarrow \neg sweeping \Rightarrow \neg x.sweep$ aus (P5), sowie $over \Rightarrow \neg check[z,y]$ aus (P17)

$over \mapsto z.sweep$
, aus der letzten Formel durch Verschieben der Allquantifizierung nach links □

Beweis: (DP9,R1) **invariant** $\neg(x.free \wedge x.food)$

$\neg x.sweep \ \ unless \ \ \neg x.sweep \wedge over$
, (P20)

$\neg x.white \ \ unless \ \ x.white \wedge x.swept$
, (P21)

$\neg x.sweep \vee \neg x.white \ \ unless \ \ x.white \wedge \neg x.sweep \wedge over$
, Disjunktion und Vereinfachung der Konsequenz durch Widerspruch mit $x.sweep \Rightarrow \neg x.swept$ aus (P9)

$x.garbage \ \ unless \ \ x.garbage \wedge x.free$
, (DP4)

$x.garbage \vee \neg x.sweep \vee \neg x.white \ \ unless \ \ T \wedge x.food \wedge x.white$
, Disjunktion und Vereinfachung durch $x.white \Rightarrow \neg x.free$ aus (P3) sowie Abschwächung der Konsequenz durch $over \Rightarrow T$ (DP8)

$T \wedge x.food \Rightarrow \neg x.white$
, aus der Definition von T und *food*

$x.garbage \vee \neg x.sweep \vee \neg x.white \ \ unless \ \ false$
, aus der vorletzten Formel mit Vereinfachung der rechten Seite durch Widerspruch mit der vorhergehenden Formel

initially $\neg(x.food \wedge x.sweep \wedge x.white)$
, aus der Startmarkierung von N_{COL}

invariant $x.sweep \wedge x.white \Rightarrow \neg x.food$
, aus den beiden letzten Formeln und der Definition einer Invariante sowie der Definition von *garbage* (1)

$\neg x.free \ \ unless \ \ \neg x.free \wedge x.white \wedge x.sweep$
, (P22)

$x.garbage \ \ unless \ \ x.food \wedge \neg x.free$
, (P26)

$\neg x.free \vee \neg x.food \;\; unless \;\; x.food \wedge x.white \wedge x.sweep$
, Disjunktion und Abschwächung der Konsequenz

$\neg x.free \vee \neg x.food \;\; unless \;\; false$
, Vereinfachung der rechten Seite durch Widerspruch mit (1)

initially $\neg(x.free \wedge x.food)$
, aus der Startmarkierung von N_{COL}

invariant $\neg(x.free \wedge x.food)$
, aus den beiden vorhergehenden Formeln und der Definition einer Invariante

□

10.4 Abstützung der Lebendigkeit durch elementare Eigenschaften

In der letzten Verfeinerung der Systemspezifikation werden die noch verbleibenden Lebendigkeitsanforderungen, auf die sich die vorhergehenden Spezifikationen stützen, so weit "zerkleinert", daß sie direkt an der Implementierung ablesbar sind. Diese Anforderungen betreffen sowohl die Markierungsphase als auch die Bereinigungsphase. Im einzelnen heißt das:

- In der Bereinigungsphase wird jeder Knoten nach endlicher Zeit bereinigt: $x.sweep \mapsto x.swept$ (R9).
- Während der Markierungsphase wird jeder Knoten immer wieder ruhig: $marking \mapsto x.quiet$ (R10).

Die zuletzt genannte Anforderung stützt sich u.a. auf die Eigenschaft, daß während der Markierungsphase niemals existierende Kanten überprüft werden müssen, die auf freie Knoten zeigen (siehe DP10).

10.4.1 Fairness zwischen Collector und Mutator

Um (R10) beweisen zu können, ist noch eine weitere, ganz besondere Eigenschaft erforderlich. Damit der Collector die Markierungsphase immer wieder beenden kann, darf er nicht an der Überprüfung von Kanten gehindert werden. Genau diese Situation kann jedoch eintreten, wenn der Mutator eine bestimmte Kante fortlaufend löscht und dann wieder erzeugt. Unter Umständen ist es dem Collector dann nicht möglich, diese Kante zu überprüfen, da sie immer nur für einen sehr kurzen Zeitraum existiert bzw. nicht existiert und die meiste Zeit damit verbringt, ihren Zustand zu wechseln.

Um solche Situationen auszuschließen, muß lokale Fairness gefordert werden. Wir fordern, daß die Stelle E sich fair verhält, d.h. Transitionen, die die Stelle E in ihrem Vorbereich haben, dürfen nicht unendlich oft aktiviert sein, ohne zu schalten. Anders ausgedrückt: wenn eine dieser Transitionen "genügend" oft aktiviert war, dann darf sie auch irgendwann schalten. Abbildung 10.3 zeigt ein Petrinetz, das den entscheidenden System-

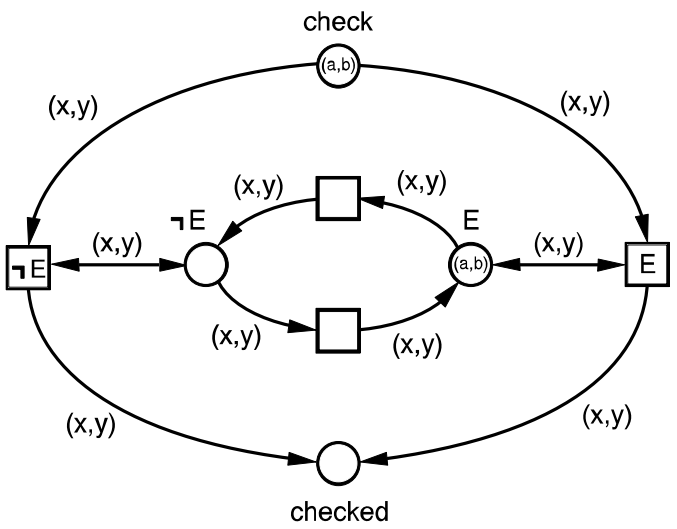


Abb. 10.3. Eine Situation, die Fairness erfordert

ausschnitt modelliert. Ohne Fairness-Annahme ist nicht sichergestellt, daß die Kante (*a*,*b*), die sich in *check* befindet, jemals nach *checked* gelangt. An dieser Stelle sei auf die formale Beschreibung des Fairness-Theorems in Abschnitt 4.5.1 verwiesen, welches im Beweis von *SPEC5 System* Verwendung findet:

10.4.2 Spezifikation *SPEC5 System*

Eigenschaften

(P1-P27)

$\neg x.free \;\; unless \;\; x.free \wedge x.swept$ (P28)

$checked[x,y] \;\; unless \;\; \langle \forall\, y :: checked[x,y] \rangle$ (P29)

$x.white \wedge x.sweep \wedge \neg x.black \wedge \neg x.free \wedge \neg x.swept \;\; ensures$
$(x.free \wedge x.swept) \vee (x.black \wedge check[x,y])$ (P30)

$x.black \wedge x.sweep \wedge \neg x.white \wedge \neg x.free \;\; ensures \;\; x.swept$ (P31)

$x.free \wedge x.sweep \wedge \neg x.black \wedge \neg x.white \;\; ensures \;\; x.swept$ (P32)

$\langle \forall\, y :: checked[x,y] \rangle \;\; ensures \;\; x.quiet$ (P33)

$check[x,y] \wedge y.black \;\; ensures^{E[x,y]} \;\; checked[x,y] \vee y.white$ (P34)

$E[x,y] \;\; fair$ (P35)

$check[x,y] \wedge y.black \wedge E[x,y] \;\; unless$
$checked[x,y] \vee y.white \vee (check[x,y] \wedge \neg E[x,y])$ (P36)

$check[x,y] \wedge \langle \forall\, z :: \neg check[y,z] \rangle \;\; ensures$
$checked[x,y] \vee \neg E[x,y] \vee \neg y.white \vee \neg y.quiet \vee y.black$ (P37)

$check[x,y] \;\; ensures \;\; checked[x,y] \vee E[x,y]$ (P38)

10.4.3 Theorem

Aus (P1-P38) kann die folgende Eigenschaft abgeleitet werden:
invariant $check[x,y] \wedge E[x,y] \Rightarrow \neg y.free$ (DP10)

10.4.4 Theorem

SPEC4 System kann aus *SPEC5 System* hergeleitet werden, d.h. (R7,R9,R10) sind aus *SPEC5 System* herleitbar.

Beweis: (DP10) **invariant** $check[x,y] \wedge E[x,y] \Rightarrow \neg y.free$

$\neg z.blackreach \;\; unless \;\; \neg z.blackreach \wedge (z.food \vee z.free)$
, (DP7)

$\neg E[z,x] \;\; unless \;\; x.black$
, aus (P24) durch Abschwächung der Konsequenz

$\neg z.blackreach \vee \neg E[z,x] \;\; unless \;\; x.black \vee (E[z,x] \wedge (z.food \vee z.free))$
, Disjunktion und Abschwächung der Konsequenz

$\neg z.blackreach \vee \neg E[z,x] \;\; unless \;\; \neg x.free$
, Vereinfachung der rechten Seite durch $x.black \Rightarrow \neg x.free$ aus (P3)
und $z.food \wedge E[z,x] \Rightarrow x.food$ aus (DP1) sowie $x.food \Rightarrow \neg x.free$
(DP9) und $z.free \Rightarrow \neg E[z,x]$ (P2)

$\langle \forall\, z :: \neg z.blackreach \vee \neg E[z,x] \rangle \;\; unless \;\; \neg x.free$
, einfache Konjunktion über alle z

$\neg x.black \;\; unless \;\; x.black$
, Antireflexivitat von *unless*

$\neg x.black \wedge \langle \forall\, z :: \neg z.blackreach \vee \neg E[z,x] \rangle \;\; unless \;\; \neg x.free$
, Konjunktion und Vereinfachung der Konsequenz durch
$x.black \Rightarrow \neg x.free$ aus (P3)

$\neg x.blackreach \;\; unless \;\; \neg x.free$
, Anwendung der Definition von *blackreach* und
$\langle \exists z :: z.black \wedge R[z,x] \rangle \;\Leftrightarrow\; x.black \vee \langle \exists z :: z.blackreach \wedge E[z,x] \rangle$

$\neg x.free \;\; unless \;\; x.free \wedge x.swept$
, (P28)

$\neg x.blackreach \vee \neg x.free \;\; unless \;\; x.blackreach \wedge x.free \wedge x.swept$
, Disjunktion der beiden vorhergehenden Formeln

$x.swept \;\; unless \;\; clean$
, (P6)

$\neg x.blackreach \vee \neg x.free \vee x.swept$ *unless* *false*
, Disjunktion und Vereinfachung der rechten Seite durch Widerspruch mit $clean \Rightarrow \langle \forall z :: \neg z.black \rangle$ (DP5) und $\langle \forall z :: \neg z.black \rangle \Rightarrow \neg x.blackreach$

initially $\neg x.blackreach \vee \neg x.free \vee x.swept$
, aus der Startmarkierung von N_{COL}

invariant $y.blackreach \wedge \neg y.swept \Rightarrow \neg y.free$
, aus den beiden vorhergehenden Formeln und der Definition einer Invariante

invariant $check[x,y] \Rightarrow x.black$
, (DP6)

invariant $check[x,y] \wedge E[x,y] \Rightarrow \neg y.free$
, aus den beiden vorhergehenden Invarianten mit $x.black \wedge E[x,y] \Rightarrow y.blackreach$ sowie $check[x,y] \Rightarrow marking \Rightarrow \neg sweeping \Rightarrow \neg y.swept$ aus (P5) □

Beweis: (R9) $x.sweep \mapsto x.swept$

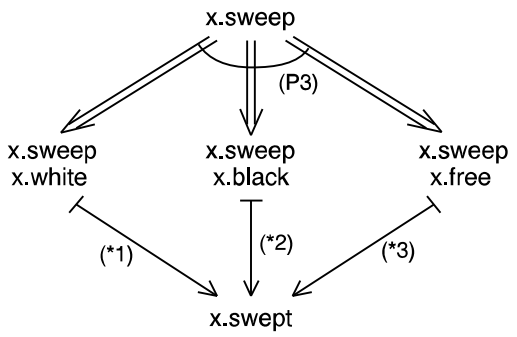


Abb. 10.4. Proof Lattice 3

siehe Proof Lattice 3 in Abbildung 10.4 □

Beweis: (*1,R7) $x.white \wedge x.sweep \mapsto x.free \wedge x.swept$

$x.white \wedge x.sweep \wedge \neg x.black \wedge \neg x.free \wedge \neg x.swept$ *ensures*
$(x.free \wedge x.swept) \vee (x.black \wedge check[x,y])$
, (P30)

$x.white \wedge x.sweep \;\; ensures \;\; (x.free \wedge x.swept) \vee (x.black \wedge check[x,y])$
, aus der letzten Formel mit $x.white \Rightarrow \neg x.free \wedge \neg x.black$ aus (P3) und $x.sweep \Rightarrow \neg x.swept$ aus (P9)

$x.sweep \;\; unless \;\; x.swept$
, (P23)

$x.white \wedge x.sweep \;\; ensures \;\; x.free \wedge x.swept$
, Konjunktion und Vereinfachung der Konsequenz durch Widerspruch mit $x.sweep \Rightarrow \neg x.swept$ aus (P9) und $(x.sweep \vee x.swept) \Rightarrow sweeping \Rightarrow \neg marking \Rightarrow \neg check[x,y]$ aus (P5)

$x.white \wedge x.sweep \mapsto x.free \wedge x.swept$
, aus der letzten Formel und der Definition von $\mapsto$ □

Beweis: (*2) $x.sweep \wedge x.black \mapsto x.swept$

$x.black \wedge x.sweep \wedge \neg x.white \wedge \neg x.free \;\; ensures \;\; x.swept$
, (P31)

$x.black \wedge x.sweep \;\; ensures \;\; x.swept$
, aus der letzten Formel mit $x.black \Rightarrow \neg x.white \wedge \neg x.free$ aus (P3); zusammen mit der Definition von $\mapsto$ folgt die Behauptung □

Beweis: (*3) $x.sweep \wedge x.free \mapsto x.swept$

$x.free \wedge x.sweep \wedge \neg x.black \wedge \neg x.white \;\; ensures \;\; x.swept$
, (P32)

$x.free \wedge x.sweep \;\; ensures \;\; x.swept$
, aus der letzten Formel mit $x.free \Rightarrow \neg x.black \wedge \neg x.white$ aus (P3); zusammen mit der Definition von $\mapsto$ folgt die Behauptung □

Beweis: (R10) $marking \mapsto x.quiet$

siehe Proof Lattice 4 in Abbildung 10.5 □

Beweis: (*1) $check[x,y] \wedge E[x,y] \Rightarrow y.black \vee y.white$

invariant $check[x,y] \wedge E[x,y] \Rightarrow \neg y.free$
, (DP10); zusammen mit $\neg y.free \Rightarrow y.black \vee y.white$ aus (P3) folgt die Behauptung □

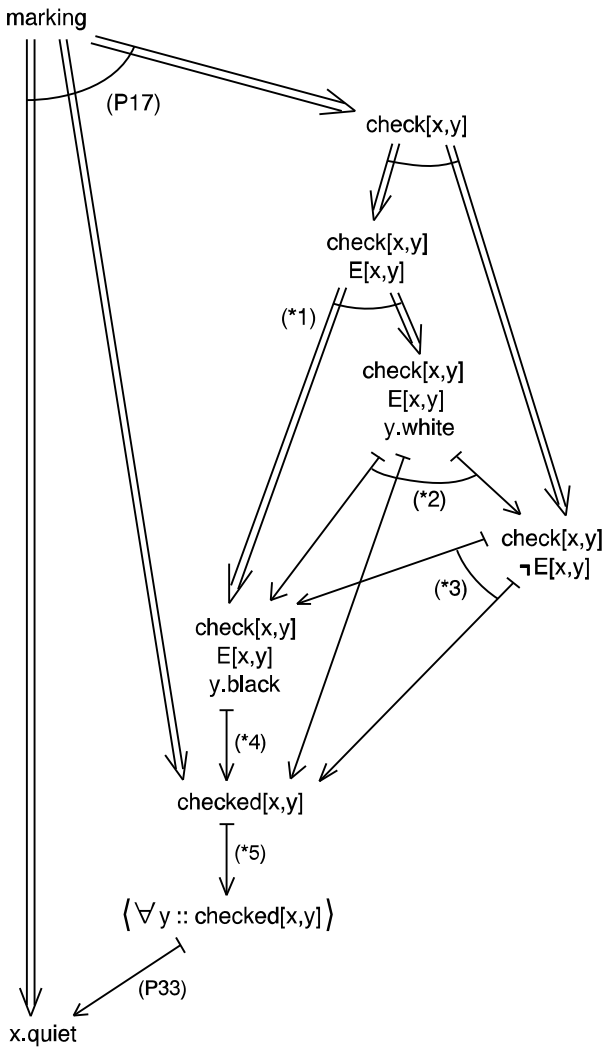


Abb. 10.5. Proof Lattice 4

Beweis: (*2) $check[x,y] \wedge y.white \wedge E[x,y] \mapsto$
$checked[x,y] \vee (check[x,y] \wedge (\neg E[x,y] \vee (E[x,y] \wedge y.black)))$

$check[x,y]$ *unless* $checked[x,y]$
, aus (P19) durch Abschwächung der Konsequenz

$y.quiet$ *unless* $y.black \vee y.sweep$
, (P7)

$check[x,y] \wedge y.quiet$ *unless* $(check[x,y] \wedge y.black) \vee checked[x,y]$
, Konjunktion und Vereinfachung der Konsequenz durch Widerspruch
mit $y.sweep \Rightarrow sweeping \Rightarrow \neg marking \Rightarrow \neg check[x,y]$ aus (P5)

$y.white \;\; unless \;\; y.black \vee y.free$
, aus (P3)

$check[x,y] \wedge y.quiet \wedge y.white \;\; unless$
$\quad checked[x,y] \vee (check[x,y] \wedge (y.black \vee y.free))$
, Konjunktion und Abschwächung der Konsequenz

$E[x,y] \;\; unless \;\; \neg E[x,y]$
, Antireflexivität von *unless*

$check[x,y] \wedge y.quiet \wedge y.white \wedge E[x,y] \;\; unless$
$\quad checked[x,y] \vee (check[x,y] \wedge (\neg E[x,y] \vee (E[x,y] \wedge y.black)))$
, Konjunktion der beiden letzten Formeln und Abschwächung der Konsequenz sowie Vereinfachung durch Widerspruch mit
$check[x,y] \wedge E[x,y] \Rightarrow \neg y.free$ (DP10) (1)

$check[x,y] \wedge \langle \forall\, z :: \neg check[y,z] \rangle \;\; ensures$
$\quad checked[x,y] \vee \neg E[x,y] \vee \neg y.white \vee \neg y.quiet \vee y.black$
, (P37)

$check[x,y] \wedge E[x,y] \wedge y.quiet \wedge y.white \;\; \mapsto$
$\quad checked[x,y] \vee \neg E[x,y] \vee \neg y.white \vee \neg y.quiet \vee y.black$
, aus der vohergehenden Formel und der Definition von ensures (Konjunktion der negierten Nachbedingung auf die Vorbedingung) sowie $y.quiet \Rightarrow \neg check[y,z]$ aus (P17), dann Anwendung der Definition von $\mapsto$

$check[x,y] \wedge y.quiet \wedge y.white \wedge E[x,y] \;\; \mapsto$
$\quad checked[x,y] \vee (check[x,y] \wedge (\neg E[x,y] \vee (E[x,y] \wedge y.black)))$
, PSP-Theorem auf der letzten Formel und (1) sowie Vereinfachung der rechten Seite durch Widerspruch mit $check[x,y] \Rightarrow \neg checked[x,y]$ aus (P17) und $y.white \Rightarrow \neg y.black$ aus (P3) (2)

$\neg marking \;\; unless \;\; marking \wedge (x.quiet \vee x.black)$
, aus (P16) durch Abschwächung der Konsequenz

$x.quiet \;\; unless \;\; x.black \vee x.sweep$
, (P7)

$\neg marking \vee x.quiet \;\; unless \;\; marking \wedge x.black$
, Disjunktion und Vereinfachung der Konsequenz durch Widerspruch mit $marking \Rightarrow \neg sweeping \Rightarrow \neg x.sweep$ aus (P5)

$\neg x.white \;\; unless \;\; x.white \wedge x.swept$
, (P21)

$\neg marking \vee \neg x.white \vee x.quiet \;\; unless \;\; false$
, Disjunktion und Vereinfachung der rechten Seite durch Widerspruch mit $x.white \Rightarrow \neg x.black$ aus (P3), sowie $x.swept \Rightarrow \neg marking$ aus (P5)

initially $\neg marking \vee \neg x.white \vee x.quiet$
, aus der Startmarkierung von N_{COL}

invariant $marking \wedge x.white \Rightarrow x.quiet$
, aus den beiden vorhergehenden Formeln und der Definition einer Invariante

$check[x,y] \wedge y.white \wedge E[x,y] \;\mapsto$
$\quad check[x,y] \wedge y.quiet \wedge y.white \wedge E[x,y]$
, aus der vorhergehenden Formel mit $check[x,y] \Rightarrow marking$, sowie Anwendung des Implikationstheorems für $\mapsto$; die Behauptung folgt aus der Transitivität von $\mapsto$ mit (2) □

Beweis: (*3) $check[x,y] \wedge \neg E[x,y] \;\mapsto$
$\quad checked[x,y] \vee (check[x,y] \wedge E[x,y] \wedge y.black)$

$check[x,y] \;\; unless \;\; checked[x,y]$
, aus (P19) durch Abschwächung der Konsequenz

$\neg E[x,y] \;\; unless \;\; E[x,y] \wedge y.black$
, (P24)

$check[x,y] \wedge \neg E[x,y] \;\; unless$
$\quad checked[x,y] \vee (check[x,y] \wedge E[x,y] \wedge y.black)$
, Konjunktion der beiden letzten Formeln und Abschwächung der Konsequenz

$check[x,y] \wedge \neg E[x,y] \;\mapsto\; checked[x,y] \vee E[x,y]$
, aus (P38) mit Konjunktion der negierten Nachbedingung $\neg E[x,y]$ auf die Vorbedingung und Anwendung der Definition von $\mapsto$

$check[x,y] \wedge \neg E[x,y] \;\mapsto$
$\quad checked[x,y] \vee (check[x,y] \wedge E[x,y] \wedge y.black)$
, PSP-Theorem und Verkürzung der rechten Seite durch Widerspruch mit $check[x,y] \Rightarrow \neg checked[x,y]$ aus (P17) □

Beweis: (*4) $check[x,y] \wedge E[x,y] \wedge y.black \;\mapsto\; checked[x,y]$

$check[x,y] \;\; unless \;\; checked[x,y]$
, aus (P19) durch Abschwächung der Konsequenz

$y.black \;\; unless \;\; y.black \wedge y.sweep$
, (P4)

$check[x,y] \wedge y.black \;\; unless \;\; checked[x,y] \wedge y.black$
, Konjunktion der beiden vorhergehenden Formeln und Vereinfachung der rechten Seite durch Widerspruch mit $check[x,y] \Rightarrow marking \Rightarrow \neg sweeping \Rightarrow \neg y.sweep$ aus (P5) (1)

$check[x,y] \wedge y.black \;\; ensures^{E[x,y]} \;\; checked[x,y] \vee y.white$
, (P34)

$E[x,y] \;\; fair$
, (P35)

$check[x,y] \wedge y.black \wedge E[x,y] \;\; unless$
$checked[x,y] \vee y.white \vee (check[x,y] \wedge \neg E[x,y])$
, (P36)

$check[x,y] \wedge \neg E[x,y] \;\; ensures \;\; checked[x,y] \vee y.white \vee E[x,y]$
, aus (P38) mit Konjunktion der negierten Nachbedingung $\neg E[x,y]$ auf die Vorbedingung und Abschwächung der Konsequenz durch $y.white$

$check[x,y] \wedge y.black \wedge E[x,y] \;\mapsto\; checked[x,y] \vee y.white$
, Fairness-Theorem auf den letzten 4 Formeln

$check[x,y] \wedge y.black \wedge E[x,y] \;\mapsto\; checked[x,y]$
, PSP-Theorem mit (1) und Vereinfachung der Konsequenz durch Widerspruch mit $check[x,y] \Rightarrow \neg checked[x,y]$ aus (P17) und $y.black \Rightarrow \neg y.white$ aus (P3) □

Beweis: (*5) $checked[x,y] \;\mapsto\; \langle \forall\, y \;::\; checked[x,y] \rangle$

$marking \wedge \neg x.quiet \;\Rightarrow\; checked[x,y] \vee check[x,y]$
, aus (P17)

$check[x,y] \;\mapsto\; checked[x,y]$
, siehe Proof Lattice 4 aus (*1 - *4)

$marking \wedge \neg x.quiet \;\mapsto\; checked[x,y]$
, Implikationstheorem auf der vorletzten Formel und Löschungstheorem mit der vorhergehenden Formel

$marking \wedge \neg x.quiet \;\mapsto\; checked[x,y] \vee \langle \forall\, y \;::\; checked[x,y] \rangle$
, Abschwächung der Konsequenz

$checked[x,y] \;\; unless \;\; \langle \forall\, y \;::\; checked[x,y] \rangle$
, (P29)

$marking \wedge \neg x.quiet \mapsto \langle \forall\ y :: checked[x,y] \rangle$

, Vervollständigungstheorem auf den letzten beiden Formeln

$checked[x,y] \mapsto marking \wedge \neg x.quiet$

, aus (P17) und dem Implikationstheorem für $\mapsto$; die Behauptung folgt aus der Transitivität von $\mapsto$ und der letzten Formel □

11 Aufteilung der Systemspezifikation

In der Spezifikationsverfeinerung des letzten Kapitels haben wir über Eigenschaften des Gesamtsystems gesprochen. Da unsere Implementierung N_{COL} aber nur den Garbage Collector modelliert, nicht aber den Mutator, müssen wir jetzt die Eigenschaften des Systems auf die Eigenschaften der Komponenten Mutator und Collector zurückführen. Diese Systematik hat den Vorteil, daß wir den Mutator als "variable" Umgebung betrachten können. Ohne einen speziellen Mutator als Beispiel heranzuziehen, sind wir in der Lage, die Korrektheit unserer Collector-Implementierung nachzuweisen. Als Schnittstellenbeschreibung bleibt uns die Spezifikation des Mutators, denn jedes Netz, das sich an die nun folgende Spezifikation hält, ist ein Mutator, mit dem N_{COL} korrekt zusammenarbeiten kann (korrekt im Sinne von *SPEC1 Collector*). Der Nachweis, daß die beiden Komponenten *Mutator* und *Collector* zusammen die letzte Systemspezifikation *SPEC5 System* tragen können, wird im Anschluß an die Spezifikationen der Teilsysteme durchgeführt.

11.1 Spezifikation des Mutators

Die in der nachfolgenden Spezifikation benutzten logischen Operatoren tragen den Index κ, um anzudeuten, daß es sich um kompositionale Operatoren handelt. Insbesondere darf für ihre Herleitung nicht das Substitutions-Axiom benutzt werden. Für eine Erläuterung dieses Sachverhalts sei auf Abschnitt 5.3 verwiesen.

11.1.1 Spezifikation *SPEC Mutator*

Eigenschaften

$\neg E[x,y]$ *unless*$_\kappa$ $\neg E[x,y] \wedge (y.food \vee y.free)$ (MP1)

$\neg E[x,y]$ *unless*$_\kappa$ $E[x,y] \wedge \neg x.free \wedge y.black$ (MP2)

$E[x,y]$ *fair*$_\kappa$ (MP3)

$x.garbage$ *unless*$_\kappa$ $x.food \wedge \neg x.free$ (MP4)

invariant$_\kappa$ $x.free$ *or* $x.black$ *or* $x.white$ (MP5)

stable$_\kappa$ $\neg x.free$ (MP6)

stable$_\kappa$ $x.black$ (MP7)

$\neg x.black$ *unless*$_\kappa$ $\neg x.black \wedge x.quiet \wedge (x.free \vee x.food)$ (MP8)

$x.white$ *unless*$_\kappa$ $x.black \wedge check[x,y]$ (MP9)

stable$_\kappa$ $\neg x.white$ (MP10)

constant$_\kappa$ $x.sweep$ (MP11)

constant$_\kappa$ $x.swept$ (MP12)

$$\textbf{constant}_{\mathrm{K}}\ \ checked[x,y] \tag{MP13}$$
$$\textbf{stable}_{\mathrm{K}}\ \ check[x,y] \tag{MP14}$$
$$\neg check[x,y]\ \ unless_{\mathrm{K}}\ \ \neg check[x,y] \wedge x.quiet \tag{MP15}$$
$$\neg check[x,y]\ \ unless_{\mathrm{K}}\ \ check[x,y] \wedge x.black \wedge \neg x.quiet \tag{MP16}$$
$$\textbf{stable}_{\mathrm{K}}\ \ \neg x.quiet \tag{MP17}$$
$$x.quiet\ \ unless_{\mathrm{K}}\ \ \neg x.quiet \wedge check[x,y] \wedge x.black \tag{MP18}$$
$$x.quiet\ \ unless_{\mathrm{K}}\ \ x.quiet \wedge x.white \wedge x.food \tag{MP19}$$

11.1.2 Bewertung

Einige der "Kooperations-Auflagen" die in *SPEC Mutator* formuliert sind, wurden bereits in Abschnitt 7.2 informell vorweggenommen. Der Zusammenhang stellt sich dabei folgendermaßen dar: (MS1) entspricht (MP1), (MS2) wird durch (MP2) ausgedrückt, (MS3) entspricht (MP9), und (MS4) ist in (MP8) enthalten. Alle anderen Formeln in *SPEC Mutator* können grob in zwei Klassen eingeteilt werden:

- Die privaten Stellen des Collectors sind im Mutator konstant (vgl. (MP11-13)). Die anderen Stellen bilden die Schnittstelle zwischen Collector und Mutator; abgesehen von E und $\neg E$ darf der Mutator entweder aus einer Stelle Marken nur entnehmen oder in eine Stelle Marken nur einbringen. Daher ist im Mutator jede Stelle des Collectors oder deren Komplement stabil (vgl. (MP6, MP7, MP10, MP14, MP17)).
- Die verbleibenden Formeln regeln die Details der Kooperation. So wird z.B. durch (MP5) festgelegt, daß jeder Knoten genau in einer der drei Stellen *black*, *free* oder *white* enthalten sein muß. (MP3) fordert, daß sich die Stelle E fair verhalten soll.

Einige der Kooperationsbedingungen sind härter als es für die korrekte Funktion des Collectors erforderlich wäre. Dazu zählt insbesondere (MP2). Wir fordern dort, daß der Mutator, der eine Kante erzeugt, gleichzeitig den Zielknoten schwarz färben muß. Für den Algorithmus des Collectors würde es ausreichen, daß die beiden Ereignisse hintereinander eintreten, d.h. zuerst wird die Kante erzeugt und dann wird der Zielknoten nach *black* gebracht. Durch die Verschärfung der Kooperationsbedingungen ergibt sich jedoch ein etwas einfacherer Korrektheitsnachweis.

11.2 Terminale Spezifikation des Collectors

SPEC2 Collector ist die Verfeinerung der ursprünglichen Collector-Spezifikation *SPEC1 Collector*. Die Eigenschaften (CP1-CP3) aus *SPEC1* werden direkt nach *SPEC2* übertragen, da sie im Gesamtsystem nicht sichtbar sind. Die Anforderungen (R1) und (R2) waren Grundlage der Systemspezifikation und werden letztendlich durch (CP4-CP43) getragen.

11.2.1 Spezifikation *SPEC2 Collector*

Eigenschaften

$\mathbf{stable}_{\kappa}\ \ x.free$ (CP1)

$\mathbf{stable}_{\kappa}\ \ \neg E[x,y]$ (CP2)

$\mathbf{stable}_{\kappa}\ \ E[x,y] \vee x.free$ (CP3)

$\mathbf{invariant}_{\kappa}\ \ x.free\ or\ x.black\ or\ x.white$ (CP4)

$\mathbf{invariant}_{\kappa}\ \ x.free \Rightarrow \neg E[x,y]$ (CP5)

$\neg x.free\ \ unless_{\kappa}\ \ \neg x.free \wedge x.white \wedge x.sweep$ (CP6)

$\neg x.free\ \ unless_{\kappa}\ \ x.free \wedge x.swept$ (CP7)

$x.black\ \ unless_{\kappa}\ \ x.black \wedge x.sweep$ (CP8)

$\neg x.black\ \ unless_{\kappa}\ \ \neg x.black \wedge (x.food \vee \langle \exists\, z :: check[z,x] \wedge E[z,x] \rangle)$ (CP9)

$\neg x.black\ \ unless_{\kappa}\ \ check[x,y]$ (CP10)

$x.white\ \ unless_{\kappa}\ \ (x.free \wedge x.swept) \vee (x.black \wedge check[x,y])$ (CP11)

$\neg x.white\ \ unless_{\kappa}\ \ x.white \wedge x.swept$ (CP12)

$\mathbf{invariant}_{\kappa}\ x.sweep\ or\ x.swept\ or\ x.quiet\ or\ check[x,y]\ or\ checked[x,y]$ (CP13)

$x.sweep\ \ unless_{\kappa}\ \ x.swept$ (CP14)

$x.sweep \wedge x.white \wedge \neg x.black \wedge \neg x.free \wedge \neg x.swept\ \ ensures_{\kappa}$
$(x.free \wedge x.swept) \vee (x.black \wedge check[x,y])$ (CP15)

$x.sweep \wedge x.black \wedge \neg x.white \wedge \neg x.free\ \ ensures_{\kappa}\ \ x.swept$ (CP16)

$x.sweep \wedge x.free \wedge \neg x.black \wedge \neg x.white\ \ ensures_{\kappa}\ \ x.swept$ (CP17)

$\neg x.sweep\ \ unless_{\kappa}\ \ \neg x.sweep \wedge over$ (CP18)

$\neg x.sweep\ \ unless_{\kappa}\ \ \langle \forall\, z :: z.sweep \rangle$ (CP19)

$x.swept\ \ unless_{\kappa}\ \ clean$ (CP20)

$clean\ \ unless_{\kappa}\ \ \langle \forall\, x,y :: x.quiet \vee (check[x,y] \wedge x.black) \rangle \wedge check[r,z]$ (CP21)

$clean \wedge r.white\ \ ensures_{\kappa}\ \ x.quiet \vee checked[x,y] \vee r.black$ (CP22)

$\neg x.swept\ \ unless_{\kappa}\ \ x.sweep$ (CP23)

$\neg x.swept\ \ unless_{\kappa}\ \ x.swept \wedge (x.white \vee x.free)$ (CP24)

$x.quiet\ \ unless_{\kappa}\ \ \langle \forall\, z :: z.sweep \rangle \vee (check[x,y] \wedge x.black)$ (CP25)

$x.quiet\ \ unless_{\kappa}\ \ (x.quiet \wedge x.white) \vee over$ (CP26)

$x.quiet\ \ unless_{\kappa}\ \ check[x,y] \vee over$ (CP27)

$x.quiet\ \ unless_{\kappa}\ \ \langle \forall\, z :: z.sweep \rangle \vee$
$(x.quiet \wedge x.white \wedge \langle \exists\, y :: check[y,x] \rangle)$ (CP28)

$over \wedge \langle \forall\, x :: \neg x.sweep \rangle\ \ ensures_{\kappa}\ \ x.sweep \vee \langle \exists\, z,y :: check[z,y] \rangle$ (CP29)

$\neg x.quiet\ \ unless_{\kappa}\ \ \neg x.quiet \wedge (checked[x,y] \vee x.swept)$ (CP30)

$check[x,y]\ \ unless_{\kappa}\ \ \neg check[x,y] \wedge checked[x,y]$ (CP31)

$check[x,y]\ \ unless_{\kappa}\ \ checked[x,y] \wedge (\neg E[x,y] \vee y.black)$ (CP32)

$check[x,y] \wedge y.black\ \ ensures_{\kappa}^{E[x,y]}\ \ checked[x,y] \vee y.white$ (CP33)

$E[x,y]\ \ fair_{\kappa}$ (CP34)

$check[x,y] \wedge y.black \wedge E[x,y]$ $unless_{\kappa}$
$\quad checked[x,y] \vee y.white \vee (check[x,y] \wedge \neg E[x,y])$ (CP35)

$check[x,y] \wedge \langle \forall\, z :: \neg check[y,z] \rangle$ $ensures_{\kappa}$
$\quad checked[x,y] \vee \neg E[x,y] \vee \neg y.white \vee \neg y.quiet \vee y.black$ (CP36)

$check[x,y]$ $ensures_{\kappa}$ $checked[x,y] \vee E[x,y]$ (CP37)

$\neg check[x,y]$ $unless_{\kappa}$ $\neg check[x,y] \wedge (clean \vee x.quiet)$ (CP38)

$\neg check[x,y]$ $unless_{\kappa}$ $check[x,y] \wedge x.black$ (CP39)

$checked[x,y]$ $unless_{\kappa}$ $x.quiet$ (CP40)

$\langle \forall\, y :: checked[x,y] \rangle$ $ensures_{\kappa}$ $x.quiet$ (CP41)

$checked[x,y]$ $unless_{\kappa}$ $\langle \forall\, y :: checked[x,y] \rangle$ (CP42)

$\neg checked[x,y]$ $unless_{\kappa}$ $\neg checked[x,y] \wedge check[x,y]$ (CP43)

11.2.2 Theorem

SPEC2 Collector kann aus N_{COL} hergeleitet werden, d.h. (CP1-CP43) sind Eigenschaften von N_{COL}.

Der Beweis dieses Theorems wird hier nur exemplarisch ausgeführt (CP11, CP29, CP30), da die meisten Formeln sich sehr einfach am Netz ablesen lassen:

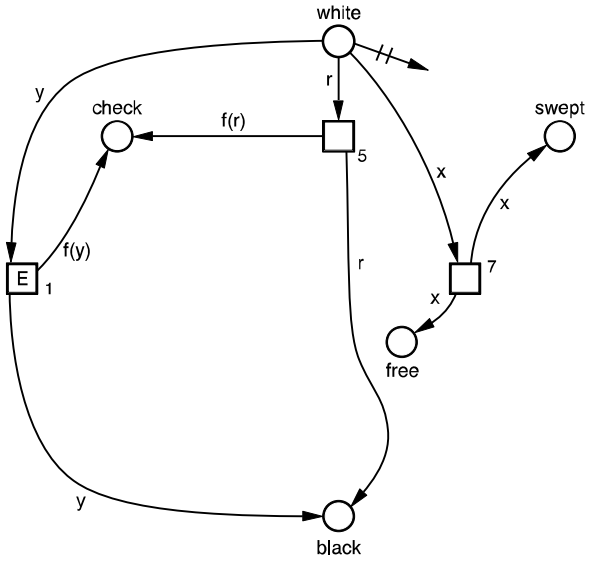


Abb. 11.1. Ablesen der Eigenschaft (CP11)

- Abbildung 11.1 zeigt den für (CP11) relevanten Ausschnitt aus N_{COL} (vgl. S. 92). Ein Knoten x ist solange weiß, bis er entweder bereinigt wird und dann in *free* und *swept* liegt, oder er wird schwarz und seine Kanten kommen zur Überprüfung nach *check*. Insbesondere gibt es keine anderen Transitionen, die aus der Stelle *white* Marken entnehmen, daher

$$x.white \quad unless_{\kappa} \quad (x.free \wedge x.swept) \ \vee \ (x.black \wedge check[x,y])$$

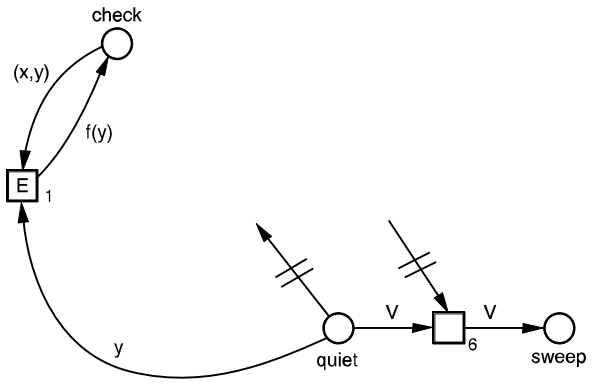


Abb. 11.2. Ablesen der Eigenschaft (CP29)

- Anhand des Ausschnitts in Abbildung 11.2 kann (CP29) abgelesen werden. Wenn sich alle Knoten in *quiet* befinden und es keine Kante mehr in *check* gibt, dann wird irgendwann die Transition 6 schalten und alle Knoten nach *sweep* bringen, daher

$$over \ \wedge \ \langle \forall \, x \ :: \ \neg x.sweep \rangle \quad ensures_{\kappa} \quad x.sweep \ \vee \ \langle \exists \, z,y \ :: \ check[z,y] \rangle$$

- Abbildung 11.3 schließlich zeigt einen Ausschnitt von N_{COL}, aus dem die Eigenschaft (CP30) ablesbar ist. Wenn ein Knoten nicht “ruhig” ist, dann kann er erst ruhig werden, wenn alle seine Kanten in *checked* liegen oder alle Knoten bereinigt sind, also

$$\neg x.quiet \quad unless_{\kappa} \quad \neg x.quiet \ \wedge \ (checked[x,y] \ \vee \ x.swept)$$

11.2.3 Nebenbedingungen

Abgesehen von zwei Formeln können alle Eigenschaften am Netz N_{COL} abgelesen oder aus S-Invarianten hergeleitet werden. Die beiden nicht ablesbaren Formeln existieren parallel zum Netz als Nebenbedingungen:

- Freie Knoten haben keine abgehenden Kanten, oder $x.free \Rightarrow \ \neg E[x,y]$ (siehe CP5).
- Die Stelle E hat sich fair zu verhalten, oder $E[x,y]$ $fair_{\kappa}$ (CP34).

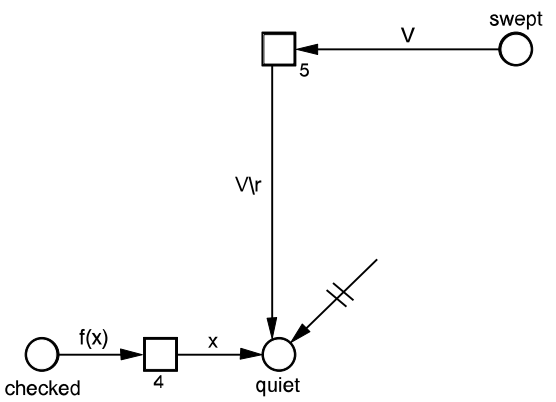


Abb. 11.3. Ablesen der Eigenschaft (CP30)

11.3 Abstützung des Systems auf die Komposition von Mutator und Collector

Die Komposition von Mutator und Collector durch Vereinigung notieren wir als *Mutator* ‖ *Collector*. Zwei Petrinetze werden vereinigt, indem die jeweiligen Bestandteile vereinigt werden. Dabei ist zu beachten, daß die Transitionsmengen disjunkt sind und die Startmarkierungen identisch (vgl. 5.1.1). Darüberhinaus muß das Gesamtnetz, ebenso wie die beiden Komponenten Mutator und Collector, 1-sicher sein. Von unserem Collector N_{COL} haben wir die 1-Sicherheit bereits gezeigt (vgl. 7.3.1). Die Forderung nach der 1-Sicherheit des Mutators und der Komposition *Mutator* ‖ *Collector* bleibt als Nebenbedingung bestehen. Wir werden im weiteren diese Voraussetzungen als gegeben annehmen.

Eigenschaften, die aus einer Komponente allein abgelesen werden, gelten u.U. auch in der Komposition. Diesen Sachverhalt formuliert das Vereinigungstheorem, welches in 5.2.2 notiert ist. Nachfolgend werden die Eigenschaften (P1-P38) aus *SPEC5 System* durch die Eigenschaften der Systemkomponenten Mutator und Collector abgestützt. Für jede dieser Eigenschaften ergibt sich in etwa das gleiche Beweisschema:

- Anpassung einer Eigenschaft des Mutators an eine Eigenschaft des Collectors, oder umgekehrt.
- Anwendung des Vereinigungstheorems
- Streichung des Operatorindex κ, da die Operatoren im Gesamtsystem nicht mehr kompositional sein müssen. Insbesondere ist damit die Verwendung des Substitutions-Axioms erlaubt.

Eigenschaften der Komposition, die sich in den einzelnen Beweisen als nützlich erwiesen haben, werden als (K1-K4) vorangestellt:

11.3.1 Theorem

Aus der Komposition von *SPEC Mutator* und *SPEC2 Collector* lassen sich die folgenden Eigenschaften ableiten:

invariant *x.sweep or x.swept or x.quiet or*
check[*x*,*y*] *or checked*[*x*,*y*] in *Mutator* [] *Collector* (K1)

¬*marking unless clean* in *Mutator* [] *Collector* (K2)

invariant *marking or sweeping* in *Mutator* [] *Collector* (K3)

invariant *x.free or x.black or x.white* in *Mutator* [] *Collector* (K4)

11.3.2 Theorem

SPEC5 System kann aus der Komposition von *SPEC Mutator* und *SPEC2 Collector* hergeleitet werden, d.h. (P1-P38) sind Eigenschaften in *Mutator* [] *Collector*.

Beweis: (K1) **invariant** *x.sweep or x.swept or x.quiet or*
check[*x*,*y*] *or checked*[*x*,*y*] in *Mutator* [] *Collector*

stable$_\kappa$ ¬*x.quiet* in *Mutator*
, (MP17)

stable$_\kappa$ *check*[*x*,*y*] in *Mutator*
, (MP14)

¬*x.quiet* ∧ *check*[*x*,*y*] *unless*$_\kappa$ *false* in *Mutator*
, Konjunktion der beiden letzten Formeln (1)

¬*check*[*x*,*y*] *unless*$_\kappa$ *check*[*x*,*y*] ∧ *x.black* ∧ ¬*x.quiet* in *Mutator*
, (MP16)

x.quiet unless$_\kappa$ ¬*x.quiet* ∧ *check*[*x*,*y*] ∧ *x.black* in *Mutator*
, (MP18)

x.quiet ∧ ¬*check*[*x*,*y*] *unless*$_\kappa$ ¬*x.quiet* ∧ *check*[*x*,*y*] in *Mutator*
, Konjunktion und Abschwächung der Konsequenz

stable$_\kappa$ *x.quiet or check*[*x*,*y*] in *Mutator*
, Disjunktion der letzten Formel mit (1)

stable$_\kappa$ *x.sweep or x.swept or x.quiet or check*[*x*,*y*] *or checked*[*x*,*y*]
in *Mutator*
, aus der letzten Formel mit **constant** *x.sweep* (MP11),
constant *x.swept* (MP12) und **constant** *checked*[*x*,*y*] (MP13) durch

einfache Disjunktion und einfache Konjunktion über den Aufbau von *or*

invariant$_K$ *x.sweep or x.swept or x.quiet or*
check[*x*,*y*] *or checked*[*x*,*y*] in *Collector*
, (CP13); die Behauptung folgt aus dieser und der letzten Formel durch das Vereinigungstheorem □

Beweis: (K2) ¬*marking unless clean* in *Mutator* [] *Collector*

¬*x.quiet unless* ¬*x.quiet* ∧ (*checked*[*x*,*y*] ∨ *x.swept*)
in *Mutator* [] *Collector*
, aus (CP30) und (MP17): **stable**$_K$ ¬*x.quiet* und dem Vereinigungstheorem

¬*check*[*x*,*y*] *unless* ¬*check*[*x*,*y*] ∧ (*clean* ∨ *x.quiet*)
in *Mutator* [] *Collector*
, aus (CP38) und (MP15) durch Abschwächung der Konsequenz sowie dem Vereinigungstheorem

¬*x.quiet* ∧ ¬*check*[*x*,*y*] *unless*
¬*x.quiet* ∧ ¬*check*[*x*,*y*] ∧ (*clean* ∨ *checked*[*x*,*y*])
, Konjunktion der beiden vorhergehenden Formeln und Abschwächung der Konsequenz

¬*checked*[*x*,*y*] *unless* ¬*checked*[*x*,*y*] ∧ *check*[*x*,*y*]
in *Mutator* [] *Collector*
, aus (CP43) und (MP13): **constant**$_K$ *checked*[*x*,*y*] sowie dem Vereinigungstheorem

¬*x.quiet* ∧ ¬*check*[*x*,*y*] ∧ ¬*checked*[*x*,*y*] *unless*
¬*x.quiet* ∧ ¬*check*[*x*,*y*] ∧ ¬*checked*[*x*,*y*] ∧ *clean*
, Konjunktion der beiden letzten Formeln und Abschwächung der Konsequenz

⟨∀ *x*,*y* :: ¬*x.quiet* ∧ ¬*check*[*x*,*y*] ∧ ¬*checked*[*x*,*y*]⟩ *unless*
⟨∀ *x*,*y* :: ¬*x.quiet* ∧ ¬*check*[*x*,*y*] ∧ ¬*checked*[*x*,*y*]⟩ ∧ *clean*
, Konjunktion über alle *x* und *y*

¬*marking unless clean* in *Mutator* [] *Collector*
, Anwendung der Definition von *marking* und Abschwächung der Konsequenz □

Beweis: (K3,P5) **invariant** *marking or sweeping* in *Mutator* [] *Collector*

¬*marking unless clean* in *Mutator* [] *Collector*
, (K2)

clean unless ⟨∀ *x*,*y* :: *x.quiet* ∨ *check*[*x*,*y*]⟩ in *Mutator* [] *Collector*
, aus (CP21) und **constant**$_\kappa$ *x.swept* (MP12) sowie dem Vereinigungstheorem

¬*marking* ∨ *clean unless* ⟨∀ *x*,*y* :: *x.quiet* ∨ *check*[*x*,*y*]⟩
, Löschungstheorem auf den beiden vorhergehenden Formeln

¬*marking unless* ⟨∀ *x*,*y* :: *x.quiet* ∨ *check*[*x*,*y*]⟩
, aus der letzten Formel und *clean* ⇒ ⟨∀*x* :: *x.swept*⟩ ⇒ ¬*marking*
aus (K1) (1)

x.swept unless clean in *Mutator* [] *Collector*
, aus (CP20) und **constant**$_\kappa$ *x.swept* (MP12) sowie dem Vereinigungstheorem

x.sweep unless x.swept in *Mutator* [] *Collector*
, aus (CP14) und **constant**$_\kappa$ *x.sweep* (MP11) mit dem Vereinigungstheorem

x.sweep ∨ *x.swept unless clean*
, Disjunktion und Abschwächung der Konsequenz

sweeping unless clean
, einfache Disjunktion der letzten Formel über alle *x* und Anwendung der Definition von *sweeping*

clean unless ⟨∀ *x*,*y* :: *x.quiet* ∨ *check*[*x*,*y*]⟩ in *Mutator* [] *Collector*
, aus (CP21) und **constant**$_\kappa$ *x.swept* (MP12) sowie dem Vereinigungstheorem

sweeping ∨ *clean unless* ⟨∀ *x*,*y* :: *x.quiet* ∨ *check*[*x*,*y*]⟩
, Löschungstheorem auf den letzten beiden Formeln

sweeping unless ⟨∀ *x*,*y* :: *x.quiet* ∨ *check*[*x*,*y*]⟩
, Vereinfachung der linken Seite durch *clean* ⇒ *sweeping*

¬*marking* ∧ *sweeping unless* ⟨∀ *x*,*y* :: *x.quiet* ∨ *check*[*x*,*y*]⟩
, einfache Konjunktion der letzten Formel mit (1) (2)

¬*x.sweep unless* ⟨∀ *z* :: *z.sweep*⟩ in *Mutator* [] *Collector*
, aus (CP19) und **constant**$_\kappa$ *x.sweep* (MP11) mit dem Vereinigungstheorem

$\neg x.swept \;\; unless \;\; x.sweep$ in *Mutator* [] *Collector*
, aus (CP23) und **constant**$_\kappa$ *x.swept* (MP12) mit dem Vereinigungstheorem

$\neg x.sweep \wedge \neg x.swept \;\; unless \;\; \langle \forall\, z :: z.sweep \rangle$
, Konjunktion und Abschwächung der Konsequenz

$\neg sweeping \;\; unless \;\; \langle \forall\, z :: z.sweep \rangle$
, einfache Konjunktion der letzten Formel über alle *x* und Anwendung der Definition von *sweeping* (3)

$x.quiet \;\; unless \;\; \langle \forall\, z :: z.sweep \rangle \vee check[x,y]$ in *Mutator* [] *Collector*
, aus (MP18) und (CP25) jeweils mit Abschwächung der Konsequenz; Anwendung des Vereinigungstheorems

$check[x,y] \;\; unless \;\; \neg check[x,y] \wedge checked[x,y]$ in *Mutator* [] *Collector*
, aus (CP31) und **stable**$_\kappa$ *check*[*x*,*y*] (MP14) mit dem Vereinigungstheorem

$x.quiet \vee check[x,y] \;\; unless$
$(\neg x.quiet \wedge checked[x,y] \wedge \neg check[x,y]) \vee \langle \forall\, z :: z.sweep \rangle$
, Disjunktion und Abschwächung der Konsequenz

$checked[x,y] \;\; unless \;\; x.quiet$ in *Mutator* [] *Collector*
, aus (CP40) und **constant**$_\kappa$ *checked*[*x*,*y*] (MP13) mit dem Vereinigungstheorem

$x.quiet \vee check[x,y] \vee checked[x,y] \;\; unless \;\; \langle \forall\, z :: z.sweep \rangle$
, Disjunktion und Abschwächung der Konsequenz

$marking \;\; unless \;\; \langle \forall\, z :: z.sweep \rangle$
, einfache Disjunktion der letzten Formel über alle *x* und Anwendung der Definition von *marking*

$marking \wedge \neg sweeping \;\; unless \;\; \langle \forall\, z :: z.sweep \rangle$
, einfache Konjunktion der letzten Formel mit (3)

$marking \;\; or \;\; sweeping \;\; unless \;\; false$
, Disjunktion mit (2) und Verwendung von $\langle \forall x :: x.sweep \rangle \Rightarrow \neg z.quiet \wedge \neg check[z,y] \wedge \neg checked[z,y] \Rightarrow \neg marking$ sowie $\langle \forall x,y :: x.quiet \vee check[x,y] \rangle \Rightarrow \neg z.sweep \wedge \neg z.swept \Rightarrow \neg sweeping$ aus (K1)

initially $marking \;\; or \;\; sweeping$
, aus der Startmarkierung von N_{COL}

invariant *marking or sweeping* in *Mutator* [] *Collector*
, aus den beiden vorhergehenden Formeln und der Definition einer Invariante □

Beweis: (K4,P3) **invariant** *x.free or x.black or x.white* in *Mutator* [] *Collector*

aus (MP5) und (CP4) durch das Vereinigungstheorem □

Beweis: (P1) $\neg E[x,y]$ *unless* $\neg E[x,y] \wedge (y.food \vee y.free)$ in *Mutator* [] *Collector*

stable$_\kappa$ $\neg E[x,y]$ in *Collector*
, (CP2)

$\neg E[x,y]$ *unless*$_\kappa$ $\neg E[x,y] \wedge (y.food \vee y.free)$ in *Mutator*
, (MP1); die Behauptung folgt zusammen mit der vohergehenden Formel und dem Vereinigungstheorem □

Beweis: (P2) **invariant** $x.free \Rightarrow \neg E[x,y]$ in *Mutator* [] *Collector*

$\neg E[x,y]$ *unless*$_\kappa$ $E[x,y] \wedge \neg x.free$ in *Mutator*
, aus (MP2) durch Abschwächung der Konsequenz

$\neg x.free$ *unless*$_\kappa$ *false* in *Mutator*
, (MP6)

$\neg E[x,y] \vee \neg x.free$ *unless*$_\kappa$ *false* in *Mutator*
, Disjunktion der beiden letzten Formeln

invariant $x.free \Rightarrow \neg E[x,y]$ in *Mutator* [] *Collector*
, aus der letzten Formel und (CP5) durch das Vereinigungstheorem □

Beweis: (P3) **invariant** *x.free or x.black or x.white* in *Mutator* [] *Collector*

identisch mit (K4) □

Beweis: (P4) *x.black unless x.black* $\wedge$ *x.sweep* in *Mutator* [] *Collector*

stable$_\kappa$ *x.black* in *Mutator*
, (MP7)

x.black unless$_\kappa$ *x.black* $\wedge$ *x.sweep* in *Collector*
, (CP8); die Behauptung folgt aus dieser und der letzten Formel mit dem Vereinigungstheorem □

Beweis: (P5) **invariant** *marking or sweeping* in *Mutator* [] *Collector*

identisch mit (K3) □

Beweis: (P6) *x.swept unless clean* in *Mutator* [] *Collector*

aus (CP20) und **constant**$_{\kappa}$ *x.swept* (MP12) sowie dem Vereinigungstheorem □

Beweis: (P7) *x.quiet unless x.black* $\vee$ *x.sweep* in *Mutator* [] *Collector*

aus (CP25) und (MP18) jeweils mit Abschwächung der Konsqequenz; Anwendung des Vereinigungstheorems □

Beweis: (P8) *x.quiet unless* (*x.quiet* $\wedge$ *x.white*) $\vee$ *over* in *Mutator* [] *Collector*

folgt aus (CP26) und (MP19) durch Abschwächung der Konsequenz mit dem Vereinigungstheorem □

Beweis: (P9) **invariant** *sweeping* $\Rightarrow$ *x.sweep or x.swept* in *Mutator* [] *Collector*

folgt aus (K1) und (K3) □

Beweis: (P10) $\neg$*check*[*x*,*y*] *unless check*[*x*,*y*] $\wedge$ *x.black* in *Mutator* [] *Collector*

folgt aus (CP39) und (MP16) mit dem Vereinigungstheorem □

Beweis: (P11) $\neg$*x.black unless* $\neg$*x.black* $\wedge$
(*x.free* $\vee$ *x.food* $\vee$ $\langle\exists\, z :: check[z,x] \wedge E[z,x]\rangle$)
in *Mutator* [] *Collector*

folgt aus (CP9) und (MP8) durch Abschwächung der Konsequenz und dem Vereinigungstheorem □

Beweis: (P12) *clean* $\wedge$ *r.white ensures marking* $\vee$ *r.black* in *Mutator* [] *Collector*

x.swept unless$_{\kappa}$ *false* in *Mutator*
, aus (MP12)

clean unless$_{\kappa}$ *false* in *Mutator*
, einfache Konjunktion der letzten Formel über alle *x*

r.white unless$_{\kappa}$ *r.black* $\wedge$ *check*[*r*,*y*] in *Mutator*
, aus (MP9) durch Ersetzen von *x* durch *r*

clean $\wedge$ *r.white unless*$_{\kappa}$ *r.black* in *Mutator*
, Konjunktion der letzten beiden Formeln und Abschwächung der Konsequenz

clean $\wedge$ *r.white* *ensures*$_\kappa$ *x.quiet* $\vee$ *checked*[*x*,*y*] $\vee$ *r.black*
in *Collector*
, (CP22); die Behauptung folgt zusammen mit der vorhergehenden Formel aus dem Vereinigungstheorem und der Definition von *marking* □

Beweis: (P13) $\neg$*x.swept* *unless* *x.swept* $\wedge$ (*x.white* $\vee$ *x.free*)
in *Mutator* [] *Collector*

folgt aus (CP24) und **constant**$_\kappa$ *x.swept* (MP12) mit dem Vereinigungstheorem □

Beweis: (P14) *marking* *unless* *marking* $\wedge$ *over* in *Mutator* [] *Collector*

x.quiet *unless* *over* $\vee$ *check*[*x*,*y*] in *Mutator* [] *Collector*
, aus (CP27) und (MP18) mit Abschwächung der Konseqenz; Anwendung des Vereinigungstheorems

check[*x*,*y*] *unless* $\neg$*check*[*x*,*y*] $\wedge$ *checked*[*x*,*y*] in *Mutator* [] *Collector*
, aus (CP31) und **stable**$_\kappa$ *check*[*x*,*y*] (MP14) mit dem Vereinigungstheorem

x.quiet $\vee$ *check*[*x*,*y*] *unless*
($\neg$*x.quiet* $\wedge$ *checked*[*x*,*y*] $\wedge$ $\neg$*check*[*x*,*y*]) $\vee$ *over*
, Disjunktion und Abschwächung der Konsequenz

checked[*x*,*y*] *unless* *x.quiet* in *Mutator* [] *Collector*
, aus (CP40) und **constant**$_\kappa$ *checked*[*x*,*y*] (MP13) mit dem Vereinigungstheorem

x.quiet $\vee$ *check*[*x*,*y*] $\vee$ *checked*[*x*,*y*] *unless* *over*
, Disjunktion und Abschwächung der Konsequenz

marking *unless* *marking* $\wedge$ *over*
, einfache Disjunktion der letzten Formel über alle *x* und Anwendung der Definition von *marking* sowie Abschwächung der Konsequenz mit *over* $\Rightarrow$ *marking* □

Beweis: (P15) $\neg$*x.black* *unless* *marking* in *Mutator* [] *Collector*

$\neg$*x.black* *unless*$_\kappa$ *x.quiet* in *Mutator*
, aus (MP8) durch Abschwächung der Konsequenz

$\neg$*x.black* *unless*$_\kappa$ *check*[*x*,*y*] in *Collector*
, (CP10); die Behauptung folgt aus dieser und der letzten Formel durch das Vereinigungstheorem und der Definition von *marking* □

Beweis: (P16) $\neg marking \;\; unless \;\; marking \wedge r.black \wedge (x.quiet \vee x.black)$
in *Mutator* [] *Collector*

$clean \;\; unless_{\kappa} \;\; false$ in *Mutator*
, aus (MP12) durch einfache Konjunktion über alle x

$clean \;\; unless \;\; \langle \forall x,y :: x.quiet \vee (check[x,y] \wedge x.black) \rangle \wedge check[r,z]$
in *Mutator* [] *Collector*
, aus (CP21) und der letzten Formel durch das Vereinigungstheorem

$clean \;\; unless \;\; marking \wedge r.black \wedge (x.quiet \vee x.black)$
, Abschwächung der Konsequenz mit $check[r,z] \Rightarrow marking$ und $check[r,z] \Rightarrow \neg r.quiet$ aus (K1)

$\neg marking \;\; unless \;\; clean$ in *Mutator* [] *Collector*
, (K2)

$\neg marking \vee clean \;\; unless \;\; marking \wedge r.black \wedge (x.quiet \vee x.black)$
, Löschungstheorem auf den beiden vorhergehenden Formeln

$\neg marking \;\; unless \;\; marking \wedge r.black \wedge (x.quiet \vee x.black)$
in *Mutator* [] *Collector*
, aus der letzten Formel und $clean \Rightarrow \langle \forall x :: x.swept \rangle \Rightarrow \neg marking$ aus (K1)

□

Beweis: (P17) **invariant** $marking \Rightarrow x.quiet \;\; or \;\; check[x,y] \;\; or \;\; checked[x,y]$
in *Mutator* [] *Collector*

folgt aus (K1) und (K3) □

Beweis: (P18) $x.quiet \;\; unless \;\; \langle \forall z :: z.sweep \rangle \vee$
$(x.quiet \wedge x.white \wedge (\langle \exists y :: check[y,x] \rangle \vee x.food))$
in *Mutator* [] *Collector*

folgt aus (CP28) und (MP19) jeweils mit Abschwächung der Konsequenz und aus dem Vereinigungstheorem □

Beweis: (P19) $check[x,y] \;\; unless \;\; checked[x,y] \wedge (\neg E[x,y] \vee y.black)$
in *Mutator* [] *Collector*

folgt aus (CP32) und $\mathbf{stable}_{\kappa} \; check[x,y]$ (MP14) durch das Vereinigungstheorem □

Beweis: (P20) $\neg x.sweep \;\; unless \;\; \neg x.sweep \wedge over$ in *Mutator* [] *Collector*

folgt aus (CP18) und $\mathbf{constant}_{\kappa} \; x.sweep$ (MP11) durch das Vereinigungstheorem □

Beweis: (P21) $\neg x.white \;\; unless \;\; x.white \wedge x.swept$ in *Mutator* [] *Collector*

folgt aus (CP12) und $\mathbf{stable}_{\kappa}\, \neg x.white$ (MP10) durch das Vereinigungstheorem □

Beweis: (P22) $\neg x.free \;\; unless \;\; \neg x.free \wedge x.white \wedge x.sweep$ in *Mutator* [] *Collector*

folgt aus (CP6) und $\mathbf{stable}_{\kappa}\, \neg x.free$ (MP6) durch das Vereinigungstheorem □

Beweis: (P23) $x.sweep \;\; unless \;\; x.swept$ in *Mutator* [] *Collector*

folgt aus (CP14) und $\mathbf{constant}_{\kappa}\, x.sweep$ (MP11) durch das Vereinigungstheorem □

Beweis: (P24) $\neg E[x,y] \;\; unless \;\; E[x,y] \wedge y.black$ in *Mutator* [] *Collector*

folgt aus $\mathbf{stable}_{\kappa}\, \neg E[x,y]$ (CP2) und (MP2) mit Abschwächung der Konsequenz; Anwendung des Vereinigungstheorems □

Beweis: (P25) $x.white \;\; unless \;\; (x.free \wedge x.swept) \vee (x.black \wedge check[x,y])$
in *Mutator* [] *Collector*

folgt aus (CP11) und (MP9) mit Abschwächung der Konsequenz durch das Vereinigungstheorem □

Beweis: (P26) $x.garbage \;\; unless \;\; x.food \wedge \neg x.free$ in *Mutator* [] *Collector*

$\mathbf{stable}_{\kappa} \;\; \neg E[x,y]$ in *Collector*
, (CP2)

$\mathbf{stable}_{\kappa} \;\; x.garbage$ in *Collector*
, aus der letzten Formel und der Definition von *garbage*

$x.garbage \;\; unless_{\kappa} \;\; x.food \wedge \neg x.free$ in *Mutator*
, (MP4); mit der letzten Formel folgt die Behauptung aus dem Vereinigungstheorem □

Beweis: (P27) $over \wedge \langle \forall\, x :: \neg x.sweep \rangle \;\; ensures \;\; x.sweep \vee \langle \exists\, z,y :: check[z,y] \rangle$
in *Mutator* [] *Collector*

$x.quiet \;\; unless_{\kappa} \;\; check[x,y]$ in *Mutator*
, aus (MP18) mit Abschwächung der Konsequenz

$over \;\; unless_{\kappa} \;\; \langle \exists\, z,y :: check[z,y] \rangle$ in *Mutator*
, einfache Konjunktion der letzten Formel über alle x und y

$\langle \forall\, x :: \neg x.sweep \rangle \;\; unless_{\kappa} \;\; \langle \exists\, x :: x.sweep \rangle$
, Antireflexivität von *unless*

$over \wedge \langle \forall\, x :: \neg x.sweep \rangle$ $unless_{\kappa}$
$\langle \exists\, z,y :: check[z,y] \rangle \vee (over \wedge \langle \exists\, x :: x.sweep \rangle)$ in *Mutator*
, Konjunktion der beiden vorhergehenden Formeln und Abschwächung der Konsequenz

$over \wedge \langle \forall\, x :: \neg x.sweep \rangle$ $ensures_{\kappa}$ $x.sweep \vee \langle \exists\, z,y :: check[z,y] \rangle$
in *Collector*
, (CP29)

$over \wedge \langle \forall\, x :: \neg x.sweep \rangle$ *ensures*
$x.sweep \vee \langle \exists\, z,y :: check[z,y] \rangle \vee (over \wedge \langle \exists\, x :: x.sweep \rangle)$
in *Mutator* [] *Collector*
, aus den beiden letzten Formeln durch das Vereinigungstheorem; die Behauptung folgt mit Vereinfachung der rechten Seite durch Widerspruch mit $over \Rightarrow \langle \forall\, x :: \neg x.sweep \rangle$ aus (K1) □

Beweis: (P28) $\neg x.free$ *unless* $x.free \wedge x.swept$ in *Mutator* [] *Collector*

folgt aus (CP7) und $\mathbf{stable}_{\kappa}\, \neg x.free$ (MP6) durch das Vereinigungstheorem □

Beweis: (P29) $checked[x,y]$ *unless* $\langle \forall\, y :: checked[x,y] \rangle$ in *Mutator* [] *Collector*

folgt aus (CP42) und $\mathbf{constant}_{\kappa}\, checked[x,y]$ (MP13) durch das Vereinigungstheorem □

Beweis: (P30) $x.white \wedge x.sweep \wedge \neg x.black \wedge \neg x.free \wedge \neg x.swept$ *ensures*
$(x.free \wedge x.swept) \vee (x.black \wedge check[x,y])$ in *Mutator* [] *Collector*

$x.sweep$ $unless_{\kappa}$ *false* in *Mutator*
, aus (MP11)

$x.white$ $unless_{\kappa}$ $x.black \wedge check[x,y]$ in *Mutator*
, (MP9)

$x.sweep \wedge x.white$ $unless_{\kappa}$ $x.black \wedge check[x,y]$ in *Mutator*
, Konjunktion der beiden vorhergehenden Formeln und Abschwächung der Konsequenz

$\neg x.black$ $unless_{\kappa}$ $\neg x.black \wedge x.quiet$ in *Mutator*
, aus (MP8) durch Abschwächung der Konsequenz

$x.sweep \wedge x.white \wedge \neg x.black$ $unless_{\kappa}$ $x.sweep \wedge x.quiet$ in *Mutator*
, Konjunktion der beiden letzten Formeln und Abschwächung der Konsequenz

$x.sweep \wedge x.white \wedge \neg x.black \wedge \neg x.free \wedge \neg x.swept \;\; unless_{\kappa}$
$x.sweep \wedge x.quiet$ in *Mutator*
, aus der vorhergehenden Formel durch einfache Konjunktion mit (MP6) und (MP12)

$x.white \wedge x.sweep \wedge \neg x.black \wedge \neg x.free \wedge \neg x.swept \;\; ensures_{\kappa}$
$(x.free \wedge x.swept) \vee (x.black \wedge check[x,y])$ in *Collector*
, (CP15)

$x.white \wedge x.sweep \wedge \neg x.black \wedge \neg x.free \wedge \neg x.swept \;\; ensures$
$(x.free \wedge x.swept) \vee (x.black \wedge check[x,y]) \vee (x.sweep \wedge x.quiet)$
in *Mutator* [] *Collector*
, aus den beiden letzten Formeln durch das Vereinigungstheorem; die Behauptung folgt durch Vereinfachung der rechten Seite durch Widerspruch mit $x.sweep \Rightarrow \neg x.quiet$ aus (K1) □

Beweis: (P31) $x.black \wedge x.sweep \wedge \neg x.white \wedge \neg x.free \;\; ensures \;\; x.swept$
in *Mutator* [] *Collector*

$x.black \wedge x.sweep \wedge \neg x.white \wedge \neg x.free \;\; unless_{\kappa} \;\; false$ in *Mutator*
, einfache Konjunktion von (MP11), (MP7), (MP10) und (MP6)

$x.black \wedge x.sweep \wedge \neg x.white \wedge \neg x.free \;\; ensures_{\kappa} \;\; x.swept$
in *Collector*
, (CP16); die Behauptung folgt zusammen mit der letzten Formel aus dem Vereinigungstheorem □

Beweis: (P32) $x.free \wedge x.sweep \wedge \neg x.black \wedge \neg x.white \;\; ensures \;\; x.swept$
in *Mutator* [] *Collector*

$x.sweep \;\; unless_{\kappa} \;\; false$ in *Mutator*
, (MP11)

$\neg x.white \;\; unless_{\kappa} \;\; false$ in *Mutator*
, (MP10)

$x.sweep \wedge \neg x.white \;\; unless_{\kappa} \;\; false$ in *Mutator*
, Konjunktion der beiden vorhergehenden Formeln

$\neg x.black \;\; unless_{\kappa} \;\; \neg x.black \wedge x.quiet$ in *Mutator*
, aus (MP8) durch Abschwächung der Konsequenz

$x.sweep \wedge \neg x.white \wedge \neg x.black \;\; unless_{\kappa} \;\; x.sweep \wedge x.quiet$ in *Mutator*
, Konjunktion und Abschwächung der Konsequenz

$x.free \;\; unless_{\kappa} \;\; \neg x.free$
, Antireflexivität von *unless*

$x.free \wedge x.sweep \wedge \neg x.white \wedge \neg x.black \;\; unless_{\kappa}$
$(x.sweep \wedge x.quiet) \vee (\neg x.free \wedge \neg x.black \wedge \neg x.white)$ in *Mutator*
, Konjunktion der beiden letzten Formeln und Abschwächung der Konsequenz

$x.free \wedge x.sweep \wedge \neg x.black \wedge \neg x.white \;\; ensures_{\kappa} \;\; x.swept$
in *Collector*
, (CP17)

$x.free \wedge x.sweep \wedge \neg x.white \wedge \neg x.black \;\; ensures$
$(x.sweep \wedge x.quiet) \vee (\neg x.free \wedge \neg x.black \wedge \neg x.white) \vee x.swept$
in *Mutator* [] *Collector*
, aus den beiden letzten Formeln durch das Vereinigungstheorem; die Behauptung folgt mit Vereinfachung der rechten Seite durch Widerspruch mit $x.sweep \Rightarrow \neg x.quiet$ aus (K1) und *x.free or x.black or x.white* (K4) □

Beweis: (P33) $\langle \forall\, y :: checked[x,y] \rangle \;\; ensures \;\; x.quiet$ in *Mutator* [] *Collector*

folgt aus (CP41) und $\mathbf{constant}_{\kappa}\, checked$ (MP13) durch das Vereinigungstheorem □

Beweis: (P34) $check[x,y] \wedge y.black \;\; ensures^{E[x,y]} \;\; checked[x,y] \vee y.white$
in *Mutator* [] *Collector*

folgt aus (CP33) und $\mathbf{stable}_{\kappa}\, y.black$ (MP7), $\mathbf{stable}_{\kappa}\, check[x,y]$ (MP14) durch das Vereinigungstheorem □

Beweis: (P35) $E[x,y]$ *fair* in *Mutator* [] *Collector*

folgt aus (CP34) oder (MP3) durch das Vereinigungstheorem □

Beweis: (P36) $check[x,y] \wedge y.black \wedge E[x,y]$ *unless*
$checked[x,y] \vee y.white \vee (check[x,y] \wedge \neg E[x,y])$
in *Mutator* [] *Collector*

$check[x,y] \wedge y.black \;\; unless_{\kappa} \;\; false$ in *Mutator*
, einfache Konjunktion von (MP14) und (MP7)

$E[x,y] \;\; unless_{\kappa} \;\; \neg E[x,y]$
, Antireflexivität von *unless*

$check[x,y] \wedge y.black \wedge E[x,y]$ $unless_{\kappa}$ $check[x,y] \wedge \neg E[x,y]$
in *Mutator*
, Konjunktion und Abschwächung der Konsequenz

$check[x,y] \wedge y.black \wedge E[x,y]$ $unless_{\kappa}$
$checked[x,y] \vee y.white \vee (check[x,y] \wedge \neg E[x,y])$ in *Collector*
, (CP35); die Behauptung folgt mit Abschwächung der Konsequenz in der letzten Formel zusammen mit dieser Formel durch das Vereinigungstheorem □

Beweis: (P37) $check[x,y] \wedge \langle \forall\, z :: \neg check[y,z] \rangle$ *ensures*
$checked[x,y] \vee \neg E[x,y] \vee \neg y.white \vee \neg y.quiet \vee y.black$
in *Mutator* [] *Collector*

folgt aus (CP36) und **constant**$_{\kappa}$ $check[x,y]$ (MP14) durch das Vereinigungstheorem □

Beweis: (P38) $check[x,y]$ *ensures* $checked[x,y] \vee E[x,y]$ in *Mutator* [] *Collector*

folgt aus (CP37) und **constant**$_{\kappa}$ $check[x,y]$ (MP14) durch das Vereinigungstheorem □

TEIL **5**

Schluß und Anhang

12 Zusammenfassung

In der vorliegenden Arbeit wurde ein nebenläufiger Garbage Collector als Petrinetz modelliert und seine Modellierung formal verifiziert. Unsere Aufmerksamkeit widmeten wir dabei nicht einer neuen algorithmischen Idee, sondern der Adäquatheit von Modellierung und Beweistechnik.

Im einführenden Teil legten wir unser Augenmerk vor allem auf die geschichtliche Entwicklung der parallelen Garbage Collectoren und auf die Methoden, die zum Beweis ihrer Korrektheit benutzt wurden. Wir machten deutlich, daß ein nebenläufiger Garbage Collector zu einer Klasse von Systemen gehört, die mit einigen wesentlichen Aspekten der Problematik verteilter Algorithmen behaftet sind.

Der zweite Teil dieser Arbeit fundierte die theoretischen Aspekte unserer Methoden zur Modellierung und Verifikation paralleler Systeme. Schwerpunkt dieses Teils war die Einführung eines leistungsfähigen Netzkalküls und einer temporalen Logik zur Formulierung von Modelleigenschaften. Darüberhinaus wurde ein Konzept zur Komposition von Netzen durch Vereinigung vorgestellt.

Im dritten Teil modellierten wir einen parallelen Garbage Collector als Petrinetz und zeigten einige Netzeigenschaften anhand von S-Invarianten auf. Ein wesentlicher Aspekt dieser Modellierung ist, daß der Mutator, in seiner Eigenschaft als nicht-triviale Umgebung des Collectors, nicht mit-modelliert wurde, sondern als “variable” zweite Komponente des Gesamtsystems betrachtet wurde.

Daran anschließend notierten wir die formale Spezifikation des Collectors, anhand derer wir die Korrektheit seiner Modellierung nachweisen konnten. Im Zuge der Spezifikationsverfeinerung wurden gleichzeitig die Kooperationsbedingungen für den Mutator erarbeitet.

Anhand eines relativ aufwendigen Fallbeispiels gelang uns der Nachweis, daß Petrinetze zur Modellierung von komplizierten parallelen Systemen geeignet sind. Die “Portierung” der UNITY-Logik aus [CHAN88] auf das Kalkül der Petrinetze lieferte eine adäquate Beweistechnik dazu. Sie führte zu einem formalen Korrektheitsbeweis, in dem die

einzelnen Schritte zumeist auf rein syntaktischen Ableitungsregeln basieren und daher maschinell überprüfbar sind[1].

Für die Abläufe im Petrinetz wurde die Interleaving-Semantik ausgewählt, da sie unter den gegebenen Umständen die meisten Vorteile bot. Der Hauptgrund lag in der Tatsache, daß die verwendete Beweislogik in ihrem Ursprung (d.h. bei [CHAN88]) für die Interleaving-Semantik ausgelegt ist. Ein Umstieg auf die für Petrinetze adäquatere Halbordnungs-Semantik hätte die Theoriebildung aufwendiger werden lassen, aber kurzfristig keine Vereinfachung des Korrektheitsbeweises ergeben.

Wir können daraus die folgenden Schlüsse ziehen: Thema der weiteren Forschungsaktivitäten wird die Anpassung (oder Neuentwicklung) der verwendeten Beweistechnik hinsichtlich der ausdrucksstärkeren Halbordnungs-Semantik sein. Es ist anzunehmen, daß ein Korrektheitsbeweis über einer Halbordnungs-Semantik kürzer ausfällt, als ein vergleichbarer Beweis über der Interleaving-Semantik. Ein weiterer Punkt ist die bereits angesprochene maschinelle Überprüfung der Beweisschritte. Sie ist für Fallbeispiele in der Größenordnung des parallelen Garbage Collectors eigentlich Voraussetzung, um die Handhabbarkeit der Beweise zu gewährleisten (und daher wünschenswert).

1) Um keinen falschen Eindruck zu erwecken: der Beweis in dieser Arbeit wurde (leider) nicht maschinell überprüft.

Literaturverzeichnis

[ABRA87] ABRAHAM, S.G., J.H. PATEL: *Parallel Garbage Collection on a Virtual Memory System*. Proceedings of the 1987 International Conference on Parallel Processing, (University Park, PA): IEEE, 1987, S. 243-246

[ALI86] ALI, K.A.M., S. HARIDI: *Global Garbage Collection for Distributed Heap Storage Systems*. International Journal of Parallel Programming, Vol. 15, No. 5, 1986, S. 339-387

[AUGU87] AUGUSTEIJN, L.: *Garbage Collection in a Distributed Environment*. Parallel Architectures and Languages Europe, Proceedings Eindhoven, Netherlands, Vol. 2, 1987, S. 75-93

[BENA84] BEN-ARI, M.: *Algorithms for On-the-fly Garbage Collection*. ACM Transactions on Programming Languages and Systems, Vol. 6, No. 3, 1984, S. 333-344

[BEVA89] BEVAN, D.I.: *An Efficient Reference Counting Solution to the Distributed Garbage Collection Problem*. Parallel Computing, Vol. 9, No. 3, 1989, S. 179-192

[BUDD87] BUDD, T.: *A Little Smalltalk*. Reading, Massachusetts: Addison-Wesley, 1987

[CHAN88] CHANDY, K.M., J. MISRA: *Parallel Program Design: A Foundation*. Reading, Massachusetts: Addison-Wesley, 1988

[COHE81] COHEN, J.: *Garbage Collection of Linked Data Structures*. Computing Surveys, Vol. 13, No. 3, 1981, S. 341-367

[DIJK76] DIJKSTRA, E.W., L. LAMPORT, A.J. MARTIN, C.S. SCHOLTEN, E.F.M. STEFFENS: *On-the-fly Garbage Collection: An Exercise in Cooperation*. Lecture Notes in Computer Science, Vol. 46, New-York: Springer-Verlag, 1976, S. 43-55

[EBBI86] EBBINGHAUS, H.-D., J. FLUM, W. THOMAS: *Einführung in die mathematische Logik*. Darmstadt: Wissenschaftliche Buchgesellschaft, 1986

[EHN89] EHN, L.: *A Contribution to the Increase of Efficiency of On-the-fly Garbage Collection.* Computers and Artificial Intelligence, Vol. 8, No. 1, 1989, S. 83-91

[EHRI85] EHRIG, H., B. MAHR: *Fundamentals of Algebraic Specification 1*. EATCS Monographs on Theoretical Computer Science, Berlin: Springer-Verlag, 1985

[GOLD89] GOLDBERG, A., D. ROBSON: *Smalltalk-80: The Language*. Reading, Massachusetts: Addison-Wesley, 1989

[GOMM89] GOMM, D., R. WALTER: *Das Problem der verteilten Terminierung: eine Lösung mit Petrinetzen*. Universität Bonn, Diplomarbeit, 1989

[GRIE76] GRIES, D.: *An Exercise in Proving Parallel Programs Correct*. Lecture Notes in Computer Science, Vol. 46, New-York: Springer-Verlag, 1976, S. 57-81

[HARE85] HAREL, D., A. PNUELI: *On the Development of Reactive Systems*. Logics and Models of Concurrent Systems, NATO, ASI-13-Series, Springer-Verlag, 1985, S. 477-498

[HICK84] HICKEY, T., J. COHEN: *Performance Analysis of On-the-fly Garbage Collection*. Communications of the ACM, Vol. 27, No. 11, 1984, S. 1143-1154

[HOAR69] HOARE, C.A.R.: *An Axiomatic Basis for Computer Programming*. Communications of the ACM, Vol. 12, No. 10, 1969, S. 576-583

[HUGH85] HUGHES, J.: *A Distributed Garbage Collection Algorithm*. Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science 201, Berlin: Springer-Verlag, 1985, S. 256-272

[JEFF85a] JEFFERSON, D.R., H. SOWIZRAL: *Fast Concurrent Simulation Using the Time Warp Mechanism*. Distributed Simulation, 1985, S. 63-69

[JEFF85b] JEFFERSON, D.R.: *Virtual Time*. ACM Transactions on Programming Languages and Systems, Vol. 7, No. 3, 1985, S. 404-425

[JULI84] JULIAN, D., M. DAVIES: *Memory Occupancy Patterns in Garbage Collection Systems*. Communications of the ACM, Vol. 27, No. 8, 1984, S. 819-825

[KNUT73] KNUTH, D.E.: *The Art of Computer Programming*, Vol.1: Fundamental Algorithms, Reading, Massachusetts: Addison-Wesley, 1973

[KUNG77a] KUNG, H.T., S.W. SONG: *An Efficient Parallel Garbage Collection System and it's Correctness Proof*. Proceedings of the 18th Annual IEEE Symposium on Foundations of Computer Science (Providence, R.I.), New-York: IEEE, 1977, S. 120-131

[KUNG77b] KUNG, H.T., S.W. SONG: *Performance Analysis of a Parallel Garbage Collection System*. Technical Report, Dept. of Computer Science, Carnegie-Mellon University, 1977

[LAMP76] LAMPORT, L.: *Garbage Collection with Multiple Processes: an Exercise in Parallelism.* Proceedings of the 1976 International Conference on Parallel Processing, Long Beach, California: IEEE, 1976, S. 50-54

[LAMP77] LAMPORT, L.: *Proving the Correctness of Multiprocess Programs*. IEEE Trans. Softw. Eng., SE-3, No. 2, 1977, S. 125-143

[LAMP78] LAMPORT, L.: *Time, Clocks, and the Ordering of Events in a Distributed System*. Communication of the ACM, Vol. 21, No. 7, 1978, S. 558-565

[LANG87] LANG, B., F. DUPONT: *Incremental Incrementally Compacting Garbage Collection.* SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques, SIGPLAN Notes, Vol. 22, No. 7, 1987, S. 253-263

[MEYE88] MEYER, B.: *Object-oriented Software Construction*. Cambridge: Prentice Hall, 1988

[NEWM82] NEWMAN, I.A., M.C. WOODWARD: *Alternative Approaches to Multiprocessor Garbage Collection*. Proceedings of the 1982 International Conference on Parallel Processing, (Bellaire, MI): IEEE, 1982, S. 205-210

[NEWM87] NEWMAN, I.A., R.P. STALLARD, M.C. WOODWARD: *A Hybrid Multiple Processor Garbage Collection Algorithm*. The Computer Journal, Vol. 30, No. 2, 1987, S. 119-127

[OWIC76] OWICKI, S., D. GRIES: *An Axiomatic Proof Technique for Parallel Programs — Part I*. Acta Informatica 6, 1976, S. 319-340

[OWIC82] OWICKI, S., L. LAMPORT: *Proving Liveness Properties of Concurrent Programs*. ACM Transactions on Programming Languages and Systems, Vol. 4, No. 3, 1982, S. 455-495

[PIXL88] PIXLEY, C.: *An Incremental Garbage Collection Algorithm for Multi-Mutator Systems*. Distributed Computing, No. 3, Berlin: Springer-Verlag, 1988, S. 41-50

[PNUE81] PNUELI, A.: *The Temporal Semantics of Concurrent Programs*. Theoretical Computer Science 13, 1981, S. 45-60

[RAM85] RAM, A., J.H. PATEL: *Parallel Garbage Collection Algorithm without Synchronisation Overhead*. Proceedings of the 12th Annual International Symposium on Computer Architecture, (Boston, MA): IEEE, 1985, S. 84-90

[RAME83] RAMESH, S., S.L. MEHNDIRATTA: *The Liveness Property of On-the-fly Garbage Collection — A Proof*. Information Processing Letters, Vol. 17, No. 4, 1983, S. 189-195

[REIS86] REISIG, W.: *Petrinetze: eine Einführung*. Berlin: Springer-Verlag, 1986

[REIS90a] REISIG, W.: *Petri Nets and Algebraic Specifications*. TU München, Institut für Informatik, SFB-Bericht Nr. 342/1/90 B, 1990

[REIS90b] REISIG, W.: *Petrinetze: Grundfragen, Konzepte, Konsequenzen*. Gesellschaft für Mathematik und Datenverarbeitung mbH, Arbeitspapier der GMD 497, 1990

[RUBY87] RUBY, J.: *A Liveness Property of a Parallel Algorithm*. Information Processing Letters, Vol. 24, No. 4, 1987, S. 275-277

[SCHE89] SCHELVIS, M.: *Incremental Distribution of Timestamp Packets: A New Approach to Distributed Garbage Collection*. OOPSLA '89 Proceedings (Object-Oriented Programming: Systems, Languages and Applications), SIGPLAN Notes, Vol. 24, No. 10, 1989, S. 37-48

[SHIN86] SHIN, H.: *Efficient Garbage Collection with Associative Memory*. Journal of the Korean Information Science Society, Vol. 13, No. 4, 1986, S. 346-356

[SNEP87] SNEPSCHEUT VAN DE, J.L.A.: *"Algorithms for On-the-fly Garbage Collection" Revisited*. Information Processing Letters, Vol. 24, No. 4, 1987, S. 211-216

[STEE75] STEELE, G.L.: *Multiprocessing Compactifying Garbage Collection*. Communications of the ACM, Vol. 18, No. 9, 1975, S. 495-508

[STOY86] STOYAN, H., G. GÖRZ: *LISP: Eine Einführung in die Programmierung*. Berlin: Springer-Verlag, 1986

[TANE87] TANENBAUM, A.S.: *Memory Management*. Operating Systems: Design and Implementation, New Jersey: Prentice-Hall, 1987, S. 191-248

[TEL88] TEL, G., R.B. TAN, J. VAN LEEUWEN: *The Derivation of Graph Marking Algorithms from Distributed Termination Detection Protocols*. Science of Computer Programming, Vol. 10, No. 2, 1988, S. 107-137