Verteilte Constraint-basierte Eisenbahn-Simulation

vorgelegt von
Diplom-Informatiker
Hans Schlenker

Von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
– Dr.-Ing. –
genehmigte Dissertation

Promotionsauschuss:
Vorsitzender: Prof. Dr. Fritz Wysotzki
Berichter: Prof. Dr. Stefan Jähnichen
Berichter: Prof. Dr. Thom Frühwirth

Tag der wissenschaftlichen Aussprache: 8. Juli 2004

Berlin 2004
D 83

Kurzfassung

Diese Arbeit entwirft und bewertet einen neuen Algorithmus zur Simulation von Eisenbahn-Netzen: *Distributed Railway Simulation* (DRS). DRS ist ein verteilter Meta-Algorithmus, der lokale, Constraint-basierte Simulationen koordiniert.

Zur Analyse komplexer realer Systeme werden häufig Simulations-Verfahren verwendet. Eisenbahn-Gesellschaften, wie in Deutschland die Deutsche Bahn AG, setzen Simulations-Software ein, um ihre Bahnnetze und Fahrpläne zu untersuchen und bezüglich Kosten und Service-Qualität zu verbessern. Die vorliegende Arbeit fand im Rahmen eines BMBF Förderprojekts statt, in dem untersucht werden sollte, wie man mithilfe von Constraint-Propagation verschiedene Probleme existierender Simulationssysteme lösen kann. Fernziel des Projekts war die erstmalige mikroskopische Simulation des deutschen Eisenbahn-Netzes.

Zur verteilten Simulation per DRS wird zunächst das Simulationsmodell anhand des Gleisnetzes in disjunkte Teile zerlegt, welche auf die verfügbaren Rechenknoten verteilt werden. DRS veranlasst dann alle lokalen Simulatoren, ihre Teile zu berechnen. Nach jeder lokalen Simulation vergleichen die Simulatoren ihre Ergebnisse an den Schnittstellen benachbarter Teile. Solange dort Inkonsistenzen bestehen, werden die lokalen Simulationen mit den Nachbardaten als zusätzliche neue Constraints wiederholt, um die Inkonsistenzen schließlich aufzuheben.

Dieser Algorithmus ist hier detailliert formal beschrieben. Der entwickelte Formalismus ist so organisiert, dass damit zwei wichtige Eigenschaften gezeigt werden können: Terminierung (dass DRS in jeder denkbaren Konfiguration nach endlich vielen Schritten eine global konsistente Lösung berechnet) und Korrektheit (dass die berechnete Lösung ein korrektes Simulationsergebnis ist).

DRS ist theoretisch fundiert, aber auch praktisch solide umgesetzt. Aus der aktuellen Literatur werden Anforderungen an ein solches verteiltes System abgeleitet. Anhand dieser Voraussetzungen wird das Design des Systems entwickelt und beschrieben unter Verwendung des Standards UML: Architektur des Gesamt-Systems, Aufbau der Komponenten, lokale und globale Abläufe. Außerdem wird genauer eingegangen auf die Steuerung des verteilten Algorithmus (inkl. Beweis der Korrektheit der Implementierung), auf den integrierten Entwicklungsprozess und auf die Möglichkeiten der Verwendung unterschiedlicher lokaler Simulatoren.

Die empirischen Untersuchungen anhand der vorliegenden Fallstudie zeigen, dass der Algorithmus DRS und die Implementierung die angestrebten Ziele erfüllen. So können gegebene Eisenbahn-Netze bei Verwendung von DRS schneller simuliert werden als ohne. Das gilt schon für relativ kleine Netze und bei Verwendung nur eines Rechenknotens. Die Ergebnisse lassen aber auch den Schluss zu, dass damit sehr große Netze – und damit das deutsche Bahnnetz – in sehr kurzer Zeit simuliert werden können: Das Verfahren skaliert unter Verwendung vieler Rechenknoten.

Neben den empirischen Ergebnissen ist die theoretische Fundierung ein herausragendes Merkmal dieser Arbeit. Das hier entwickelte Verfahren lässt sich für viele andere Anwendungen einsetzen und bietet Ansätze für einige theoretische und praktische Erweiterungen.

Abstract

This work designs and evaluates a new algorithm for the simulation of railway systems: *Distributed Railway Simulation* (DRS). DRS is a distributed meta-algorithm, that conducts local, constraint-based simulators, to cooperatively compute a global simulation.

For the analysis of complex real-world systems, often simulation approaches are used. Railway companies, like the German Deutsche Bahn AG, use simulation software to examine and test their rail networks and timetables, in order to improve costs and service quality. This work took place within a ministry funded research and development project, where it should be investigated how Constraint Propagation can help overcome certain drawbacks of existing simulation approaches. The long-term objective of the project was the first microscopic simulation of the German rail network.

For a distributed simulation with DRS, first of all the simulation model has to be divided into a number of disjunctive parts. These parts are distributed among the available computing nodes. Then, DRS initiates on all these nodes a local simulation. After each node has completed its simulation computation, it compares its results with that of the neighboring nodes. As long as there exists an inconsistency between some neighbor's results, the local computations are repeated, now taking into account the results of the neighbors. This process is iterated until the global solution is consistent.

This algorithm is described here formally in detail. Using the herein constructed formalism, I prove the two most important properties of the algorithm: termination (it distributedly computes a result after a finite number of steps, for all possible simulation configurations), and correctness (each distributed computation yields a correct simulation result).

DRS has a well-founded theory, but has also been realized soundly. I derive requirements to the realization from state-of-the art literature. Based on the requirements, the system design is developed. The design is described using UML standards: architecture of the overall system, structure of the components, local and global sequences. Additionally, I go into the controller of the distributed computation (incl. formal proof of correctness of the implementation), into the integrated development process, and into the possibilities of using heterogeneous local simulation engines.

The empirical investigations based on the available case study show that the algorithm and the implementation fulfill the aimed goals: The simulation of given railway systems takes less time when using DRS. This even applies to rather small nets and to only one computing node. But, the results even imply that DRS can simulate very large networks – and therefore the German railway net – in very short time: the system scales using a large number of computing nodes.

In addition to the empirical results, the formal foundation is an outstanding property of this work. The presented approach can be used for a number of other applications and has propositions for some theoretical and practical extensions.

Danksagung

Mein Dank gilt vor allem Herrn Prof. Dr. Stefan Jähnichen, dem Leiter unseres Instituts *Fraunhofer FIRST*, für die Betreuung meiner Promotion und die finanzielle und organisatorische Unterstützung. Herrn Prof. Dr. Dr. Thom Frühwirth möchte ich danken für die Begutachtung dieser Arbeit und seine wertvollen Anregungen.

Herrn Prof. Dr. Ulrich Geske, Leiter der Abteilung *Planungstechnik*, gilt mein besonderer Dank. Er hat die Anregung zur Verteilten Simulation gegeben, das Forschungsprojekt *SIMONE* initiiert, und war stets mit anregenden und aufmunternden Ratschlägen zur Stelle. Er hat mir auch in schwierigen Phasen die nötige Zeit und inhaltliche Freiheit gegeben.

Dr. Armin Wolf war mir durch seine scharfen Analysen und seine wissenschaftliche Erfahrung eine besondere Hilfe. Dr. Georg Ringwelski möchte ich für die vielen fruchtbaren Diskussionen und inhaltlichen Korrekturen danken, mit ihm zusammen neue Ideen zu entwickeln war immer eine besondere Freude.

Ganz besonders danken möchte ich auch Julija Ruhmane und Patrick Mukherjee, die halfen, rudimentäre Ideen zu konkretisieren und alles in eine solide Implementierung zu gießen.

Für die Durchsicht der Arbeit, inhaltliche, organisatorische und sprachliche Korrekturen waren neben den oben genannten behilflich: Dr. Stefan Fricke, Henry Müller, Matthias Hoche, Ole Boysen und vor allem Christian Quatmann. Auch dafür vielen Dank!

Meine Eltern, viele Freunde, und allen voran meine geliebte Frau Indre haben mich immer wieder motiviert, stets unterstützt und mir oft den richtigen Weg gewiesen. Ihnen gilt mein herzlichster Dank.

Inhaltsverzeichnis

Teil I

Einführung

Dieser erste Teil führt allgemein in die Thematik der Arbeit ein und umreißt ihren Beitrag.

In Kapitel 1 [S. 2] motiviere ich einleitend die Arbeit, definiere konkret angestrebte Ziele und erläutere den Aufbau.

Die vorliegende Arbeit verbindet relativ entfernte Bereiche der Informatik: Verteilte Problemlösung, Constraint-Programmierung, Simulation. Um die Arbeit eigenständig und in sich geschlossen zu halten, biete ich in den folgenden Kapiteln entsprechend umfangreiche Einführungen an: Kapitel 2 [S. 6] erläutert Motivation und Grundkonzepte der Verteilten Problemlösung, Kapitel 3 [S. 21] umreißt die Technologie der Constraint-Programmierung, Kapitel 4 [S. 38] beschreibt grundlegende Ansätze der Simulation, und Kapitel 5 [S. 42] skizziert die konkrete Anwendung: Eisenbahn-Simulation. Am Ende jedes dieser Kapitel ordne ich die vorliegende Arbeit in den Kontext der verwandten Arbeiten ein.

Zur Konkretisierung und überleitend in den nächsten Teil folgt schließlich in Kapitel 6 [S. 47] ein künstliches Eisenbahn-Simulations-Beispiel.

---

1 Einleitung

Die folgende Einleitung beschreibt ganz grundlegend, was Eisenbahn-Simulation macht und wozu sie eingesetzt wird. Anhand von existierenden Problemen wird der Beitrag dieser Arbeit motiviert: ein neuer Algorithmus zur verteilten, Constraint-basierten Simulation. Dafür werden konkrete, hier zu erreichende Ziele definiert. Außerdem wird der wissenschaftliche Beitrag, den diese Arbeit bereits erbracht hat, skizziert. Abschließend folgen einige Erläuterungen zum Aufbau der Arbeit.

1.1 Motivation und Ziele

Zur Analyse komplexer realer Systeme werden häufig Simulations-Verfahren verwendet. Eisenbahnnetze sind sehr komplexe Systeme. Dafür setzen Eisenbahn-Gesellschaften, wie in Deutschland die Deutsche Bahn AG, Simulations-Software ein. Damit können sie ihre Netze untersuchen und auf Grund der gewonnenen Erkenntnisse verbessern. Weil der Betrieb von Eisenbahnnetzen außerordentlich viel Geld kostet, stellen die Betreiber besonders hohe Anforderungen an die Simulations-Werkzeuge, die sie verwenden.

Eisenbahn-Simulation bedeutet, virtuelle Züge durch ein virtuelles Gleisnetz fahren zu lassen. Damit können zum Beispiel gegebene Fahrpläne auf ihre Korrektheit und Robustheit hin untersucht werden. Diese Simulationen sollen so detailliert wie möglich berechnet werden, damit sie möglichst genau mit der Wirklichkeit übereinstimmen. Andererseits sind die regionalen und nationalen Eisenbahnnetze hochgradig miteinander vernetzt, so dass man möglichst große Netze in einer einzigen Simulation berechnen möchte. Wenn beide Anforderungen in einer Simulation berücksichtigt werden sollen, und genau das ist die Erwartung der Simulations-Anwender, muss das verwendete Simulations-System mit sehr großen Simulations-Modellen umgehen können. Konkret soll in unserem Fall das gesamte Netz der deutschen Eisenbahn Spurplan-genau simuliert werden können: 65000 Kilometer Gleis mit ca. 500000 Gleisabschnitten, über die täglich 35000 Züge fahren.

Die bislang existierenden Systeme können das nicht. Deshalb finanziert das deutsche Forschungsministerium *BMBF* ein Forschungs- und Entwicklungsprojekt unter dem Namen *SIMONE* (*Simulation von Trassen-Slots und Zuglagen in Eisenbahnnetzen*), in dem es darum geht, zwei neue Ansätze in die Eisenbahn-Simulation einzubringen: Constraint-Programmierung und Verteilte Problemlösung.

Constraint-Programmierung ist eine Technologie zur Lösung von Planungs- und Optimierungsproblemen. Dabei wird das zu lösende Problem zunächst deklarativ als *Constraint-Satisfaction Problem* (*CSP*) formuliert, und dann unter Verwendung von generischen *Constraint-Solvern* und ebenso allgemeinen Such-Verfahren gelöst. Diese Technik ist neu in der Simulation und insbesondere der Eisenbahn-Simulation. Sie hilft hier unter anderem, in bisherigen Verfahren gegebene *Deadlock*-Schwierigkeiten aufzulösen. Dies gestattet es, Simulationen schneller als bisher zu berechnen.

Weil die abstrakten Simulations-Modelle für große Eisenbahnnetze gerade bei Verwendung der Constraint-Programmierung schnell extrem groß werden können, muss die Rechen-Struktur, auf der die Simulation ausgeführt wird, hochgradig skalierbar sein. Monolithische, serielle Computer sind aber nur sehr

begrenzt skalierbar. Besser sind da schon Parallel-Rechner, noch besser aber verteilte Rechen-Systeme bzw. Verteilte Problemlösung.

Dieser Ansatz ist freilich nicht neu, sondern fast so alt wie die Informatik selbst. Während aber serielle Programmierung sehr gut erforscht ist und breit eingesetzt wird, gilt das für parallele oder gar verteilte Systeme nicht. Zu groß sind immer noch die Schwierigkeiten, die bei Entwurf und Umsetzung solcher Systeme zu bewältigen sind. Allerdings gewinnt das Thema in letzter Zeit immer mehr Aufmerksamkeit, vor allem infolge der Verbreitung des Internet und der zunehmenden Verfügbarkeit von Möglichkeiten, solche Systeme zu konstruieren und zu realisieren. In der Eisenbahn-Simulation ist das aber noch völlig neu.

Ich entwerfe und realisiere in der vorliegenden Arbeit einen Algorithmus, DRS (*Distributed Railway Simulation*), der lokale Constraint-basierte Simulatoren in einem kooperativen, verteilten Prozess eine global korrekte Simulation berechnen lässt.

1.2 Angestrebte Ergebnisse

Hier soll ein Algorithmus entworfen werden, der es erlaubt, sehr große Eisenbahn-Simulationen in annehmbarer Zeit zu berechnen.

Dieser Algorithmus, DRS, soll lokale Simulationen iterativ global konsistent machen. Er soll also nicht selbst unmittelbar die Simulation berechnen, sondern vorhandene Simulationsverfahren in einen kooperativen Prozess einbinden, so dass die Ergebnisse der lokalen Simulationen schließlich in ein global korrektes Ergebnis überführt werden.

Zugleich soll der Algorithmus DRS möglichst geringe Anforderungen an die lokale Simulation stellen. Zum einen ist dadurch die Entwicklung des lokalen Simulators möglichst unabhängig von der des verteilten Systems DRS. Zum anderen kann man dadurch DRS prinzipiell leichter mit anderen lokalen Simulatoren als dem im Moment vorgesehenen verwenden.

Per DRS sollen sehr komplexe Eisenbahnnetze simuliert werden können. Dazu muss DRS gut skalieren: der Algorithmus soll speziell für große Probleme mit sehr vielen Rechenknoten gut funktionieren, die Laufzeit soll durch die Hinzunahme zusätzlicher Knoten möglichst linear abnehmen.

Bei verteilten Algorithmen ist oft nicht klar, ob sie bezüglich der zu berechnenden Funktion korrekt sind, oder auch nur, ob sie in jedem konkreten Anwendungsfall terminieren. In dieser Arbeit soll theoretisch nachgewiesen werden, ob DRS über diese Eigenschaften verfügt.

Neben dem abstrakten Algorithmus soll hier auch eine konkrete Realisierung entstehen. Dieses Ziel ist nicht zuletzt eine Vorgabe des Forschungs- und Entwicklungs-Projekts SIMONE.

Aus der verteilten Struktur des Algorithmus ergeben sich einige spezielle Anforderungen an das zu realisierende System. Vor allem muss das System performant sein: Kommunikation und lokale Berechnungs- und Verwaltungsfunktionen sollen schnell sein.

Das System muss in zweierlei Hinsicht heterogen sein: Es muss unterschiedliche Implementierungen des lokalen Simulators einbinden können, und es muss auf unterschiedlicher Hardware laufen. Wir wollen für die Berechnung der Simulation ja möglichst viel Rechenleistung nutzen. Und unserer Ansicht nach bekommt man für eine gegebene Menge Geld die maximale Rechenleistung,

wenn man Standard-Hard- und Software nutzen und möglichst viel vorhandene Kapazität einbinden kann.

Standard-Komponenten haben den Nachteil, dass die Kommunikation zwischen Rechenknoten relativ zeitaufwändig ist. Deshalb soll die Verteilung der Informationen im gesamten System so angelegt sein, dass Kommunikation vermieden wird. Außerdem soll die Umgebungs-abhängige Konfiguration der einzelnen Komponenten zur Einbindung in ein laufendes System möglichst wenige Einstellungen benötigen.

Das System soll, auch wenn es zunächst nur prototypischen Charakter hat, sehr stabil sein. Das ist vor allem wegen der verteilten System-Struktur besonders wichtig: Dadurch sind besonders viele – auch heterogene – Komponenten am System beteiligt, wodurch sich die Störungsanfälligkeit insgesamt zunächst erhöht. Das System soll dann unter anderem den Ausfall einzelner Komponenten verkraften können.

Schließlich benötigen wir einen Entwicklungsprozess, der die Entwicklungs--Arbeit an dem System vereinfacht. Dazu gehören automatisierte Abläufe und die Möglichkeit, Teile des Systems zur Laufzeit austauschen zu können.

Nochmal zusammenfassend: Der Algorithmus DRS soll lokale Simulationen iterativ global konsistent machen, wenig Anforderungen an die lokalen Simulationen stellen, gut skalieren und theoretisch fundiert sein. Die Realisierung soll performant sein, heterogene Standards unterstützen, eine optimale Informationsverteilung haben, sehr stabil sein und einen eigenen verteilten Entwicklungsprozess besitzen.

1.3 Wissenschaftlicher Beitrag

Dieser Arbeit ging eine intensive Beschäftigung vor allem mit Problemen der Constraint-Programmierung voraus. Auch wenn manche Vorarbeiten nur mittelbar mit der vorliegenden zu tun haben, wäre letztere doch ohne die anderen nicht möglich gewesen. Insgesamt konnten wir eine ganze Reihe wissenschaftlicher Beiträge ableiten:

In [AS99b][AS99a][AS97] veröffentlichten wir ein iteratives, Constraint-basiertes Verfahren zur Erzeugung von Dienstplänen für Schichtdienst zum Beispiel auf Krankenstationen.

In [ABC+99] konnten wir eine neue Multi-Agenten-Plattform für Telekommunikationsdienste vorstellen, die es Software-Agenten ermöglicht, wie auf einem echten Markt miteinander zu handeln.

In [RS00a][RS00b] haben wir uns mit getypten CHR-Programmen und automatischer Typ-Inferenz beschäftigt.

In [Sch00] habe ich ein Labeling-Verfahren, *Reduce-To-The-Max*, für Ressourcen-Optimierungs-Probleme vorgestellt, das Symmetrien in der Reihenfolge der Belegung von Arbeits-Aufgaben auf Maschinen erkennt und dadurch Suchschritte erheblich reduziert.

In [SR00a] konnten wir zeigen, wie sich das berühmte verteilte Constraint-Such-Verfahren *Weak-Commitment-Search* verallgemeinern und damit in ein Propagations-Verfahren erweitern lässt.

In [SR00b] plädierten wir für die Einführung von *Threads* in Prolog-Systeme. Wir begründeten dies mit der Notwendigkeit von Parallelverarbeitung in modernen Programmsystemen und beschrieben eine einfache aber mächtige Prolog-Thread-Schnittstelle.

In [SGO02] haben wir unsere Arbeiten an einem System zur Zuordnung von Assistenzärzten auf Klinik-Arbeitsplätze veröffentlicht. Unser Ansatz berücksichtigt ganz besonders die Fähigkeiten der Ärzte und die Anforderungen der Arbeitsplätze.

In [Sch02a] konnte ich erste Ideen zur Verteilten Eisenbahn-Simulation veröffentlichen.

In [RS02] haben wir erste Ansätze zum *Asynchronen Constraint-Lösen* veröffentlicht, einem Verfahren zum dynamischen, verteilten Lösen von Constraint-Problemen.

In [SR02b] veröffentlichten wir einen Ansatz zur Objekt-Orientierten Constraint-Programmierung: *POOC*, eine abstrakte Schnittstelle zu diversen Constraint-Solver-Implementierungen. Diese Arbeit hat jüngst (in [Wal03]) prominente Aufmerksamkeit erfahren.

In [SR02a] haben wir einen Ansatz zur Objekt-Orientierten Modellierung von Constraint-Problemen im Prolog-System SICStus vorgestellt.

In [HMSW03] stellten wir unseren neuen, mächtigen und äußerst effizienten Constraint-Solver *FIRSTCS* vor.

In [Sch03] schließlich habe ich die jüngsten Entwicklungen zur Verteilten Eisenbahn-Simulation skizziert.

1.4 Aufbau der Arbeit

Diese Arbeit ist in vier große Teile gegliedert: Einführung, Algorithmus DRS, Realisierung und Ergebnisse.

Dies ist eine Querschnittsarbeit, sie findet in relativ weit entfernten Gebieten statt: Verteilte Problemlösung, Constraint-Programmierung, Simulation. Um die Arbeit dennoch in sich geschlossen zu halten, bietet Teil I [S. 1] einführende Übersichten über die Gebiete Verteilte Problemlösung, Constraint-Programmierung, Simulation und die konkrete Anwendung der Eisenbahn-Simulation an. Damit entsprechen Rang und Folge der Einführungskapitel dem Titel der Arbeit. Die Einführungskapitel selbst sind alle nach folgendem Schema gegliedert: Konzept (*was*), Motivation (*warum*), Probleme und Lösungsansätze (*wie*), Anwendungen (*wofür*), Einordnung dieser Arbeit (*wo*).

In Teil II [S. 51] wird der Algorithmus DRS schrittweise sowohl anschaulich als auch formal beschrieben. Darauf aufbauend werden die wesentlichen Eigenschaften des Algorithmus – Korrektheit und Terminierung – formal bewiesen.

Teil III [S. 97] beschreibt die konkrete Umsetzung des Ansatzes. Design und Abläufe sind dabei in UML (*Unified Modeling Language*, [BRJ99][RJB99]) formuliert. Für die Kontrolle des verteilten Algorithmus ist ein Beweis der Korrektheit der Implementierung angegeben.

In Teil IV [S. 159] schließlich werden Ansatz und Umsetzung von DRS empirisch evaluiert. Ferner finden sich dort eine Zusammenfassung der Ergebnisse dieser Arbeit und ein Ausblick auf mögliche Weiterentwicklungen.

2 Verteilte Problemlösung

Verteilte Problemlösung ist ein moderner Ansatz zur Behandlung besonders komplexer Rechen-Probleme. Er nutzt die Möglichkeiten der dezentralen Verteilung von Rechen- und Speicherkapazitäten in Netzwerken und der gemeinsamen, kooperativen Nutzung dieser Ressourcen. Dabei gilt es allerdings, einige Schwierigkeiten zu überwinden. Im Folgenden werden die existierenden Ansätze und Anwendungsmöglichkeiten beschrieben.

> Verteilte Algorithmen sind Verfahren, die dadurch charakterisiert sind, dass mehrere autonome Prozesse gleichzeitig Teile eines gemeinsamen Problems in kooperativer Weise bearbeiten und der dabei erforderliche Informationsaustausch ausschließlich über Nachrichten erfolgt. Derartige Algorithmen kommen im Rahmen von verteilten Systemen zum Einsatz, bei denen kein gemeinsamer Speicher existiert und die Übertragungs- und Bearbeitungsdauer von Nachrichten i.a. nicht vernachlässigt werden kann. Die hierdurch bedingte ausschließlich partielle Systemsicht der einzelnen Komponenten und die Abwesenheit einer gemeinsamen Zeitbasis führt, zusammen mit anderen typischen Eigenschaften der Verfahren wie Nebenläufigkeit und Nichtdeterminismus, zu schwierigen aber interessanten Problemen.

So charakterisierte Mattern schon 1989 in [Mat89, 1.0] verteilte Algorithmen und Systeme.

Das ist keine formale Definition, sondern eher eine pragmatische. Formale Definitionen verteilter Systeme erweisen sich als schwierig oder unzureichend. Dietsch [Die94] beispielsweise weist, basierend auf LeLann [LeL79], darauf hin, dass für eine genaue Definition verteilter Systeme anhand physischer Eigenschaften letztlich Schwellwerte für *räumliche Entfernung, Verzögerungszeit, Durchsatz usw.* angegeben werden müssten. Schließlich gebe es in jedem Rechensystem Netz-artige Verbindungen zwischen unterschiedlichen Hardware-Komponenten. Umgekehrt könne von einem logischen, also nicht-physischen, Standpunkt aus auch das Zusammenwirken unterschiedlicher Programme in einem Rechner als kooperatives verteiltes System betrachtet werden. Auch so also ist eine klare Abgrenzung kaum möglich. Dietsch [Die94] entwirft deshalb ein sehr vereinfachendes Bild verteilter Systeme: siehe Abbildung 2.1 [S. 7].

Gegenüber zentralen sequentiellen Systemen haben parallele und verteilte Systeme vor allem den Vorteil des Performanzgewinns. Hwang und Briggs definieren [HB85, 1.1.2]:

> Parallel processing is an efficient form of information processing which emphasizes the exploitation of concurrent events in the computing process. Concurrency implies parallelism, simultaneity, and pipelining. [...] Concurrent events are attainable in a computer system at various processing levels. Parallel processing demands concurrent execution of many programs in the computer. It is in contrast to sequential processing. It is a cost-effective means to improve system performance through concurrent activities in the computer.

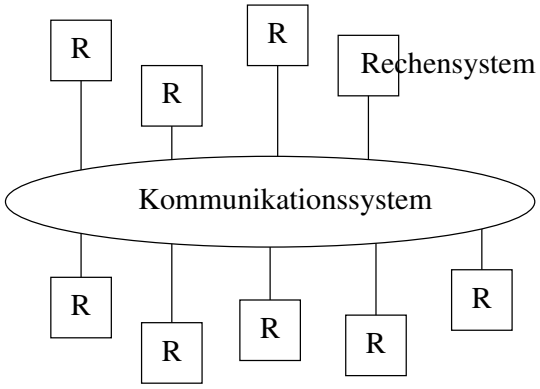


Abbildung 2.1: [Die94] *Das Verteilte Rechensystem: Eine verbreitete Übereinkunft.*

Im Gegensatz zur verteilten Problemlösung bezieht sich Parallelverarbeitung explizit auf *einen* Computer. [HB85, 1.1.2] weiter:

> [We] concentrate on parallel processing with centralized computing facilities. Distributed processing on physically dispersed and loosely coupled computer networks is beyond the scope [...], though a high degree of concurrency is often exploitable in distributed systems.

Hwang und Briggs geben in [HB85] eine wegweisende Einführung in Parallelverarbeitung und zeigen verschiedene technische Realisierungen auf. Matterns Überblick in [Mat89] ist ebenso herausragend und vielzitiert, konzentriert sich aber auf verteilte Algorithmen. Zomaya et al. dagegen geben in [Zom96] einen sehr breiten Überblick über verteilte Systeme. Den entsprechend aktuellsten Überblick geben Tanenbaum und van Steen in [TvS03].

2.1 Motivation

Performanzgewinn ist *ein* Vorteil verteilter Systeme. Es gibt aber noch andere Vorteile, insbesondere auch solche, die verteilte Systeme den parallelen voraus haben. Mattern [Mat89, 1.1] nennt einige davon und bezeichnet sie als *potentielle Fähigkeiten*, solche also, die verteilte Systeme nicht automatisch haben, sondern die in der Regel erst realisiert werden müssen. Alle diese Punkte sind im System DRS verwirklicht und zeigen sich in der Realisierung als Vorteile (siehe Kapitel 24, S. 155).

Performanzgewinn

Wenn mehr als eine Recheneinheit zur Verfügung steht – und erst dann kann man ja von einem verteilten System sprechen –, können mehr als eine Aufgabe echt gleichzeitig bearbeitet werden bzw. die eine zu bearbeitende von mehreren Rechnern gleichzeitig. Zusätzlich zur Verarbeitungskapazität steht in der Regel im verteilten System ein vielfaches an Speicherkapazität zur Verfügung, was in der Praxis zu weiteren Zeitersparnissen führen kann.

Hwang und Briggs motivieren in [HB85] parallele oder verteilte Systeme mit der damit möglichen Ausnutzung erweiterter Rechenkapazität. Ringwelski

schreibt in [Rin03, 1.1] zu diesem Thema:

> Wenn parallele [...] Verarbeitung möglich ist, können Probleme gelöst werden, die zu groß sind, um in monolithischen, deterministischen Systemen bearbeitet zu werden.

Und Hannebauer motiviert in [Han01, 2.3.1] entsprechend:

> Typically, a single entity only has limited time of attention. The work-load that can be assigned to an entity either by someone else or by itself is not infinitely scalable. Though today's computers increase in compute and storage capabilities quite quickly, the core problem of limited local resources will most likely remain since it resides on the layer of the used computation models.

Performanzgewinn lässt sich messen. Man unterscheidet dabei gerne die Größen *Speedup* und *Scaleup*. *Speedup* [SWSM96] ist der zeitliche Gewinn, den man erhält, indem man ein gegebenes Problem anstatt auf einem Knoten auf mehreren Knoten rechnet. *Scaleup* [DG92] hingegen bedeutet den Gewinn, den man durch die verteilte Berechnung eines n-fach größeren Problems erhält. Etwas formaler: Sei $t(x, y)$ die Zeit für die Lösung eines Problems der Größe x auf y Knoten. Damit ist $t(1, 1)$ die Zeit für die Lösung eines Problems a auf einem Knoten, $t(n, 1)$ die Zeit für die Lösung eines Problems $n \times a$ auf einem Knoten, und $t(n, n)$ die Zeit für dasselbe Problem auf n Knoten. Damit ist der Speedup $p := t(1, 1)/t(n, 1)$ und der Scaleup $c := t(1, 1)/t(n, n)$. Im optimalen Fall gilt $p = n$ und $c = 1$.

Fehlertoleranz und Ausfallsicherheit

Zunächst sind verteilte Systeme viel fehleranfälliger als klassische, zentrale: Weil in einem verteilten System naturgemäß viel mehr Komponenten beteiligt sind als in einem zentralen monolithischen, können natürlich auch viel mehr Komponenten fehlerhaft sein, Fehler erzeugen oder sogar ganz ausfallen. Deshalb ist es für verteilte Systeme viel wichtiger als für klassische, Redundanzen einzubauen.

In verteilten Systemen ist dann aber anders als in klassischen schon strukturell die Möglichkeit der Redundanz gegeben. In der Regel haben verteilte Systeme sogar eine sehr hohe Redundanz: viele ähnliche Komponenten übernehmen ähnliche Aufgaben. Wenn eine der Komponenten ausfällt oder Fehler zeigt, kann eine andere Komponente deren Aufgaben übernehmen. Erforderlich sind deshalb *Orts- und Ausfalltransparenz* [Dal94].

In einem zentralen System ist das in der Regel nicht der Fall: wenn es beispielsweise vollständig ausfällt, sind alle rechnenden Prozesse ergebnislos beendet.

Inkrementelle Erweiterbarkeit

Verteilte Systeme können meist sehr leicht erweitert werden: Neue Rechenknoten können hinzugefügt werden, neue Geräte wie spezielle Drucker oder Speichermedien können in das System eingebunden werden. Durch die vorhandene Redundanz lassen sich oft sogar während des Betriebs Teile des Systems ersetzen: alte Rechner durch neue, die Software durch neue Implementierungen.

Örtliche Bereitstellung von Rechenleistung

Weil verteilte Systeme immer durch ein Kommunikationssystem untereinander verbunden sind und über Mechanismen verfügen, Aufgaben zu verteilen, kann

die Rechenleistung des gesamten Systems oder von Teilen davon an vielen Orten bereitgestellt werden. Die Leistung des Systems ist fast unabhängig vom Ort und kann deshalb überall erbracht werden.

Dieser Punkt gewinnt in letzter Zeit zunehmend an Bedeutung:

- im Internet ist der komplette Datenbestand aller Webseiten an jedem Internet-Terminal auf der ganzen Welt auf etwa die gleiche Weise verfügbar,
- für extrem komplexe Rechenaufgaben werden immer häufiger verteilte *Grid-Systeme* (Abschnitt 2.3.2, S. 14) verwendet und ersetzen damit zentrale Rechenzentren.

Datensicherheit

Hannebauer [Han01] weist darauf hin, dass in verteilten Systemen auf *soziale Grenzen* Rücksicht genommen werden kann. Wenn in einem solchen System zwei Parteien a und b beteiligt sind, kann es sein, dass a nicht möchte, dass b Einblick in seine Daten oder Strukturen hat. In einem verteilten System können die Daten von a und b auf unterschiedlichen Rechnern gespeichert sein und somit leicht vor dem Zugriff anderer geschützt werden.

Wie die Fehlertoleranz hat auch diese Medaille zwei Seiten: an der Verarbeitung im verteilten System ist immer Kommunikation beteiligt. Diese Kommunikation aber ist stärker als ein zentrales System Angriffen von außen ausgesetzt. Heutzutage wird als Kommunikationssystem in verteilten Systemen häufig das Internet eingesetzt [MH01], wo diese Problematik besonders groß ist. Ringwelski [Rin03] weist in diesem Zusammenhang darauf hin, dass *Privacy* leichter als *Security* zu haben ist.

Übrigens geht es nicht nur um den Schutz von *Daten*, sondern auch von *Strukturen* oder Verfahren. Z.B. könnten in einem virtuellen Markt Software-*Agenten* (siehe auch Abschnitt 2.3.2, S. 14) Preise für Dienste aushandeln. Jeder Agent könnte dabei eine eigene Verhandlungs-Strategie verfolgen, die den anderen nicht bekannt sein darf. Genauso könnte in einem verteilten Terminplanungs-System [Rin03] ein Teilnehmer a einen anderen Teilnehmer b als unwichtig einschätzen und ihm nur selten Besprechungstermine einräumen. Dass das auf Grund der Einschätzung von a passiert, darf b natürlich nicht wissen.

Kosteneffizienz

Die jüngste Entwicklung zeigt, dass Parallelrechner wie überhaupt sehr schnelle Zentralrechner extrem teuer sind. So kostet zur Zeit ein Hochleistungsrechner *Sun Fire 15K Server* mit 72 Prozessoren laut Preisliste mehr als 2,3 Mio. Euro [Sunb]. Ein Netzwerk aus ebenso vielen etwa proportional dazu ausgestatteten PCs kostet weniger als ein Zehntel davon.

2.2 Probleme

Neben all den Vorteilen, die verteilte Systeme haben, gibt es natürlich auch eine Reihe von Nachteilen. Wieder Mattern [Mat89, 1.1] weist auf die meisten davon hin, Peleg [Pel00] auf einige andere.

Informationsdefizit

In einem verteilten System hat in aller Regel jeder Knoten des Netzes nur eine eingeschränkte Sicht auf das Gesamtsystem. Mattern spricht hier von *lokal unscharfer Information*. Theoretisch ließe sich zwar auch immer und überall

vollständige Information herstellen, das würde aber immensen Kommunikationsaufwand bedeuten. Dieser Kommunikationsaufwand wiederum würde das ganze System lahmlegen.

Inkonsistenzen

Aus der lokalen Unschärfe folgt unmittelbar, dass keine zwei Komponenten die gleiche Sicht auf das System haben. Das führt sogar oft zu Inkonsistenzen zwischen den lokalen Informationen unterschiedlicher Komponenten. Zum Beispiel haben in verteilten Systemen selten zwei Komponenten die gleiche System-Zeit [CDK01, 10].

Nichtdeterminismus und Determiniertheit

Herrtwich und Hommel definieren [HH89, 1.1]:

> Ein Programm, dessen Ablauf eindeutig vorherbestimmt ist, nennt man deterministisch (*deterministic*). Ein Programm, das bei gleichen Eingaben gleiche Ausgaben produziert, bezeichnet man als determiniert (*determined*).

Durch Nebenläufigkeit von Berechnungen und Kommunikation und unterschiedlichen Eigenschaften der beteiligten Hardware kommt es in verteilten Systemen oft zu Nichtdeterminismus: Nehmen wir als Beispiel drei Komponenten a, b, c und Nachrichten l, m, n. a schicke immer m an b und gleich anschließend l an c. b schicke immer sofort nach Erhalt von m eine Nachricht n an c. Jetzt kann es sein, dass in c zuerst l eintrifft und dann n. Es kann aber auch sein, dass – durch unterschiedliche Laufzeiten im Netzwerk – in c zuerst n und dann l eintrifft. Wenn c nun sowohl auf l als auch auf n unmittelbar reagiert und beispielsweise eine entsprechende Berechnung beginnt, erzeugt dieses Szenario in c einen Nichtdeterminismus.

Oft führt Nicht-*Determinismus* zu Nicht-*Determiniertheit*. Meist aber ist das nicht akzeptabel, man möchte, dass das Verfahren – in diesem Fall der verteilte Algorithmus – bei denselben Eingaben dieselbe Ausgabe, dasselbe Ergebnis produziert. Weil verteilte Systeme inhärent nichtdeterministisch sind, müssen die Algorithmen entsprechend Sorge tragen.

Terminierung

Viele parallele oder verteilte Algorithmen lösen ein einziges quasi globales Problem kooperativ. Während des Lösungsprozesses haben die im verteilten System beteiligten Komponenten jedes ein eigenes Teil einer Gesamt-Lösung. Da oft keine zentrale Kontrolle existiert (siehe nächsten Punkt), existiert auch nirgends eine konsistente globale Sicht auf das System [Mat89, 3.1]. Insbesondere *weiß* keine beteiligte Komponente, ob eine global konsistente Lösung vorliegt bzw. der Algorithmus beendet ist.

Das kooperative Feststellen eines Endzustandes ist ein Problem für sich. Folgendes Beispiel aus [Mat89, 4.4] soll das illustrieren: Gegeben Komponenten a, b, z. z sei hier eine zentrale Instanz, die sich um die Terminierung kümmert, und zwar per einfachem Zählverfahren. Immer wenn a eine Nachricht an b schickt, sendet a die Meldung an z, dass es eine Nachricht verschickt hat. Und wenn a eine Nachricht erhält, meldet a an z den Erhalt einer Nachricht. b verfährt genauso. z versucht nun, anhand einfachen Zählens der gesendeten und empfangenen Nachrichten festzustellen, wann keine Nachrichten mehr im System unterwegs sind. Das ist quasi eine Grundvoraussetzung für Terminierung.

Es kann folgendes passieren: a schickt b eine Nachricht, und gleichzeitig schickt b eine an a. Beide melden an z. Jetzt kommen aber die Meldungen bei z in folgender Reihenfolge an: a : *gesendet*, a : *erhalten*, b : *gesendet*, b : *erhalten*. Zwischen a : *erhalten* und b : *gesendet* ist für z die gesamte Anzahl gesendeter und empfangener Nachrichten ausgeglichen, z könnte also hier – aufgrund anderer Informationen die vor a : *gesendet* noch nicht vorlagen – Terminierung feststellen, obwohl noch Nachrichten im System unterwegs sind.

Neben dem technischen Problem, einen globalen Endzustand *festzustellen*, ist es bei verteilten Algorithmen oft nicht klar, ob ein solcher überhaupt irgendwann *eintritt*: Typischerweise arbeiten verteilte Algorithmen ja nebenläufig und kooperativ. Und solche Prozesse können sich leicht unendlich aufschaukeln. Für verteilte Algorithmen muss man also oft explizit zeigen, dass sie konvergieren und endlich terminieren. Einen solchen Beweis führe ich für den Algorithmus DRS in Teil II [S. 51].

Kontrolle und Verantwortlichkeiten

Typischerweise fehlt in verteilten Systemen eine zentrale Kontrollinstanz, vor allem weil eine solche ein Performanz-Problem darstellen würde: Die Kommunikation mit der Zentrale wäre ein Engpass, der *alleine* die Performanz des Gesamt-Systems beeinträchtigen würde. Außerdem widerspricht die Existenz einer einzigen Kontrollinstanz dem Bedürfnis nach Redundanz.

Deshalb ist die Kontrolle des Gesamt-Systems meist verteilt auf mehrere Komponenten, wenn nicht sogar auf alle. Die Wahl der optimalen Kontrollstruktur für ein gegebenes Berechnungsproblem ist ein Problem für sich.

Konfiguration

Selbst wenn die grundsätzliche Struktur der Verantwortlichkeiten und der Kontrolle gegeben ist, muss oft noch eine konkrete Konfiguration gefunden werden, wie das zu berechnende globale Problem auf die zur Verfügung stehenden Rechnerknoten verteilt wird. In manchen Fällen ist dieses sogar ein dynamisches Problem, ändert sich also während der Laufzeit des Programms [Han00]. Zusätzlich muss manchmal die Netz-Topologie, also das Verbindungsschema zwischen den Rechnerknoten, optimal bestimmt werden [PD00].

Kommunikation

Jedes verteilte System benötigt ein Kommunikationssystem. Dieses wird auf der untersten Übertragungsebene [TvS03] durch ein Rechnernetz realisiert. Auf höheren Ebenen gibt es unterschiedlichste Übertragungs-Protokolle. Die Probleme die damit zusammenhängen, sind enorm vielfältig [Fra86][PD00][Tan00]. Glücklicherweise gibt es dafür heute viele Standardlösungen. Allerdings ist nicht jede gleichermaßen für alle Probleme geeignet. Hier muss für das gegebene Problem die richtige Lösung gefunden werden.

Algorithmen und Programmierung

Dies ist sogar einer der wichtigsten und schwierigsten Punkte. Ein Algorithmus, der für einen sequentiellen Rechner entworfen wurde, kann nicht so ohne weiteres auf mehrere Rechner übertragen werden. Der Algorithmus muss in der Regel vollkommen neu entwickelt werden. Es gibt zwar Ansätze, existierende Algorithmen durch spezielle Compiler automatisch zu parallelisieren, diese sind aber laut Rauber und Rünger [RR00a, 3.4] bisher wenig erfolgreich.

Manche Algorithmen sind gar nicht parallelisierbar. Andere aber sind von

Natur aus parallel und fast leichter auf ein verteiltes System zu implementieren als auf ein lokal sequentielles. Verteilte Programme haben gegenüber (lokal) parallelen auch noch das Problem, dass kein gemeinsamer Speicher existiert, alle Kommunikation zwischen Programmteilen über relativ langsame Nachrichtenkanäle laufen muss.

In jedem Fall sind verteilte Programme ganz anders strukturiert als sequentielle und stellen deshalb andere Ansprüche an Programmierer und Programmierwerkzeuge. Peleg meint gar [Pel00, 1.3]:

> A new style is needed for describing distributed algorithms.

Heterogenität
Moderne verteilte Systeme sind meist sehr heterogen: die beteiligten Rechner haben unterschiedliche Fähigkeiten, benutzen unterschiedliche Betriebssysteme oder Netzwerkanbindungen. Oft müssen auch existierende *Legacy Systems* (engl. Vermächtnis oder Altlast) eingebunden werden, wie Datenbanken, Verwaltungs- oder Planungs-Systeme.

Bewertung
Will man ein gegebenes Verfahren bewerten, also z.B. die Laufzeit im Verhältnis zur Problemgröße messen, muss eine wiederherstellbare und vergleichbare Rechen-Situation geschaffen werden. Das ist schon für zentrale sequentielle Algorithmen nicht ganz leicht. Besonders schwierig aber ist es im verteilten Kontext. Vor allem Nichtdeterminismus und die außerordentliche Dynamik verteilter Systeme machen hier zu schaffen.

2.3 Ansätze

Verteilte Systeme haben also auch eine Reihe von Nachteilen. Und doch sind sie für viele algorithmische Probleme derart gut geeignet, dass dort die Vorteile die Nachteile mehr als aufwiegen. Außerdem sind die Nachteile, wie wir wissen, *schwierig aber interessant* [Mat89]. Forschung und Entwicklung haben deshalb in den letzten 25 Jahren viele Lösungen entworfen, die zur Verbreitung verteilter Systeme in der Informatik geführt haben.

2.3.1 Theorie

Für Entwurf und Verifikation verteilter Systeme sind einige formale Ansätze entstanden. Ich gebe eine Übersicht über die wichtigsten Vertreter. Bei Krishnamurthy [Kri89] finden sich einige davon weiter vertieft.

Communicating Sequential Processes
Hoare entwarf in den 80er Jahren die Theorie der kommunizierenden sequentiellen Prozesse (CSP) [Hoa84]. Dabei werden klassische Programmier-Paradigmen um Operatoren zur Kommunikation mit anderen Programmen erweitert. Ein solches Programm läuft dann als *Prozess* auf einem Rechnersystem und kann mit anderen *nebenläufigen* Prozessen kommunizieren. Von Apt gibt es umfangreiche Beweistheorien zur Verifikation von CSP [Apt84]. Praktische Relevanz bekamen die CSP durch die Umsetzung der Konzepte in der Programmiersprache *Occam* [SF88][Hyd98] und deren Implementierung auf *Transputer*-Rechnern.

Logiken

Auch temporale Logiken (i.e. Modal-Logiken mit zeitlichen Operatoren) eignen sich für die oben genannten Zwecke. Kröger schreibt [Krö87]:

> Important goals of mathematical logic are to: (1) provide languages for the precise formulation of propositions, (2) investigate mechanisms for finding out the truth or falsity of propositions.

Er entwirft in [Krö87] eine temporale Aussagenlogik, eine temporale Prädikatenlogik (*First-Order Logic*), und zeigt Anwendungen der Spezifikation und Verifikation von u.a. nebenläufigen Programmen auf.

Etwa zur gleichen Zeit entstand Lamports *Temporal Logic of Actions*, ebenfalls eine temporale Logik, allerdings mit anderen Operatoren als bei Kröger [Lam94b][Lam94a].

Eine interessante Constraint-basierte (siehe Kapitel 3, S. 21) Beweismaschine entwirft Frühwirth unter Verwendung seiner *Constraint Handling Rules* in [Frü94].

Algebren

Parallel zu den logischen entstanden algebraische Theorien nebenläufiger Programme. Basierend auf Milners *Calculus of Communicating Systems* [Mil80], entwickeln und popularisieren dieses Konzept u.a. Hennessy [Hen88], Baeten [BW90][Bae91] und Fokkink [Fok00].

Petri-Netze

Rosenstengel und Winand charakterisieren in [RW91] Petri-Netze wie folgt:

> Die Petri-Netz-Theorie ist eine axiomatisierte mathematische Theorie, [...] vor allem aber eine anschauliche Theorie.

Damit heben sie den wohl wichtigsten Grund hervor, warum Petri-Netze in den letzten Jahren so erfolgreich sind.

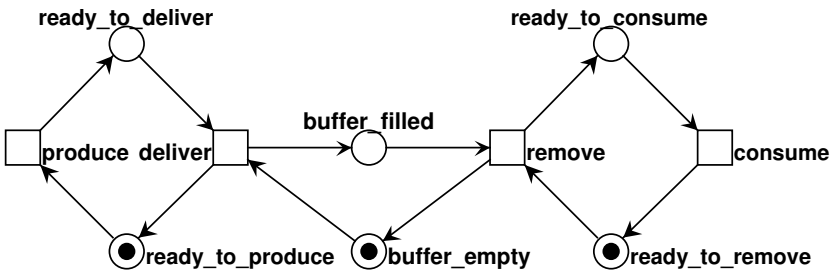


Abbildung 2.2: *Producer/Consumer*-System als Petri-Netz [Rei98, 1.6]

Petri-Netze gehen zurück auf C. A. Petri [Pet62] und beschreiben Zusammenhänge zwischen Zuständen und Zustandsübergängen, z.B. in verteilten Systemen. Abbildung 2.2 [S. 13] zeigt ein Petri-Netz für das berühmte *Producer/Consumer*-System: Ein Erzeuger (links im Beispiel-Netz) produziert Güter, die er über einen Puffer (mitte) an den Verbraucher (rechts) gibt, der diese konsumiert. Der Erzeuger kann sich in den *Zuständen* (dargestellt durch Kreise)

`ready-to-produce` und `ready-to-deliver` befinden, wobei er über die Aktionen (dargestellt durch Quadrate) `produce` und `deliver` von einem in den anderen Zustand gelangen kann. Die Marke im Zustand `ready-to-produce` zeigt den augenblicklichen Zustand des Erzeugers an. Entsprechendes gilt für den Puffer und den Verbraucher.

In dem in Abbildung 2.2 [S. 13] dargestellten globalen Zustand ist nur der Übergang bzw. die Aktion `produce` möglich, alle anderen Aktionen haben nicht genügend Vorbedingungen: Damit eine Aktion a ausgeführt werden kann, müssen alle Vorbedingungen gelten, d.h. alle Kreise, von denen ein Pfeil *nach* a führt, müssen mit einer Marke belegt sein. Außerdem darf kein Kreis, zu dem ein Pfeil *von* a führt, mit einer Marke belegt sein. In Abbildung 2.2 [S. 13] kann also nur die Aktion `produce` ausgeführt werden. Nach Ausführung von `produce` würde der Zustand `ready-to-produce` nicht mehr gelten, die Marke wäre nach `ready-to-deliver` gewandert. Das heißt, der Erzeuger ist anschließend im Zustand `ready-to-deliver`. Dann gelten für die Aktion `deliver` beide Vorbedingungen, die Marken könnten wandern. Anschließend würde gelten `buffer-filled` und wieder `ready-to-produce`.

Dieses ist ein sehr einfaches Beispiel. Ich werde später noch genauer auf Petri-Netze eingehen und mit ihrer Hilfe Eigenschaften der Realisierung von DRS beweisen (siehe Kapitel 20, S. 130).

In der Literatur finden sich zahlreiche Erweiterungen der einfachen Petri-Netze und Anwendungen auf parallele und verteilte Systeme, z.B. [Abe90][RW91] [Bau96][Rei98].

2.3.2 Organisationsstrukturen

Auch für das Problem der optimalen Kontroll- oder Organisations-Strukturen existieren eine Reihe wichtiger Ansätze.

Client/Server

Der wohl ursprünglichste ist der *Client/Server*-Ansatz [Inm93][Ben01]: Ein Server bedient viele Clients. Obwohl dieser Ansatz einige oben genannte Vorteile nicht hat, v.a. Redundanz und Fehlertoleranz, oft auch Performanzprobleme, ist er für viele Anwendungen noch immer in Gebrauch. Vor allem zentrale Datenbanken oder Datendienste auch im Internet arbeiten nach diesem Prinzip. Client/Server ist immer dann leicht zu implementieren, wenn viele Anwender einen gemeinsamen, konsistenten und für alle identischen Datenbestand nutzen wollen.

Cluster

Auch das *Cluster-Computing* [Tur96] ist bereits etwas betagt, aber immer noch beliebt. Cluster werden vor allem dann eingesetzt, wenn es um maximale Rechenleistung geht, für Probleme die gut parallelisiert werden können. Cluster zeichnen sich aus durch einheitliche Hard- und Software-Plattformen und eine einheitliche und sehr schnelle Kommunikationsinfrastruktur. Allerdings sind Cluster dadurch in der Regel auch nur für den einen ihnen zugedachten Zweck geeignet, ein Cluster ist gewissermaßen ein Spezialrechner.

Agenten und DAI

Anfang der 90er Jahre kam die *Agenten*-Technik groß in Mode [DLC91][ABC+99] [Fri00][Han01]. Biologen, Soziologen, Psychologen und Informatiker entwarfen

interdisziplinär vor allem Software-*Agenten*. Das sollen Programme sein, die eigenständig ihre Umwelt wahrnehmen, *pro-aktiv* agieren und kooperativ handeln. Sie sollen sich untereinander nicht mit schlichten Protokollen verständigen, sondern mit echten Sprachen [Fip]. Auf diese Weise sollen ganze Gesellschaften von virtuellen Agenten Märkte bilden oder andere Probleme lösen. Agenten heißen deshalb so, weil sie in der realen Welt einen Herrn haben, in dessen Auftrag sie in den virtuellen Welten agieren. Neben Software-Agenten gibt es auch Hardware-Agenten: intelligente Roboter. Agenten sollen immer *intelligent* sein, weshalb sich für dieselbe Technik auch der Begriff *Distributed Artificial Intelligence* (DAI) oder *Verteilte Künstliche Intelligenz* (VKI) etabliert hat.

Die Agenten-Technik ist, nicht zuletzt wegen der vielen beteiligten Disziplinen, sehr spannend, hat sich aber in der Anwendung noch nicht recht beweisen können. Das kann aber auch an ihrer relativen Jugend liegen.

Peer-to-Peer

Ganz neu ist der Begriff *Peer-to-Peer* (P2P, [Ora01][SFT02]) zwar nicht, doch lassen sich unter ihm die allerneusten Entwicklungen im Bereich der verteilten Systeme zusammenfassen. Auch bei Peer-to-Peer geht es um sehr viele gleichberechtigte Partner in einem Netz. Doch anders als Agenten sind Peers immer Rechner, meist sogar Standard-Rechner, verbunden durch aktuelle Standard-Netzwerke.

Das *Internet* ist ein P2P-System und wohl der Auslöser und wesentliche Ideengeber für die folgenden Entwicklungen. Dort herrscht Gleichberechtigung: Jeder kann für wenig Geld seine Dokumente ins Netz stellen und noch einfacher die der anderen konsumieren [MH01]. Etwas spezieller und quasi Teil des Internet sind Tauschbörsen wie Napster [Shi01] oder Gnutella [Kan01]. In ihnen kann man vor allem Musik-Dateien, aber auch Bilder, Filme, Texte, Programme usw. tauschen. Nicht zuletzt weil sie urheberrechtlich problematisch sind, haben sie in letzter Zeit enorme Aufmerksamkeit erhalten.

Die ersten Ansätze kooperativen Rechnens im Internet waren die Decodierung von RC5- und DES-Schlüsseln ab 1997 [dis]. Damals wurde ein Programm entwickelt, das sich regelmäßig von einem zentralen Server eine kleine Teilaufgabe des Gesamtproblems holt und nach Erfüllung das Ergebnis an den Server zurückschickt. Dieses Programm wurde zum kostenlosen Download ins Internet gestellt und verbreitete sich schnell in der einschlägigen Gemeinde. Es fanden sich sehr schnell viele Leute, die ihre jeweils verfügbare Rechenleistung einem gemeinsamen Rechenprojekt zur Verfügung stellen wollten.

Sehr erfolgreich ist seit 1993 SETI@home [SET][And01]. Es entstand aus der Notlage, dass die NASA der Suche nach Extra-Terrestrischer Intelligenz die Zuwendungen strich. Damals wurde ebenfalls ein Programm entwickelt, das die SETI-Berechnungen auf vielen Computern im Internet ausführen kann. Seither haben sich über 4,6 Mio. Internet-Nutzer an dem Projekt beteiligt und 1,5 Mio. Jahre CPU-Zeit dafür zur Verfügung gestellt.

RC5-Entschlüsslung und SETI@home waren die Ursprünge des *Grid-Computing* [ABG02][FKT02][BG02][Wed02][EUR]. Ein Ausgangspunkt für diesen Ansatz ist die Beobachtung, dass moderne Arbeitsplatz-Computer zu höchstens 20% ausgelastet sind [Wed02]. Generell geht es also beim Grid-Computing darum, die ungenutzten Ressourcen im Internet gemeinsam zu nutzen. Dabei müssen die oben genannten Probleme möglichst allgemein gelöst werden. Es sollte eine gemeinsame Plattform geben, die sowohl von den Anbietern von Rechenka-

pazität als auch von den Nutzern einfach eingesetzt werden kann und jeweils die spezifischen Bedürfnisse befriedigt. So sollten solche Dienste in Zukunft auch abgerechnet werden können. Bald werden nicht mehr nur wissenschaftliche *No-cost* Projekte im Internet laufen, sondern auch kommerzielle, für die die Rechen-Anbieter natürlich Geld haben wollen.

2.3.3 Kommunikation

Technisch benötigt man für ein verteiltes System auch immer ein verbindendes Rechnernetz [Tan00]. Rechnernetze sind wirklich eine Wissenschaft für sich. Peterson und Davie beispielsweise beschäftigen sich in [PD00] mit Netztopologien, Paketvermittlung, Inter-Networking (Hierarchien sehr unterschiedlicher Netze), Protokollen, Fehlertoleranz, Datensicherheit. Franck gibt in [Fra86] Lösungen an für Betriebs- und Übertragungsarten, Adressierung (Namensgebung, Routing), Konzentrieren und Multiplexen, Transportzeit und Durchsatz, Fehlererkennung und Fehlerreaktion, Flusskontrolle.

Glücklicherweise gibt es für Anwender von Rechnernetzen preiswerte Standardlösungen, sodass man nicht alle Probleme selbst lösen muss. Wichtig ist aber schon zu wissen, warum eine Lösung besser sein kann als eine andere. Auf lokaler Ebene bieten die Standardlösungen (*Local Area Network*, LAN) Übertragungsgeschwindigkeiten von 10 bis 1000 Megabit pro Sekunde, je nach Aufwand und je nachdem ob Leitungs- oder Funkbasiert. Für große Entfernungen (*Wide Area Network*, WAN) gibt es entsprechend Lösungen von einigen Kilobit pro Sekunde bis zu einigen Megabit. Diese Übertragungsgeschwindigkeit ist aber nur der maximal mögliche *Durchsatz*. Für eine konkrete Anwendung kann auch die *Latenz* des Mediums, also die minimale Zeit für die Übertragung einer Information [CDK01, 3.1], eine wichtige Rolle spielen. Viele Anwendungen nämlich müssen sehr viele Nachrichten schicken, die aber nicht sehr groß sind. Und genau dafür ist eine kurze Latenz notwendig. Die klassischen Netzwerke haben aber genau dort ihren wesentlichen Nachteil. Besser geeignet für die Übertragung sehr kurzer Nachrichten ist beispielsweise Myrinet [Myr], das deshalb oft in speziellen Clustern zum Einsatz kommt. Bei den neusten P2P-Ansätzen hat man bzgl. des Netzwerks in der Regel ohnehin keine Wahl, es werden ja Rechner benutzt, die irgendwie im Internet zur Verfügung stehen.

Das Netzwerk ist als Übertragungsmedium quasi die Hardware-Seite der Kommunikation. Software-seitig kann die Anwendung relativ direkt Nachrichtenkommunikation verwenden. Auch dazu gibt es System-übergreifende Standards wie *Sockets* oder *Message Passing Interface* (MPI) [TvS03].

In der Regel werden aber noch weit abstraktere Protokolle benutzt, etwa *Remote Procedure Call* (RPC, z.B. Java RMI [RMI]), *Common Object Request Broker Architecture* (Corba, [TvS03]), *Parallel Virtual Machine* (PVM, [RR00a] [PVM]) oder *Distributed Shared Memory* (DSM, [CJ97]). Diese Techniken unterscheiden sich weniger in der Performanz – da sind sie der Nachrichtenkommunikation über Sockets oder MPI ohnehin unterlegen –, sondern vielmehr in der Struktur und Anwendung. RPC erlaubt einem Anwendungsprogramm, Methoden eines anderen Programms aufzurufen, praktisch so, als wäre diese Methode Teil des eigenen Prozesses. Corba zielt auf System-übergreifenden Objekt-Zugriff: damit können verschiedene Rechner mit unterschiedlichen Betriebs- und Anwendungssystemen jeweils auf Objekte der anderen zugreifen. PVM bildet aus den beteiligten Rechnern eine einzige virtuelle Maschine. DSM schließlich

erzeugt für alle Rechner einen virtuellen gemeinsamen Speicher. Die verteilten Komponenten haben ja keinen echten gemeinsamen Speicher (auf den alle komplett in gleicher Weise und mit gleicher Geschwindigkeit zugreifen können), durch DSM wird ein solcher simuliert.

2.3.4 Konfiguration

Probleme, die in verteilten Systemen gerechnet werden, haben sehr oft in irgendeiner Art eine Netzstruktur. Beispiel hierfür sind: Verteilte Dateisysteme wie das *Sun Network File System* [Cal00], das verteilte Dokumentensystem *World Wide Web*, paralleles Lösen von Gleichungssystemen [KK99], und parallele [SKK00a] oder verteilte Simulationen.

Um für ein solches Problem und ein konkret gegebenes verteiltes System eine passende Konfiguration zu erhalten, muss das zugrunde liegende Netz partitioniert werden. Dieses Netz hat typischerweise an Kanten und Knoten Gewichte. Eine optimale Partitionierung eines solchen gewichteten Netzes ordnet jedem Knoten genau eine Partition zu, sodass die Knoten-Gewichte gleichmäßig über alle Partitionen verteilt sind und die Gewichte der durchschnittenen Kanten minimal sind. Ein sehr schnelles Verfahren, das gute Ergebnisse liefert, ist *Metis* bzw. die parallele Version *ParMetis* von Karypis, Kumar und Schloegel [KK99][SKK00b][SKK01]. Dieses Verfahren verwenden auch wir: siehe Kapitel 21 [S. 141].

Oft aber ändert sich das im verteilten System zu lösende Problem während der Laufzeit. Dann werden Algorithmen benötigt, die die äußere und auch innere Konfiguration der beteiligten Knoten (z.B. Agenten) dynamisch berechnen und anpassen können. Hannebauer schlägt eine solche dynamische Rekonfiguration vor [Han00], insbesondere für verteilte Constraint-Probleme (siehe Kapitel 3, S. 21).

2.3.5 Sicherheit

Das Thema Daten-Sicherheit ist genau wie das der Kommunikation eine Wissenschaft für sich. Gerade in verteilten Systemen ist dieser Aspekt von großer Bedeutung. Ich werde ihn allerdings – vor allem wegen seiner Komplexität – in dieser Arbeit nicht behandeln und möchte hier nur auf die sehr gute Übersicht zum Thema Sicherheit in verteilten Systemen von Tanenbaum und van Steen in [TvS03, 8] hinweisen.

2.3.6 Enterprise Computing

Ein ganz modernes Schlagwort ist *Enterprise Computing*. Darunter versteht man große unternehmensweite Informationssysteme. In jüngster Zeit entstehen hierfür Werkzeuge zur Entwicklung, Verteilung und zum Betrieb solcher Anwendungen. Die beiden großen konkurrierenden Systeme sind *Java Enterprise Edition* von SUN [J2Eb] und *.NET* [NET] von Microsoft. Sie bieten vor allem Methoden an zum Betrieb von Anwendungen auf Servern (*Application Server*), zur Verteilung und Aufstellung (*Deployment*) von Anwendungen auf diese Server, zur Neu-Entwicklung verteilter Anwendungen, zum Management solcher Anwendungen, zur Anbindung externer Ressourcen und *Legacy Systems*, und zur Einbindung von Web-Diensten [J2Ea]. Das DRS-System benutzt einige dieser neuen Techniken, wie wir später im Teil III [S. 97] noch sehen werden.

2.3.7 Terminierung

Wir haben oben schon gesehen (Abschnitt 2.2, S. 9), dass es in verteilten Systemen nicht trivial ist, einen globalen Zustand festzustellen. Dazu gibt es einige Lösungen: Algorithmen zur Ermittlung eines *distributed Snapshot* oder zur *distributed Termination Detection*. Die berühmtesten stammen von Chandy und Lamport [CL85] und von Mattern [Mat89]. Einen etwas neueren Überblick samt Klassifikation bieten Matocha und Camp in [MC98].

Sehr modern ist der Algorithmus von Mahapatra und Dutt [MD01], den ich hier kurz skizzieren möchte, weil er elegant und anschaulich ist und weil wir ihn in DRS verwenden (siehe Kapitel 20, S. 130). Leider hat dieser Algorithmus keinen eindeutigen Namen, ich nenne ihn zur Abkürzung schlicht DTD. Der Algorithmus, der per DTD terminiert werden soll, heiße A.

Für DTD müssen die Knoten des verteilten Systems in einem Baum angeordnet werden, was man wie folgt machen kann: Zuerst nimmt man aus der Menge der Knoten einen beliebigen als Wurzel. Aus der Rest-Menge nimmt man dann zwei Knoten und fügt diese als *Kinder* des ersten in den Baum ein. Dann nimmt man für jeden dieser beiden wieder zwei aus der Menge und fügt diese jeweils als Kinder in den Baum ein. Und so weiter bis die Menge leer ist. Man erhält einen nicht notwendigerweise ausgeglichenen binären Baum.

Wir gehen im weiteren davon aus, dass jeder Knoten mit jedem kommunizieren kann. Für die einfache Version von DTD kann jeder Knoten den Zustand *busy* oder *idle* haben. Die Knoten kommunizieren nur bzgl. DTD miteinander, nicht bzgl. A! A stößt den Prozess an, wodurch alle Knoten in den Zustand *busy* gehen. Die Knoten rechnen also alle einen Teil (von A) aus und irgendwann stellt jeder für sich fest, dass er fertig ist, und geht in den Zustand *idle*. DTD erfordert nun, dass jeder Knoten, wenn er *idle* ist und keinen Kind-Knoten hat, sofort seinem Vater-Knoten eine `stop`-Meldung schickt. Ein Knoten, der Kind-Knoten hat, schickt genau dann seinem Vater die `stop`-Meldung, wenn er selbst *idle* ist und von allen Kindern die `stop`-Meldung erhalten hat. Wenn dann der Wurzel-Knoten selbst *idle* ist und von seinen Kindern `stop`-Meldungen erhalten hat, sind alle *idle*. Der Wurzel-Knoten meldet das allen anderen mit einer `termination`-Nachricht.

Im allgemeinen Fall dürfen sich die Knoten auch bezüglich A Nachrichten schicken, so genannte *Primary Messages*. Solche Primary Messages sind gewissermaßen Arbeitsaufträge, die ein Knoten einem anderen schickt. Der Empfänger muss dann irgendwie auf diese Nachricht reagieren und deshalb eventuell vom Zustand *idle* in *busy* gehen.

Betrachte folgendes Beispiel: Abbildung 2.3 [S. 19]. Die Knoten 1, 3, 7, 6 und 5 sind bereits *idle*, 0, 2 und 4 rechnen noch. 5 hat 2 bereits die `stop`-Meldung geschickt. Jetzt (a) schickt 4 der 5 eine Primary Message, also eine Nachricht bezüglich A. Weil 5 dadurch neue Arbeit hat, geht er in den Zustand *busy*. Nun (b) muss 5 der 2 eine `resume`-Meldung schicken, um anzuzeigen, dass er nicht mehr *idle* ist. 2 aber ändert den Zustand nicht, er rechnet ja ohnehin. Weil 2 der erste in dieser Kette ist, der seinen Zustand nicht ändert, schickt er (c) über 5 eine `acknowledge`-Meldung zurück an 4. 4 darf daraufhin – und erst jetzt! – eine `stop`-Meldung an 0 schicken (d). 4 hat in diesem Beispiel nach dem Schicken der Primary Messages keine eigentliche Arbeit mehr, sondern wartet nur noch auf das `acknowledge`, um dann selbst *idle* zu gehen und das an 0 zu melden.

Diese Darstellung ist stark verkürzt, DTD benutzt beispielsweise tatsächlich

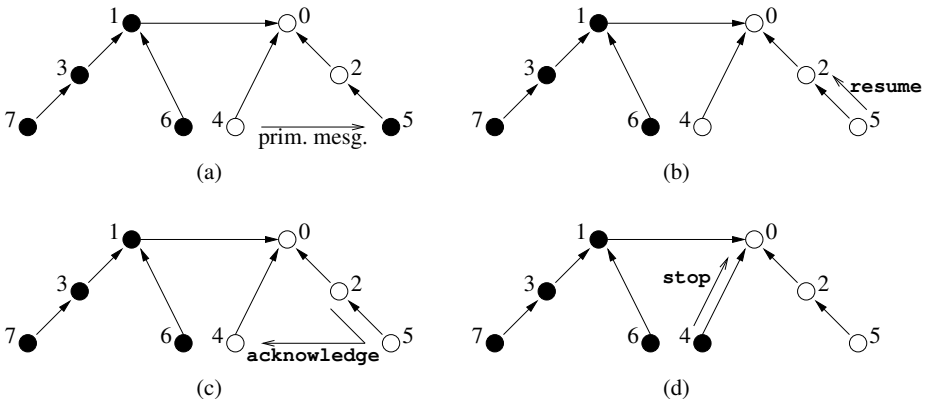


Abbildung 2.3: Beispiel des Terminierungsalgorithmus' aus [MD01]. Schwarze Knoten sind *idle*, weisse *busy*. Die Knoten sind in einem Baum angeordnet, 0 ist die Wurzel. Die ausgefüllten Pfeile beschreiben den Baum, einfache Pfeile sind Nachrichten.

mehr als nur zwei Knoten-Zustände. Mahapatra und Dutt zeigen in [MD01], dass DTD korrekt ist und sehr effizient.

2.4 Anwendungen

Verteilte Systeme sind sehr in Mode, deshalb gibt es auch zahlreiche Anwendungen. Ich zeige nur exemplarisch die wichtigsten Richtungen auf.

Weiter oben (Abschnitt 2.3.2, S. 14) haben wir schon Anwendungen des Grid-Computing gesehen, unter anderen Napster, Gnutella und SETI@home.

McClelland und Rumelhart nennen in [MR88] Kognitive Prozesse und Neuronale Netze. Besonders interessant ist diese Arbeit aber, weil es die wohl erste ist, die als Anwendung *Constraint-Satisfaction in Parallel Distributed Systems* vorschlägt. Die Autoren geben dort auch erste Methoden zur Lösung an. Siehe hierzu auch Abschnitt 3.4.11 [S. 34].

Dietsch nennt in [Die94] Echtzeit-Anwendungen und technisch-wissenschaftliches Höchstgeschwindigkeitsrechnen. Man kann übrigens auch DRS als einen Vertreter der letzten Gruppe ansehen.

Bärnreuther und Dietsch beschreiben in [BD94] Computer Integrated Manufacturing (CIM) als verteilten kooperativen Prozess zwischen Unternehmensplanung, Produktionsplanungssystem, Computer-Aided Design (CAD), Qualitätskontrolle und dem eigentlichen Produktionsprozess im Computer-Aided Manufacturing (CAM).

Weigelt und Mertens beschreiben in [WM94] eine Agenten-basierte Produktionssteuerung.

Hannebauer entwickelt in [Han01] ein System zur verteilten Patienten-Termin-Planung in großen medizinischen Einrichtungen.

Schließlich gibt es viele Simulations-Systeme, die parallel oder verteilt angelegt sind. Siehe hierzu Abschnitt 4.1.3 [S. 40].

2.5 Einordnung

Die Realisierung von DRS hat einige Gemeinsamkeiten mit Cluster-Computing und viele mit dem Grid-Ansatz. Die Konfiguration der Verteilung wird jeweils zu Beginn eines Simulationslaufs einmal berechnet: das Problem wird unter Anwendung von Netz-Partitionierungsverfahren zerlegt. Zur Kommunikation verwende ich Java RMI, einen abstrakten RPC-Ansatz. Zur theoretischen Fundierung benutze ich unter anderem Petri-Netze. Terminierung wird – soweit nötig – mit dem oben dargestellten Verfahren ermittelt. Auf all diese Dinge werde ich im Teil III [S. 97] noch genauer eingehen.

3 Constraint-Programmierung

Viele Planungs- und Optimierungs-Probleme lassen sich sehr elegant als Constraint-Problem darstellen. Außerdem gibt es viele generische Algorithmen, die bei der Suche nach einer Lösung des Constraint-Problems helfen. Insgesamt also kann man mit Constraint-Programmierung zahlreiche sehr schwierige Probleme relativ einfach lösen. Beschreibungs-Mächtigkeit und Propagations-Verfahren machen Constraint-Programmierung für unsere verteilte Simulation besonders interessant.

Constraint-Programmierung (CP) beschäftigt sich mit der Lösung von *Constraint-Satisfaction-Problems* (CSP) [Van89][FA97][Sch02b]. CSP sind Probleme, die aus einer Menge von Variablen bestehen, denen man Werte aus einer gemeinsamen Grundmenge (z.B. der Menge der ganzen Zahlen) zuordnen kann, sodass eine Menge von Constraints erfüllt sind.

Formal ist ein CSP also ein Tupel (D, V, C). D ist die *Grundmenge*, v ist die *Menge der Variablen* und C die *Menge der Constraints*. Jedes Constraint $c \in C$ bezieht sich auf eine Menge von Variablen und beschreibt die gültigen Wertekombinationen dieser Variablen. Also ist jedes $c \in C$ ein Tupel (I, J), wobei $I \subset V$ die Menge der Variablen ist, auf die sich das Constraint bezieht, und $J \subset \{j | j : I \to D\}$ die Menge aller bezüglich c gültigen Variablenbelegungen. Eine Belegung $f : V \to D$ weist jeder Variable einen Wert aus der Grundmenge zu. Eine *Lösung* eines CSP ist eine solche Belegung f, die alle Constraints erfüllt, also $\forall (I, J) \in C : f{\downarrow}I \in J$. Hier ist $f{\downarrow}I$ die übliche Einschränkung von f auf I, siehe auch Definition 7.2 [S. 52].

In der Praxis können solche CSP sehr anschaulich deklarativ geschrieben werden. Betrachten wir als Beispiel das bekannte *Send-More-Money*-Puzzle [Van89, 5.3.4]: Die Legende sagt, ein Student schickt seinen Eltern ein Telegramm mit dem schlichten Inhalt:

```
  send
+ more
======
 money
```

Wenn hier jeder Buchstabe durch eine andere Ziffer ersetzt wird und keine der dann dastehenden Zahlen mit 0 beginnt, gibt es genau eine Lösung für die Gleichung. Laut Legende haben die Eltern die Summe natürlich dem Studenten überwiesen.

3.1 Notation

Als CSP kann man das Send-More-Money-Problem quasi unmittelbar hinschreiben. Als SICStus-Prolog-Programm ([SIC], siehe auch Abschnitt 3.4.9, S. 32) beispielsweise sieht es dann so aus: siehe Abbildung 3.1 [S. 22].

Die Zeilen 2-5 deklarieren die Variablen des CSP: `L` wird als Liste von Variablen definiert, die Elemente der Liste werden anschließend als Constraint-Variablen deklariert mit der Menge der ganzen Zahlen als Grundmenge und initialer Domäne von 0 bis 9. Außerdem dürfen `S` und `M` nicht gleich 0 sein.

In den Zeilen 6-9 werden die Constraints definiert: dass alle in der Liste `L` enthaltenen Variablen unterschiedliche Werte bekommen müssen und als arithmetische Formel das eigentliche Puzzle. Für arithmetische Formeln über der

```
1   sendmory(Lab, L):-
2           L = [S,E,N,D,M,O,R,Y],
3           domain(L, 0, 9),
4           S #> 0,
5           M #> 0,
6           all_different(L),
7                           1000*S + 100*E + 10*N + D
8           +               1000*M + 100*O + 10*R + E
9           #= 10000*M + 1000*O + 100*N + 10*E + Y,
10          labeling(Lab, L).
```

Abbildung 3.1: Constraint-Programm zur Lösung des Send-More-Money-Problems.

Grundmenge der ganzen Zahlen gibt es in SICStus (genauso wie in vergleichbaren Systemen) alle wichtigen Operatoren, außerdem können viele Constraints über aussagenlogische Formeln verbunden werden: siehe Tabelle 3.1 [S. 22].

`A #= B`	`A` gleich `B`
`A #\= B`	`A` ungleich `B`
`A #< B`	`A` kleiner `B`
`A #=< B`	`A` kleiner gleich `B`
`A #> B`	`A` größer `B`
`A #>= B`	`A` größer gleich `B`
`#\ Q`	nicht `Q`
`P #/\ Q`	`P` und `Q`
`P #\/ Q`	`P` oder `Q`
`P #\ Q`	entweder `P` oder `Q`
`P #=> Q`	wenn `P` dann `Q`
`P #<= Q`	wenn `Q` dann `P`
`P #<=> Q`	`P` genau dann wenn `Q`

Tabelle 3.1: Arithmetische und Boolesche Operatoren

In Zeile 10 des Programms `sendmory` wird übrigens das Lösungsverfahren aufgerufen, in diesem einfachen Fall ein Standard-Suchverfahren. Dieses benötigt in SICStus einige zusätzliche Parameter, die hier durch die logische Variable `Lab` repräsentiert sind.

CSP beschreiben also *deklarativ* ein Problem: Man kann direkt das Problem selbst hinschreiben, anstelle des Problemlösungsalgorithmus. Die Problemlösung wird zum Teil automatisch vom Programmsystem übernommen, zum Teil anderweitig vom Programmierer definiert.

An grundlegender Literatur zum Thema Constraint-Programmierung sind

vor allem zu nennen: Van Hentenrycks frühes und viel beachtetes Werk [Van89], Kumars schöne Einführung [Kum92] sowie die Lehrbücher von Frühwirth und Abdennadher [FA97] und von Marriot und Stuckey [MS98].

3.2 Motivation

Deklarativität, also die mögliche Trennung von Problem-*Beschreibung* und Problem-*Lösung* ist das Haupt-Argument für Constraint-Programmierung. Zusammen mit allgemeinen und sehr effizienten Algorithmen können viele reale Probleme auch wirklich gelöst werden.

Gerade in der Künstlichen Intelligenz [Gör95] gibt es viele Probleme, die sich sehr elegant als CSP beschreiben lassen. Das kanonische Beispiel hier ist *Bilderkennung*: Waltz [Wal72] war einer der ersten, die hierfür Algorithmen entwarfen, die wir heute Constraint-Algorithmen nennen. In [Wal75] beschreibt er, wie Constraints benutzt werden können, um in Zeichnungen von einfachen Block-Welt-Szenarien Grenzlinien von Objekten zu erkennen. Andere Beispiele sind: Erfüllbarkeit aussagenlogischer Formeln, Planungsprobleme, Graph-Coloring, automatisches Beweisen. Weiter unten gehe ich genauer auf die Anwendungen ein: Abschnitt 3.5 [S. 35].

3.3 Probleme

Viele der CSP sind NP-vollständig. Van Hentenryck schreibt in [Van89, 1.1] dazu:

> Since boolean satisfiability is *NP-complete*, a general algorithm to solve the whole class problem will necessarily require exponential time in problem size in the worst case (unless $P = NP$). As a consequence, backtracking is an important technique for solving CSPs. Also, it is extremely unlikely that a particular algorithm outperforms all others on the whole class. [...] This means that solving a problem from this class can require the combination of several techinques and should be considered a creative act.

Diese Aussagen sind schon ein paar Jahre alt, gelten aber unverändert. Es ist heute noch nicht klar, ob $P = NP$ [GJ79]: P ist die Klasse der deterministisch polynomiellen Verfahren, NP die Klasse der nicht-deterministisch polynomiellen Verfahren. Auf klassischen Computersystemen (anders als z.B. bei Quantencomputern [Qua]) sind deterministsch polynomielle Verfahren immer auch polynomiell, nicht-deterministisch polynomielle Verfahren hingegen sind nicht-polynomiell. Ein polynomielles Verfahren benötigt für die Lösung eines Problems der Größe n im schlechtesten Fall t Zeiteinheiten und t ist ein Polynom von n, also z.B. $t = n$ oder $t = n^2$ oder $t = n^3$. Nicht-polynomielle Verfahren benötigen dafür deutlich mehr Zeit, nämlich in der Regel und mindestens exponentiell viel Zeit, also z.B. $t = 2^n$. In die Klasse NP fallen Erfüllbarkeit, Graph-Coloring und viele Planungsprobleme.

Es ist also noch immer unklar, ob $P = NP$ oder nicht. Es spricht vieles dagegen, der theoretische Beweis aber steht noch aus. Vor allem wegen der Schwierigkeit der Probleme ist es immer noch fast eine Kunst, für ein gegebenes Problem ein schnelles Lösungsverfahren zu finden.

3.4 Ansätze

Es gibt unterschiedliche Ansätze zur Lösung von CSP. Ich zeige zuerst kurz die neuesten Entwicklungen zu *Local Search* als Methode zur Lösung von CSP, und dann die wichtigste Methode: Propagation. Danach folgen unterschiedliche Propagations-Aspekte.

3.4.1 Local Search oder Local Repair

Diese Klasse von Algorithmen wird meist *Local Search* genannt. Weil ich aber später noch von ganz anderen Such-Verfahren berichten werde, möchte ich einen anderen geläufigen Begriff dafür verwenden: *Local Repair*.

Local-Repair-Methoden [AL97] gehen generell wie folgt vor: zunächst wird eine (oder mehrere) vollständige initiale Belegung – ein *Lösungskandidat* – erzeugt. Diese wird dann Schritt für Schritt verbessert, indem in Teilen (lokal) von diesem verschiedene andere Kandidaten erzeugt und bewertet werden, die dann den aktuellen Kandidaten ersetzen. Diese Schritte werden wiederholt bis eine genügend gute Lösung erzeugt ist. Konkrete Algorithmen sind unter anderem *Simulated Annealing* [Ste99], *Hill Climbing* [GW93], *Tabu Search* [Glo90] und *Genetische Algorithmen* [Gol89].

Local Repair ist eine sehr erfolgreiche Methode. Gleichwohl gibt es wenige Ansätze, die die Deklarativität von CSP mit Local Repair verbinden. Minton und andere schlagen in [MJPL92] den Algorithmus *Min-Conflict Heuristic Repair* vor, Morris entwirft in [Mor93] den Algorithmus *Breakout*. Fabiunke schlägt in [Fab99] einen eigenen parallelen Algorithmus vor. Nareyek gibt für seine *Constraint-based Agents* in [Nar01] Local-Repair-Methoden an. Schließlich gibt es jüngst von Codognet, der viel im Bereich Constraints und Propagation gearbeitet hat, einen solchen Algorithmus [CD01].

3.4.2 Propagation

Constraint-*Propagation* ist erheblich populärer, nicht wenige setzen gar *Constraint-Programmierung* mit Propagation gleich. Ich werde deshalb im folgenden CP meist nur noch im Sinne von Constraint-Propagation verstehen und beleuchten.

Van Hentenryck [Van89, 1.2.2] zu den Anfängen von Propagation:

> Consistency techniques are based on the idea of a priory pruning, that is, using the constraints to reduce the search space before the discovery of a failure. They originated from Waltz's filtering algorithm [Wal72] and Fikes's REF-ARF [Fik68].

Andere (z.B. Saraswat [Sar93]) führen den Ansatz außerdem zurück auf das System *Sketchpad* von Sutherland [Sut63]. Grundlegende Literatur geben neben Van Hentenryck und Saraswat auch Codognet und Diaz mit Implementierungstechniken [CD96] und Apt mit seinen theoretischen Grundlagen für Konvergenz von Propagationsmethoden [Apt99].

Die Grundidee von Constraint-Propagation ist, während der Suche nach einer Lösung des CSP die Domänen der Variablen einzuschränken, basierend auf vorhandenem Wissen über Constraints und andere Variablen. So kann man sich bei der Lösungssuche auf relativ wenige Lösungskandidaten beschränken und

Teil-Lösungen, die sich nicht zu einer korrekten Gesamtlösung erweitern lassen, frühzeitig erkennen.

Constraint-Propagation ist also in folgendem Sinne *monoton*: Im initialen CSP werden die Domänen der Variablen durch Propagation eingeschränkt. Während der Lösungssuche werden Variablen belegt und logisch daraus folgende weitere Einschränkungen an den Variablen-Domänen vorgenommen. Solange das Constraint-System nicht widersprüchlich ist (in der Regel indem die Domäne einer Variablen leer wird), werden also alle Domänen immer kleiner, bis schließlich alle aus genau einem Wert bestehen und eine vollständige Lösung des CSP darstellen. Dieser Prozess ist also bezüglich der Variablen-Domänen monoton. Nur wenn ein Widerspruch festgestellt wird, müssen Variablen-Domänen auf frühere Werte zurückgesetzt werden, wodurch sie sich vergrößern (die Mengen bekommen wieder mehr Elemente).

Betrachte als Beispiel folgendes Programm in obiger Syntax (Abschnitt 3.1, S. 21):

```
1           L = [A,B,C],
2           domain(L, 3, 9),
3           A + B #= C,
4           labeling(Lab, L).
```

Dieses Programm enthält drei Constraint-Variablen, `A, B` und `C`. Deren Domänen haben die Werte 3 bis 9. Nachdem aber `C` die Summe von `A` und `B` sein soll, kann C nicht kleiner als 6 sein. Genauso können `A` und `B` nicht größer als 6 sein: Angenommen `A` wäre 7 und `B` 3, dann müsste ja `C` gleich 10 sein. Allein durch Propagation kann man bei diesem Beispiel also folgende Variablen-Domänen ableiten: `A::3..6, B::3..6, C::6..9`. Wenn nun bei der Lösungssuche beispielsweise für `A` der Wert 6 genommen wird, kann man unmittelbar für `B` die 3 und für `C` die 9 ableiten. Man hätte also in einem Schritt eine Lösung für das CSP gefunden.

Van Hentenryck zeigt in [Van89, 5.3.4], dass das Send-More-Money-Beispiel mit obiger Formulierung allein durch Propagation auf folgende Domänen reduziert werden kann: `S=9, M=1, O=0, E::4..7, N::5..8, R::2..8, D::2..8, Y::2..8`. Hier benutze ich übrigens die in einigen CLP-Systemen gebräuchliche Domänen-Intervall-Schreibweise `Variable :: Minimum .. Maximum`.

Wenn man initial von einem potentiellen Suchraum für das gegebene CSP von 10^8 ausgeht (alle 8 Variablen haben einen Wertebereich mit zehn Elementen), wäre dieser Suchraum allein durch Propagation auf $4 * 4 * 7 * 7 * 7 = 5488$ reduziert, also auf 0,005% der ursprünglichen Größe. Dieselbe Propagation sorgt dann bei der Suche dafür, dass maximal ein falscher Kandidat getestet werden muss, also maximal zwei Suchschritte ausreichen, um das Problem zu lösen. Hätte man einfach jede Möglichkeit des ursprünglichen Problems generiert und anschließend getestet, hätte man im schlechtesten Fall 10^8 Suchschritte machen müssen.

3.4.3 Constraint-Solver

Technisch wird Propagation immer in einem *Constraint-Solver* realisiert [CD96]. Dem Solver wird das CSP im verwendeten Programmiersystem bekannt gemacht und er kümmert sich während der Lösungssuche um die Propagation zwischen

Variablen und Constraints. Der Constraint-Solver sorgt dafür, dass durch Änderungen an Variablen die richtigen Constraint-Methoden aufgerufen werden, die ihrerseits wieder Variablen ändern und dadurch andere Methoden triggern. Dieser Prozess berechnet jeweils einen impliziten Fixpunkt bezüglich des CSP [CD96].

Es gibt viele wissenschaftliche und kommerzielle Constraint-Solver für die unterschiedlichsten Paradigmen. Siehe hierzu Abschnitt 3.4.9 [S. 32] und Abschnitt 3.4.10 [S. 33]. Einen interessanten Überblick geben Fernandez und Hill in [FH00]: Sie vergleichen dort sehr genau Funktionalität, Performanz und Stabilität der wichtigsten Implementierungen.

3.4.4 Constraint-Systeme

Propagation findet immer in einem sog. *Constraint-System* statt [Rin03], das im CSP die Grundmenge vorgibt. Propagations-Methoden sind immer spezifisch für das gegebene System. Die bekanntesten Systeme sind: Bäume [Col84], endliche Mengen ganzer Zahlen [Van89], reelle Zahlen [JMSY92], aussagenlogische Formeln [Mas93], rationale Zahlen [Hol95], allgemeine endliche Mengen [Ger97]. Das populärste System ist das der endlichen Mengen ganzer Zahlen, auch *Finite Domain* oder *FD* genannt. In diesem System sind die meisten Anwendungen formuliert, die meisten Constraints entwickelt. Andere Systeme wie allgemeine endliche Mengen oder logische Formeln lassen sich leicht in das FD-System übertragen, weshalb in FD-Solvern meist auch solche Constraints realisiert sind.

Wie wir wissen, ist das Finden einer passenden Lösungsmethode für ein gegebenes CSP ein kreativer Prozess. Dechter schreibt dazu in [Dec92]:

> Deciding the level of consistency that should be enforced on the [constraint] network is not a clear-cut choice. Generally speaking, backtracking will benefit from representations that are as explicit as possible, having higher consistency level. However, because the complexity of enforcing i-consistency is exponential in i, there is a trade-off between the effort spent on preprocessing and that spent on search.

Die Kunst besteht also darin, zum gegebenen Problem die optimalen Such- und Propagations-Strategien zu finden. Die beiden folgenden Abschnitte befassen sich mit den jeweils aktuell verfügbaren.

3.4.5 Constraints und Propagations-Verfahren

Aus obigen Gründen beschränke ich mich im folgenden auf das Constraint-System der endlichen Mengen ganzer Zahlen, FD. Hier lassen sich die existierenden Constraints in drei Klassen einteilen: unäre, binäre und n-äre Constraints.

Unäre Constraints

Unäre Constraints beziehen sich auf genau eine Variable, zum Beispiel `A #\= 3`, `A #>= 3` (Syntax: Abschnitt 3.1, S. 21). Diese Constraints sind alle einfach zu realisieren: Für jede Variable muss die Domäne einmal entsprechend angepasst werden. Der Solver muss also zum Beispiel bei der Variable `A` den Wert 3 streichen oder alle Werte kleiner als 3.

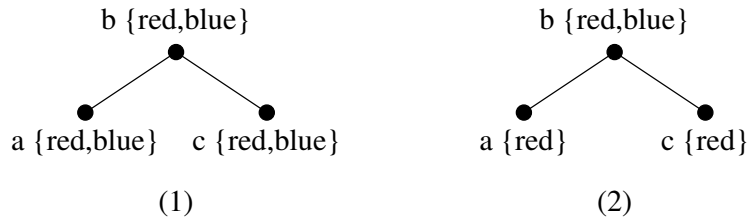


Abbildung 3.2: Beispiel zur k-Konsistenz aus [Fre82].

Binäre Constraints

Binäre Constraints beschreiben Relationen zwischen zwei Variablen. Beispiele (Syntax: Abschnitt 3.1, S. 21): `A #= B`, `A #\= B`, `A #> B`, `A #= B + 3`.

Binäre Constraints sind in der Theorie beliebt und weit erforscht. Freuder beispielsweise definiert in [Fre82] für binäre CSP den berühmten Begriff der k-Konsistenz wie folgt:

> Chose any set of $k-1$ variables along with values for each that satisfy all the constraints among them. Now choose any kth variable. There exists a value for the kth variable such that the k values taken together satisfy all constraints among the k variables.

Betrachte folgendes Beispiel: Abbildung 3.2 (1) [S. 27]. Der Graph bestehend aus den Knoten a, b und c soll gefärbt werden, und zwar so, dass keine verbundenen Knoten dieselbe Farbe erhalten. Jeder der Knoten kann rot oder blau sein. Dieses Problem kann direkt als CSP beschrieben werden: für jeden Knoten eine Variable, jeweils mit den Domänen $\{rot, blau\}$, und zwischen a, b und b, c jeweils ein Ungleichheits-Constraint. Der Graph gibt übrigens dann ein schönes Bild für das Netz des CSP.

Dieses CSP ist 1-konsistent, 2-konsistent, aber nicht 3-konsistent. Wenn man nämlich beispielsweise für Knoten a die Farbe rot wählt, und für c blau, sind alle Constraints erfüllt. Dann aber gibt es für den dritten Knoten b keine Belegung mehr, die alle Constraints erfüllt. Freuder erweitert k-Konsistenz auf *starke k-Konsistenz*: ein CSP ist *stark k-konsistent*, wenn es für alle $j \leq k$ j-konsistent ist. Es gibt auch CSP, die nicht stark k-konsistent sind, zum Beispiel das in Abbildung 3.2 (2). Dieses ist 1- und 3- konsistent, aber nicht 2-konsistent. Wenn man für b rot wählt, findet man für a keine Möglichkeit mehr, sodass alle Constraints erfüllt sind.

1-Konsistenz wird auch *Node-Consistency* genannt, 2-Konsistenz *Arc-Consistency*, und *Path-Consistency* ist dasselbe wie 3-Konsistenz ([Apt03, 5.5]). Wie wir oben gesehen haben, ist Node-Consistency leicht herzustellen. Arc-Consistency ist schon schwieriger. Dafür wurden eine Vielzahl von Lösungsalgorithmen entworfen, u.a. AC-1, AC-3 [Mac77], AC-6 [Bes94], AC-7 [BFR95]. Noch schwieriger ist Path-Consistency: z.B. PC-5 [Sin95] und PC-8 [CJ96]. Gerade letztere müssen auf expliziten Constraint-Repräsentationen arbeiten, also einer Aufzählung aller möglichen Wertekombinationen.

Starke k-Konsistenz – oft auch *vollständige Propagation* genannt – ist unter anderem deshalb so besonders interessant, weil ein CSP, das aus k Variablen besteht und stark k-konsistent ist, ohne Suche gelöst werden kann [Fre82]. Al-

lerdings ist es extrem aufwändig, k-Konsistenz herzustellen: die Zeit- und Speicher-Komplexität zur Herstellung von k-Konsistenz ist polynomiell *in* k [DB97]. Bezogen auf obige Betrachtungen (Abschnitt 3.3, S. 23) bedeutet das, es werden im schlechtesten Fall $x = n^k$ Schritte benötigt, für n-Konsistenz also wieder $x = n^n$.

Praktikabler sind hierfür die Propagations-Algorithmen *Forward-Checking* und *Looking-Ahead* [Van89]. Forward-Checking realisiert eine schwächere Form von Arc-Consistency, beschränkt auf die zuletzt instanziierte Variable [GSW00]. *Looking-Ahead* ist entsprechend eine schwächere Form von Path-Consistency.

Alle diese Ansätze sind theoretisch sehr elegant. Sie haben in der Anwendung aber zunächst das Problem, dass die meisten Algorithmen eine explizite Constraint-Repräsentation benötigen, was exponentiellen Speicherplatz-Bedarf bedeutet, und eben nur für binäre Constraints gelten. Es ist zwar theoretisch möglich, für jedes CSP ein äquivalentes anzugeben, das nur aus binären Constraints besteht [RPD90]. Allerdings müssen dann die Variablen und deren Funktion geändert werden: z.B. muss man für das Constraint `A #< B + C` für die Variablen `B` und `C` eine neue Variable einführen, die alle gültigen Wertekombinationen für `B` und `C` repräsentiert. Praktisch also lassen sich nicht alle möglichen Relationen für mehr als zwei Variablen durch binäre Constraints darstellen.

Arithmetische und Boolesche Constraints

Wie wir weiter oben schon gesehen haben, unterstützen moderne Constraint-Solver sowohl arithmetische Constraints wie `A #= B + C` oder `A #= B * C` als auch aussagenlogische Constraints [BC93]. Damit können so genannte *reifizierbare* – zum Beispiel arithmetische – Constraints durch boolesche Operatoren verknüpft werden: `A #= 3 #<=> B #< 5`, `A #= 1 #\/ B #< C`. Diese Constraints werden durch Intervall-Arithemtik realisiert.

Globale Constraints

Beldiceanu und Contejean führten in [BC94] das Konzept der *globalen Constraints* ein. Globale Constraints bilden Relationen zwischen vielen Variablen auf einmal ab. Dadurch können spezialisierte Algorithmen die Propagation für die beteiligten Variablen besorgen. So konnte Régin in [Rég94] zeigen, dass es erheblich effizienter sein kann, das `alldifferent`-Constraint über n Variablen nicht in $n(n-1)/2$ Ungleichungen zu zerlegen, sondern einen speziellen Graphen-Algorithmus zu verwenden. Effiziente und mächtige globale Constraints sind sehr wichtig für einen guten Constraint-Solver. Tabelle 3.2 [S. 29] zeigt eine Auswahl der wichtigsten globalen Constraints im System FD.

3.4.6 Suche

Zusätzlich zur optimalen Propagations-Strategie, muss man für das gegebene CSP eine Suchmethode finden. Weil wir *Suche* im Zusammenhang mit *Propagation* verwenden, sind Backtracking-Verfahren nötig: Man wählt eine Variable aus, für diese einen Wert, ordnet der Variable den Wert zu und stößt die Propagation an, um zu überprüfen, ob der Wert zu der Variablen passt. Dann wählt man die nächste Variable aus, belegt sie und so weiter. Durch Propagation kann es sein, dass im Suchprozess die Domäne einer Variablen leer wird. Das ist dann die Folge eines Widerspruchs zwischen den bereits erfolgten Belegungen und den Constraints. In diesem Fall muss im Suchbaum an eine Stelle zurückgesprungen

`alldifferent([V_i])`	Alle V_i müssen verschieden sein. [Van89]
`cardinality(Min, Max,[C_i])`	Die C_i sind *einfache* Constraints. Mindestens `Min` und höchstens `Max` davon müssen erfüllt sein. [VD91]
`cumulative(Limit, [S_i],[D_i],[R_i])`	Jedes (S_i, D_i, R_i) beschreibt einen Task mit Startzeitpunkt S_i, Dauer D_i und Ressourcenverbrauch R_i. Zu keinem Zeitpunkt ist die Summe aller Ressourcenverbräuche größer als `Limit`. [AB93][BC94][CL96][BC02]
`among(Min,Max, [X_i],[C_i],[V_j])`	Mindestens `Min` und höchstens `Max` Gleichungen $X_i + C_i$ nehmen einen Wert aus V_j an. [BC94]
`element(Ix,[V_i],V)`	$V = V_{Ix}$. [BC94]
`maximum(M,[V_i])` `minimum(M,[V_i])`	`M` ist das Maximum bzw. Minimum aller V_i. [BC94]
`cycle(N,[V_i])`	Die V_i beschreiben als Adjazenz-Liste `N` nichtverbundene Graphen, die jeweils einen Hamiltonschen Zyklus bilden. [BC94]
`diffn(R_i)`	Die R_i beschreiben n-dimensionale Rechtecke, die sich nicht überlappen. [AB93][BC94]
`disjoint(R_i)`	Die R_i beschreiben 1- oder 2-dimensionale Rechtecke, die sich nicht überlappen. [SIC] [BC01][Wol03a]
`sum(X,[V_i])`	`X` ist die Summe aller V_i. [RR00b]
`sequence(Min,Max, [V_i],[P_i])`	Die V_i erfüllen mindestens `Min` und höchstens `Max` Belegungsmuster aus P_i. [CHI01]
`stretch(...)`	Bezogen auf eine Liste von Variablen, beschränkt `stretch` die aufeinander folgend gleichen Werte auf gegebene Muster. [Pes01]
`flow(...)`	Ein Constraint für *Network Flow* Probleme. [BPA01]
`lex_chain([L_i])`	Jede Liste L_i beschreibt eine Zeichenkette und alle L_i sind lexikalisch geordnet. [SIC][FHK+02]

Tabelle 3.2: Globale FD-Constraints

werden und die Variablenbelegungen und andere Such-Informationen dorthin zurückgesetzt werden. Dann wird von dort ein anderer Weg eingeschlagen.

Die älteste und einfachste Methode ist chronologisches Backtracking: Gegeben eine Reihe von Variablen $v_1, ..., v_n$. Die Variablen werden in dieser Reihenfolge belegt und beim Zurückspringen wird immer die chronologisch letzte Belegung rückgängig gemacht, man springt also im Such-Baum immer genau eine Position zurück.

Abbildung 3.3: Deutsche Bundesländer [Bun].

Dass dieses Verfahren extrem ungünstig sein kann, zeigt sehr anschaulich Ginsberg [Gin93] anhand eines *Map-Coloring* Problems für die Vereinigten Staaten. Ich möchte es kurz auf die deutschen Bundesländer übertragen, siehe Abbildung 3.3 [S. 30]. Das Problem ist, die Länder auf der Landkarte so einzufärben, dass keine zwei Nachbarn dieselbe Farbe haben. Bekanntlich genügen 4 Farben [Wei]. Das Problem wird wie oben in Abschnitt 3.4.5 [S. 26] modelliert. Angenommen nun, wir belegen die Variablen in folgender Reihenfolge: zunächst Sachsen, dann Brandenburg und weiter an der Grenze entlang gegen den Uhrzeigersinn bis Bayern, dann Hessen, Thüringen und Sachsen-Anhalt. Die Farben seien *rot, blau, gelb* und *grün*. Weiter angenommen, wir hätten bis auf Sachsen-Anhalt schon alle Länder gefärbt und aus irgendwelchen Gründen sei Sachsen rot, Brandenburg blau, Mecklenburg-Vorpommern gelb und Niedersachsen grün. Offenbar gibt es dann keine Farbe mehr für Sachsen-Anhalt. Per chronologischem Backtracking würden wir nun aber zuerst für Thüringen alle anderen Möglichkeiten probieren, dann zurück zu Hessen, dann für Bayern und natürlich jeweils wieder alle Kombinationen von Möglichkeiten für die nachfolgenden Länder. Das Problem aber liegt bei Niedersachsen oder vielleicht sogar noch weiter vorne. Chronologisches Backtracking kann also unnötig aufwändig sein.

Deshalb wurden geschicktere allgemeine Backtracking-Verfahren entwickelt, die nicht immer schlicht eine Position zurückspringen. Wie man oben sehen kann, ist aber nicht nur die Rücksprung-Strategie wichtig, sondern während der Suche auch die Wahl der nächsten zu belegenden Variable und die Wahl des Werts dafür. Tabelle 3.3 [S. 31] zeigt die wichtigsten derartigen Verfahren. Dabei können die allgemeinen Backtracking-Verfahren beliebig mit denen für Variablen- und Werteauswahl kombiniert werden. Für einige Probleme existieren aber auch spezielle kombinierte Algorithmen.

Für Überblick und Vergleiche siehe vor allem Kondrak und van Beek [KvB95] und Schimpf et al. [SSW99].

Backtracking	Chronological Backtracking [GB65][BR75] Backmarking [Gas77] Backjumping [Gas78] Forward-Checking [HE80] Graph-based Backjumping [Dec90] Conflict-Directed Backjumping [Pro93][HMSW03] Dynamic Backtracking [Gin93] Weak-Commitment [Yok94] Foward-Checking with Backmarking and Conflict-Directed Backjumping FC-BM-CBJ [Pro95] Bounded Backtrack Search [SSW99] Timeout Search [SSW99]
Variablen-Auswahl	nächste (in der Liste) [SIC] Domäne mit kleinstem Wert [SIC] Domäne mit größtem Wert [SIC] First-Fail: kleinste Domäne [Van89] Most-Constrained [SIC] Max-Regret [CHI01]
Werte-Auswahl	kleinster Wert [SIC] größter Wert [SIC] zufälliger Wert Domain-Splitting/-Reduction [Van89, 4.4.2][Gol95]
Kombinierte Spezial-Verfahren	Reduce-to-the-Max [Sch00][Wol03b] Perfect Square Placement [BBS] Dynamic Backtracking for Dynamic Constraint Satisfaction Problems [VS94]

Tabelle 3.3: Suchstrategien

3.4.7 Dynamische und Soft-Constraints

Wir haben CSP so definiert, dass eine Lösung *alle* Constraints erfüllen *muss*. In der Praxis aber gibt es oft Probleme, für die einige Constraints erfüllt werden müssen und andere nach Möglichkeit erfüllt werden sollen, etwa weil die Constraints insgesamt widersprüchlich sind. Solche Constraints heißen *Soft-Constraints*. Das Hauptproblem der Behandlung von Soft-Constraints ist, dass man

diese nicht mehr unmittelbar zur Propagation nutzen kann, weil für ein Soft-Constraint ja nicht klar ist, ob das Constraint am Ende gilt oder nicht. Populäre Ansätze sind:

- Weak Constraints: Blake [Bla83]
- Constraint-Hierarchien: Freeman-Benson et al. [FBMB90], Borning et al. [BFBW92], Wolf [Wol98]
- Semi-Ring Constraint-System: Bistarelli et al. [BMR97]
- Distributed Partial Constraint Satisfaction: Hirayama und Yokoo [HY97]
- Adaptive Constraintverarbeitung: Wolf [Wol99]

Auch möchte man oft ein CSP, nachdem es definiert wurde, zum Beispiel während der Lösungssuche verändern. Inkrementelles Hinzufügen neuer Constraints ist in der Regel kein Problem, die meisten Implementierungen unterstützen das von Hause aus. Schwierig ist es allerdings, Constraints aus einem CSP zu entfernen. Constraints schränken ja die möglichen Werte-Kombinationen der beteiligten Variablen ein. Wenn man ein Constraint aus einem CSP weg nimmt, bedeutet das, das dann mehr Kombinationen möglich sind. Dann aber gelten die bereits durch Propagation eingeschränkten Domänen der Variablen nicht mehr. Constraint-Entfernung ist bzgl. Propagation nicht-monoton. Ansätze hierfür sind:

- Incremental Constraint Solver: Freeman-Benson et al. [FBMB90]
- DeltaBlue/SkyBlue: Sanella et al. [San94][SMFBB93]
- Cassowary: Baldros und Borning [BB98] und Hosobe [Hos00]
- Asynchronous Constraint Satisfaction: Ringwelski [Rin03]

3.4.8 Constraint-Handling Rules

Propagations-Methoden werden meist imperativ programmiert. Frühwirths *Constraint Handling Rules* (CHR, [Frü95]) stellen eine elegante Möglichkeit dar, solche Methoden Regel-basiert zu definieren. Frühwirth selbst gibt in [Frü95] und [Frü98] eine Einführung in diese Methode. Abdennadher untersucht in [AFM99] Konfluenz und Semantik spezieller CHR-Typen. Wolf nutzt CHR zur Formulierung seiner dynamischen Adaptiven Constraintverarbeitung [Wol99]. Ringwelski und Schlenker untersuchen in [RS00a] und [RS00b] typisierte CHR und geben Methoden an, diese Typen automatisch abzuleiten. Abdennadher und Rigotti geben in [AR01] Methoden an zur automatischen Generierung von CHR aus extensionalen Repräsentationen (Beispielen).

3.4.9 Constraint Logic Programming (CLP)

Constraint-Logische Programmierung (CLP) ist eine Erweiterung der Logik-Programmierung (LP) um Constraint-Propagation. Logik-Programmierung [GH89] ist ein eigenes Programmier-Paradigma: Programme werden durch Regeln formuliert, die Generierung eines Modells für das Logik-Programm führt über die Ableitung der Regeln zur Ausführung des Programms.

Die wichtigste Realisierung der Logik-Programmierung ist Prolog [Ges93]. Sterling und Shapiro schreiben zu den Anfängen von Prolog [SS94, 6.3]:

> [...] The formulation of the logic programming philosophy and computation model by Robert Kowalski [Kow74] and the design and implementation of the first logic programming language Prolog by Alain Colmerauer and his colleagues [Col70].

Heutige effiziente Implementierungen basieren alle auf der berühmten *Warren Abstract Machine* [War77]. Einen schönen Überblick über die Geschichte von Prolog geben Colmerauer und Roussel selbst [CR93]. Einführungen in Prolog bieten Sterling und Shapiro [SS94], Bartak [Bar] und Geskes Lehrbuch [Ges93].

In CLP werden gegenüber klassischer LP die logischen Variablen um Attribute erweitert und die Unifikation um Propagation. Constraint-Programmierung passt vor allem deshalb sehr gut zur LP, weil auch in der LP Programme generell deklarativ durch Regeln definiert werden und weil das Ausführungsmodell impliziert, dass per Backtracking nach einer Lösung gesucht wird, die alle Regeln erfüllt. Van Hentenryck hat in [Van89] die erste umfangreiche und heute noch gültige Darstellung von CLP einschließlich Theorie der CLP, Implementierung und Laufzeitergebnissen gegeben.

3.4.10 Object-Orientierte CP

Constraint-Programmierung wird auch immer mehr Objekt-orientiert gemacht:

- Kaleidoscope: Eine eigene objekt-orientierte Sprache von Freeman-Benson und Borning [FBB92].
- CLAIRE: ebenfalls eine eigene Sprache von Caseau et al. [CJL99]
- Mozart/OZ: [Sch02b][VH99]. Eine Multi-Paradigmen Programmiersprache (logisch, funktional, Objekt-orientiert) mit Constraint-Solver.
- Objekt-orientierte CLP: [CHI01][SR02a].
- ILOG Solver: [Pug94][ILO]. Das ist die bekannteste OO-Implementierung und überhaupt die erfolgreichste Constraint-Bibliothek. ILOG-Solver ist eine sehr umfangreiche C++-Bibliothek, mit der man in Anwendungen CSP formulieren kann, die Such-Methoden und Propagation zur Verfügung stellt.
- JSolver: Chun [Chu99]. Ein Java-basierter FD-Solver. Dieser Solver gehört mittlerweile ebenfalls ILOG und ist wohl ILOGs künftige Java-Solver-Implementierung.
- Java Constraint Library JCL: [TWF97][JCL]. Explizite Constraint-Repräsentation mit AC- und FC-Algorithmen.
- Koalog Constraint Solver: [Koa]. Ein FD-Constraint Solver, komplett in Java realisiert. Mit vielen Standard-Constraints, wahrscheinlich die zur Zeit umfangreichste einschlägige Java-Bibliothek.
- CHR und Java: Wolf [Wol01a] und JCHR von Schmauss [Sch99].

- Java Constraint Kit: [JKC]. Diese Bibliothek besteht aus JCHR, einem generischen Such-Werkzeug für JCHR und einer Visualisierung für CHR-Ableitungen.

- POOC: eine Java-Schnittstelle zu verschiedenen Constraint-Systemen von Schlenker und Ringwelski [SR02b].

- FIRSTCS: Hoche, Müller, Schlenker und Wolf [HMSW03]. Unsere neue hoch-effiziente Java-basierte FD-Constraint-Bibliothek mit einigen Standard-Algorithmen, vielen Neuerungen und innovativen Suchmethoden.

3.4.11 Verteilte CP

Für die vorliegende Arbeit von besonderer Bedeutung sind Ansätze zur verteilten Verarbeitung von Constraint-Problemen.

Formal kann man ein CSP (D, V, C) zu einem verteilten Constraint-Problem (DCSP) erweitern: (D, V, C, A, R). A ist eine Menge von Rechenknoten, und R ordnet jede Variable des Problems genau einem Knoten zu: $R : V \rightarrow A$. Gelöst wird das DCSP dann verteilt in allen Knoten. Eine Lösung des DCSP muss wieder alle Constraints erfüllen.

Den wohl ersten DCSP-Ansatz haben McClelland und Rumelhart in [MR88] formuliert, basierend auf Hinton [Hin77]. Yokoo und andere geben in [YDIK92] eine Formalisierung von DCSP an. Eaton et al. klassifizieren in [EFW98] Constraint-basierte Agenten aufgrund des Typs der jeweiligen Anwendung und geben einen umfassenden Überblick über jeweils aktuelle Arbeiten. Yokoo und Hirayama bieten in [YH00] die aktuellste Übersicht.

Besonders hervorzuheben sind auch noch folgende Arbeiten: Mammen und Lesser entwerfen in [ML98] einen generischen Generator für DCSP. Modi et al. lösen mit ihrem Algorithmus *Locally-Dynamic-AWC* dynamische DCSP. Hirayama und Yokoo behandeln in [HY97] die Lösung von *Partiellen* DCSP, also solchen mit Soft-Constraints, mit den Algorithmen *Synchronous Branch and Bound* und *Iterative Distributed Breakout*. Hannebauer beschreibt in [Han01] *Autnonome Dynamische Rekonfiguration* von DCSP. Yokoo und andere beschäftigen sich in [YSH02] mit *Secure Distributed Constraint Satisfaction* und entwerfen eine Verteilte Suche mit Umbenennung von Variablen und Verschlüsselung von Werten.

Generell lassen sich die meisten DCSP-Ansätze in drei Kategorien einteilen: *Distributed Search*, *Distributed Propagation* und *Distributed Application Propagation* (DAP). Verteilte Such-Algorithmen sind allgemeine Methoden zum Lösen von DCSP, wobei die beteiligten Knoten jeweils lokal eine Teillösung generieren und gemeinsam kooperativ diese Lösungen konsistent machen. Dabei werden direkt Variablen-Belegungen ausgetauscht. Bei verteilter Propagation werden ganze Constraints zwischen den Knoten kommuniziert, die jeweils lokal für Propagation genutzt werden. DAP-Anwendungen kommunizieren gar nicht auf der Ebene des Constraint-Problems, sondern noch abstrakter auf der Anwendungs--Ebene. Die Knoten verwenden lokal Constraint-Programmierung.

Tabelle 3.4 [S. 35] gibt einen Überblick über die jeweils wichtigsten Vertreter. Die vorliegende Arbeit kommuniziert auf der Anwendungs-Ebene und ist deshalb eindeutig der DAP-Gruppe zuzuordnen.

Distributed Search (Value-Exchange)	Distributed Constrained Heuristic Search [SRSF91] Async. Backtracking *ABT* [YDIK92] Heuristic Search for DCSP [LHB93] Distributed Breakout [YH96] Distributed Search [SGM96] Async. Weak-Commitment Search *AWC* [YDIK98] AWC mit Multi-Variablen-Agenten [YH98] AWC mit Nogood-Learning[RR96] [HY00]
Distributed Propagation (Constraint-Exchange)	Parallel AC *PTAC*, Distributed AC *PJAC* [ZM93] Distributed AC *DisAC-4* [ND95] Distributed Propagation [SGM96] Generalisierung von AWC [SR00a] Asynchronous Constraint Satisfaction [Rin03][RS02] Asynchronous Forward-Checking [MZ03]
Distributed Application Propagation	Constraint-basierte Agenten [EFW98] Coordination of Scheduling and Allocation Agents [SKÅD98] Verteilte Patienten-Terminplanung [Han01] Verteilte Eisenbahn-Simulation DRS [Sch03][Sch02a]

Tabelle 3.4: DCSP Algorithmen

3.5 Anwendungen

- Bilderkennung [Wal72][Wal75][Gör95].
- Graph-/Map-Coloring [Van89]. Siehe Abschnitt 3.4.5, S. 26.
- Puzzles [Van89] sind beliebt, um CP-Methoden zu testen. Mehrfach wurde darauf verwiesen, dass ein Algorithmus, der nicht für kleine Spiele taugt, für große Probleme erst recht nicht geeignet ist. Allerdings hat die Beschäftigung mit den Puzzles ein Eigenleben entwickelt: es existieren Algorithmen, die zwar für Puzzles hervorragend geeignet sind, sonst aber kaum.
  - Send-More-Money: Siehe Abschnitt 3.1 [S. 21].
  - N-Queens: Auf einem Schachbrett mit der Länge n müssen n Damen so aufgestellt werden, dass sich keine zwei gegenseitig bedrohen. Das ist ein beliebtes CP-Problem, allerdings mittlerweile etwas fragwürdig, weil es nicht NP-vollständig ist: Für beliebiges n lässt sich *eine* Lösung in linearer Zeit finden [SG91]. Propagation und Suche ist immer schlechter.
  - Crossword-Puzzles: Erzeugen und Lösen von Kreuzworträtseln.
- Perfect Square Placement [BBS]. Hier müssen Quadrate genau in ein großes Quadrat eingepasst werden.
- Cutting-Stock [Van89]. Das kanonische Beispiel hierfür ist [Chv83]: Ein Papierhersteller produziert Walzen in der Breite 100 Zoll, und hat Auf-

träge für unterschiedliche Breiten in unterschiedlicher Anzahl. Wie soll er die Walzen zerteilen, sodass minimaler Abfall entsteht? Dieses Problem ist dem Perfect-Square-Placement sehr ähnlich.

- Graphische Benutzer-Oberflächen [FBMB90][San94]. Die Position graphischer Elemente moderner Benutzer-Oberflächen kann mit Constraints beschrieben werden (z.B. *a auf gleicher Höhe neben b*, *a und b sollen den vertikal verfügbaren Platz gemeinsam ausfüllen*), während sich die Platzierung jeweils optimal mit CP berechnen lässt.
- Temporale Logik [Frü94].
- Ampel-Steuerung [How98].
- Personal-Einsatzplanung [AS97][AS99a][AS99b][SGO02]. Hier muss Personal, oft im Schicht-Betrieb, für unterschiedliche Arbeitsplätze eingeplant werden, sodass Arbeits-Gesetze eingehalten werden, die Arbeitsplätze optimal belegt sind und das Personal angenehme Arbeitszeiten hat.
- Stundenplanung für Schulen und Universitäten [GM99].
- Zeitliche Planung von Sport-Veranstaltungen [Rég98].
- *Planning and scheduling of goods transports (incl. staff) in a railway company* [SKÅD98]. Agenten-basiert verteilter Ansatz.
- Konfiguration technischer Anlagen [Joh02][GGJ97].
- Job-Shop Scheduling [GJ96]. Hier müssen Jobs, die in Tasks aufgeteilt sind, auf gegebene Maschinen verteilt werden, sodass alle Jobs möglichst früh erledigt sind.
- Projekt-Planung [BL97][BDM+99]. Jedes (z.B. Software-) Projekt besteht aus vielen Einzelaufgaben, die miteinander zusammenhängen und in eine optimale zeitliche Reihenfolge gebracht werden müssen. Projekt-Planung hat Ähnlichkeiten mit Job-Shop-Scheduling und meinem DRS-Ansatz.
- Gärtnerei-Planung [Sch00]. Die Bepflanzungsflächen einer Gärtnerei müssen für die verschiedenen Pflanzungen optimal verteilt und laufend dem veränderten Platzbedarf angepasst werden.
- Object Constraint Language OCL [BRJ99][RJB99]: formale Beschreibung von Objekten eines UML-Software-Modells anhand von Constraints. Mit OCL können auch relationale Datenbanken beschrieben werden [DH99]. OCL hat übrigens nichts mit Propagation zu tun, sondern ist eine Anwendung des deklarativen Formalismus.
- Design und Analyse von Netzwerken [LPRS02][BPA01].
- Verfolgung beweglicher Objekte mit Sensoren [FBKG02].
- Eisenbahn-Simulation [dO01][dOS01]. Siehe Kapitel 5 [S. 42].
- Verteilte Eisenbahn-Simulation [Sch03][Sch02a].

3.6 Einordnung

Das System DRS ist ein verteiltes Verfahren, das in den einzelnen Rechen-Knoten Constraint-Propagation benutzt, um das lokale Simulationsproblem zu lösen. Es ist insofern ein DAP-Ansatz. Wir benutzen sowohl Objekt-orientierte (Abschnitt 18.5.8, S. 112) als auch Logik-basierte (Kapitel 22, S. 146) Constraint-Verfahren. Bei der Lösungssuche kommen verschiedene spezielle Backtracking-Verfahren zum Einsatz. Zur Propagation verwenden wir arithmetische und zum Teil spezialisierte globale Constraints.

4 Simulation

Um reale Systeme genau zu verstehen, setzt man aus verschiedenen Gründen Simulationsverfahren ein. In diesem Kapitel werden die wichtigsten – insbesondere parallele und verteilte – Ansätze beschrieben und einige Anwendungen aufgezeigt.

Wenn man ein natürliches dynamisches System genau verstehen will, hat man dafür zwei grundsätzliche Möglichkeiten: Analyse und Simulation. Bei der Analyse zerlegt man das System in kleine Teile und formalisiert diese. Umgekehrt bedeutet Simulation den Aufbau eines dem natürlichen System ähnlichen Systems, und zwar in den Aspekten, an denen man interessiert ist, und die Beobachtung des Verhaltens des Simulationsmodells.

Analyse scheint also der direktere Weg zu sein. Allerdings sind viele Systeme der realen Welt derart komplex, dass man sie nicht so weit analysieren kann, bis man die interessante Frage vollständig analytisch beantwortet hat. So ist es für die Wettervorhersage zwar theoretisch denkbar, ein vollständiges analytisches Modell zu erstellen, das alle Faktoren, die das Wetter beeinflussen, berücksichtigt. Doch dieses Modell ist kaum praktisch realisierbar. Und selbst wenn man es hätte, könnte man nicht analytisch jede beliebige Frage beantworten, weil die mathematische Formulierung nur näherungsweise zu lösen ist [Deub]. Wettervorhersage zum Beispiel beruht (außer in engen Spezialfällen) immer auf einem analytisch gewonnenen Modell und einer darauf aufbauenden Simulation.

Schoemaker charakterisiert das Verhältnis Analyse/Simulation in [Sch78] so:

> Simulation is not a last resort [...]. If simulation can be used and if analytical models are available as well, this can be very beneficial for the confidence in and the validity of the overall analysis of the object system. Simulation and "classical" models do not exclude each other.

Und Decknatel weist in [Dec99] darauf hin, dass *analytische Methoden* generell *All-Aussagen* machen, *simulierende* dagegen *Existenz-Aussagen*. Das bedeutet, dass das Ergebnis *einer* Simulation eben nur *einen möglichen Ablauf* aufzeigt, das Ergebnis einer Analyse aber (in der Regel) für *alle möglichen Abläufe* des realen Systems gilt. Bezogen auf Eisenbahn-Simulation (siehe Kapitel 5, S. 42) heißt das zum Beispiel: Man könnte analytisch feststellen, dass ein konkretes Gleisnetz eine bestimmte Durchfluss-Kapazität hat, und simulieren, wie ein konkreter Ablauf (Zugfolge, Fahrzeiten, etc.) durch dieses Gleisnetz aussieht.

Sehr schöne Einführungen in das Thema sind: Ferscha [Fer96], Bossel [Bos94] mit Fokus auf Modellbildung, Fujimoto [Fuj90] mit Fokus auf *Discrete Event Simulation* und Fishman [Fis01] mit Gewicht auf Datensammlung, Wahrscheinlichkeitsrechnung und -Verteilung. Den wohl aktuellsten Überblick gibt Chen [Che03].

4.1 Ansätze

Abbildung 4.1 [S. 39] gibt eine Übersicht über die im folgenden dargestellten Verfahren.

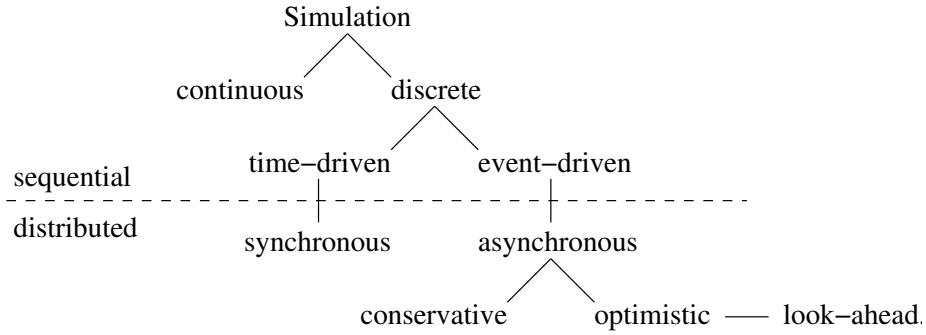


Abbildung 4.1: Simulations-Verfahren

4.1.1 Kontinuierlich vs. Diskret

Die meisten Real-Welt-Systeme sind kontinuierlich: die Zustände des Systems ändern sich kontinuierlich in der Zeit. Zum Beispiel ändert sich das Wetter nicht wirklich sprunghaft, und auch der Eisenbahnverkehr bewegt sich stetig. Es gibt aber auch diskrete dynamische Systeme: zum Beispiel die Entwicklung des Kontostands eines Bank-Kontos.

Um eine Simulation auf einem klassischen Computer berechnen zu können, muss ein Simulationsmodell aber immer irgendwie diskretisiert sein. Typischerweise genügt es, die Zeit in z.B. ganze Sekunden einzuteilen. Einige Simulations-Ansätze nennen sich aber trotzdem *kontinuierlich*, wohl weil sie generell versuchen, kontinuierliche Systeme abzubilden. Siehe zum Beispiel de Lara, Alfonseca, Decknatel und andere [dLA99][APdLO97][Dec99].

4.1.2 Time-driven vs. Event-driven

Diskrete (oder diskretisierte) Systeme werden entweder Zeit-gesteuert (Time-driven) oder Ereignis-gesteuert (Event-driven) simuliert [Fer96]. Zeit-gesteuert wird ein fester Zeittakt vorgegeben, und in jedem Takt wird das System berechnet und beobachtet. Ereignis-gesteuert wird jedes mal, wenn ein Ereignis eintritt, das den Zustand des Systems ändert, dieses Ereignis sofort in das Modell eingerechnet.

Bezogen auf das Bankkonto-Beispiel bedeutet das: Zeit-gesteuert würde man z.B. jeden Tag auf das Konto schauen, und wenn eine neue Buchung vorliegt, diese entsprechend positiv oder negativ einrechnen. Ereignis-gesteuert würde man immer von den Buchungen ausgehen und jede neue sofort verbuchen.

Der Ereignis-gesteuerte Ansatz ist populärer, benötigt aber eben diskrete Ereignisse. Bei der Wetter-Simulation beispielsweise gibt es die nicht, deshalb muss dort Zeit-gesteuert vorgegangen werden. Bei der Eisenbahn-Simulation aber ist ein solches Ereignis beispielsweise das Umschalten eines Signals von Halt auf Fahrt, was zum Anfahren eines wartenden Zuges führen kann.

4.1.3 Parallel und Verteilt

Viele Systeme, die man simulieren will, haben eine Netz-Struktur oder sind quasi natürlich verteilt: Das Wetter ist räumlich ausgedehnt, der Schienenverkehr eines Landes hängt wie ein Netz fast vollständig zusammen. Unter anderem deshalb kann man Simulationen oft gut parallel oder verteilt berechnen.

Das zentrale Problem verteilter Simulation ist Kausalität: Wenn in einem Knoten A ein Ereignis a simuliert wird, kann dieses Auswirkungen auf Teile des Systems haben, die nicht in A simuliert werden.

Relativ einfach lässt sich dieses Problem mit *synchroner Zeit-gesteuerter* [Fer96] Simulation lösen: Alle Knoten haben immer dieselbe Zeit, die in diskreten Schritten fortschreitet. Nach jedem Schritt werden alle Ereignisse zwischen den Knoten kommuniziert, und erst dann wird in allen Knoten der nächste Zeittakt eingeläutet. Diese Methode hat allerdings den erheblichen Nachteil, dass sehr viel Kommunikation stattfindet und die Möglichkeiten paralleler Berechnung sehr eingeschränkt sind.

Mehr Möglichkeiten bietet hier *asynchrone Ereignis-gesteuerte* [Fer96] Simulation: Die virtuellen Zeiten in den Knoten sind nicht synchronisiert, es werden Ereignisse kommuniziert, sobald sie auftreten. Hier werden wieder zwei Ansätze unterschieden: konservativ und optimistisch.

Der konservative Ansatz geht zurück auf das CMB-Protokoll von Chandy, Misra und Bryant [CM79][Bry84][Mis86] und sorgt dafür, dass die Uhren in den Knoten zwar unterschiedlich laufen dürfen, dass aber trotzdem nie ein Kausalitätsfehler auftritt.

Optimistische Ansätze dagegen erlauben solche Fehler und geben Methoden an, diese Fehler zu erkennen und zu beheben [Fuj90]. Hier ist besonders der *Time-Warp* Algorithmus von Jefferson und anderen zu nennen [Jef85]. Er führt so genannte *Rollbacks* ein, die an Backtracking-Suchverfahren (Abschnitt 3.4.6, S. 28) erinnern.

Eine interessante Erweiterung von Time-Warp beschreiben Beraldi und Nigro in [BN00]: Statt diskreter Zeitpunkte benutzen sie Zeit-Intervalle. Sie stellen dort auch eine Implementierung in Java vor, die als *framework for distributed simulations over internet* dienen soll.

4.1.4 Look-Ahead

Ich möchte hier eine neue Kategorie einführen, die der *vorausschauenden* oder *Look-Ahead Simulation*. Solche Simulationen beziehen die Zukunft explizit in die Berechnung der Gegenwart mit ein, um ungünstige System-Zustände – wie Deadlock-Situationen – zu vermeiden. Look-Ahead Simulation ist zwar abgeleitet von der optimistischen verteilten Simulation (Abbildung 4.1, S. 39), jedoch selbst nicht auf verteilte Simulation beschränkt, sondern ein allgemeines Prinzip.

Zum Beispiel für die Simulation von Eisenbahn-Systemen (siehe Kapitel 5, S. 42) entspricht dieses Vorgehen der Wirklichkeit: Auch dort werden von beteiligten Akteuren Situationen in der Zukunft antizipiert, um für die Gegenwart bessere Entscheidungen treffen zu können. So würden Betriebsleiter niemals zwei sich entgegenkommende Züge in eine strikt eingleisige Strecke schicken, weil diese dann ja nicht mehr raus kämen (bzw. durch eine Spezialbehandlung extrem viel Zeit verloren geht).

Die Verwendung von Constraint-Propagation realisiert implizit vorausschauende Simulation: viele Constraint-Variablen beschreiben Simulations-Zeitpunkte und -Zustände und indem der Constraint-Solver immer zwischen allen Variablen propagiert, werden Variablenbelegungen in die Simulations-Zukunft und von der Simulations-Zukunft in die -Vergangenheit propagiert.

4.2 Anwendungen

Simulationen gibt es in allen empirischen Forschungsgebieten. Eine kleine Auswahl:

- Wettervorhersage [Deub].
- Soziale Systeme [Bos94]: Bevölkerungsentwicklung, Wirtschaftsentwicklung, etc.
- Öko-Systeme [Bos94]: Wetter, Umweltbelastung, Waldsterben, Tierpopulationen, Tierbewegung.
- Technische Systeme [Bos94]: Schwingungsverhalten von Bauten und Maschinen, Steuerung und Stabilität von Fahrzeugen, nukleare und chemische Prozesse, etc.
- Technische Großgeräte wie Flugzeuge oder Züge.
- Computer-Netzwerke und -Systeme [ALL96].
- Auto-Verkehr [MRR90][HH98].
- Flucht-Bewegung von Menschen-Massen in Panik [HFV00].
- (Verteilte) Eisenbahn-Simulation: siehe Kapitel 5 [S. 42].

4.3 Einordnung

Bezüglich der genannten Kriterien hat DRS die folgenden Eigenschaften: diskret, Ereignis-gesteuert, verteilt, asynchron, Look-Ahead mit Constraint-Propagation. Wie gesagt werden mit der Constraint-basierten vorausschauenden Simulation neue Wege beschritten.

5 Eisenbahn-Simulation

Simulation hilft gerade beim extrem kapitalintensiven Eisenbahnverkehr, mit relativ wenig Aufwand existierende Netze zu optimieren und neue Konfigurationen zu bewerten. Das folgende Kapitel zeigt, welche Ansätze und konkrete Anwendungen es hier gibt.

5.1 Schienenverkehr

> Der schienengebundene Verkehr verfügt über zwei wesentliche Systemeigenschaften, die die Systemgestaltung maßgebend beeinflussen und in denen er sich insbesondere vom Straßenverkehr unterscheidet: Spurführung und [...] geringe Haftreibung.

So charakterisiert Pachl in [Pac00] den Schienenverkehr. Aus den wesentlichen Eigenschaften folgt, dass im Eisenbahnverkehr – anders als z.B. bei den Straßenbahnen – nicht auf Sicht gefahren werden kann. So hat ein Zug der 160 km/h schnell fährt, einen Bremsweg von etwa 1000m! Daraus wiederum folgt, dass *eine besondere Steuer- und Sicherungstechnik erforderlich* ist [Pac00, 1.1].

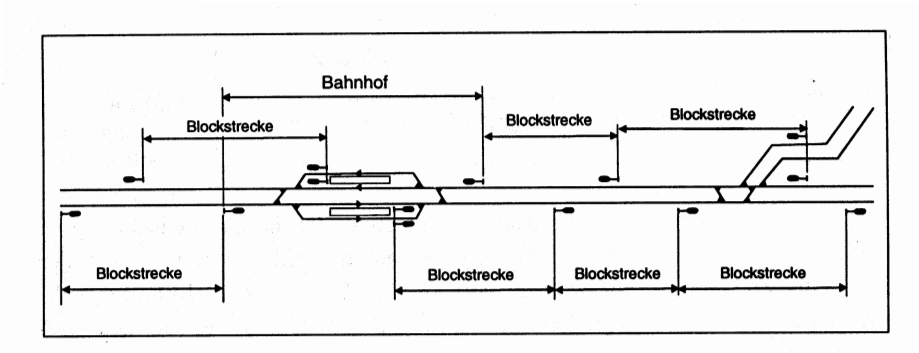


Abbildung 5.1: Blockstrecken oder -Abschnitte [Pac00, 1.3]

Die klassische Sicherungstechnik im Eisenbahnverkehr ist die Blocksicherung, auch *Fahren im festen Raumabstand* genannt [Pac00, 3.2.3]. Sie wurde im 19. Jahrhundert entwickelt und ist bis heute die Standard-Technik. Abbildung 5.1 [S. 42] zeigt exemplarisch Blockstrecken oder -Abschnitte eines kleinen Gleisnetzes. Die Grundregel der Blocksicherung ist [Pac00, 3.2.3.1]:

> In jedem Blockabschnitt darf sich immer nur ein Zug befinden.

5.2 Motivation

Eisenbahnanlagen (*Infrastruktur*) und -Geräte (*Rollmaterial*) sind wie die meisten Verkehrseinrichtungen sehr aufwendig und teuer. Deshalb ist hier der *optimale* Einsatz besonders wichtig.

Ripke formuliert in [Rip00] das Interesse der Deutschen Bahn AG an Entwurfs- und Bewertungswerkzeugen wie folgt:

> Technisch und wirtschaftlich optimierte Fahrwege sind das Ziel der DB AG, um die Wettbewerbsfähigkeit des schienengebundenen Verkehrs gegenüber anderen Verkehrssystemen zu erhöhen. Dieses erfordert neue Methoden, die bereits in der Entwicklung eine technische und wirtschaftliche Bewertung der Fahrbahnkonstruktionen erlaubt.

Was Ripke hier vor allem auf die Fahrbahn bezieht, gilt für viele Bereiche des Schienenverkehrs. Einer der wichtigsten ist der Fahrplan. Der Fahrplan wirkt sich unmittelbar aus auf Kriterien wie Streckenauslastung, Geräteauslastung, Personaleinsatz, und allen voran die Service-Qualität. Pünktlichkeit und Schnelligkeit sind erklärtermaßen die entscheidenden Faktoren im Service-Angebot der Eisenbahn-Gesellschaften.

Für einen optimalen Fahrplan sind entscheidend: Konstruktion und Bewertung. Bewertung wiederum geht a priori – wie wir oben gesehen haben – nur über Analyse und Simulation.

Für spezielle Fälle gibt es exakte Analysemethoden, zum Beispiel [Pac02, 5.3] [BP03][Pöp99][Bra93]. Meist aber erzwingt die Komplexität eines Fahrplans, für zum Beispiel den Personenverkehr der Deutschen Bahn, den Einsatz von Simulationswerkzeugen. Diese sind übrigens oft auch für die Fahrplankonstruktion geeignet, wie wir gleich sehen werden.

Ich verstehe also im Folgenden unter Eisenbahn-Simulation immer Fahrplan-Simulation.

5.3 Ansätze

Auch in der Eisenbahn-Simulation gibt es eine Unterscheidung zwischen asynchroner und synchroner Simulation [Pac00, 5.3.2]. Asynchrone Eisenbahn-Simulation berechnet die Fahrten der einzelnen Züge für jeden Zug getrennt und von den höher zu den niedriger priorisierten Zügen. Eine solche Berechnung ist aber eigentlich dasselbe wie Fahrplankonstruktion. Synchrone Eisenbahn-Simulation geht zeitsynchron vor, wie in der Simulation sonst üblich.

Bezüglich des Simulationsmodells wird generell unterschieden zwischen mikro- und makroskopischen Modellen. Siehe zum Beispiel Decknatel und Schnieder [DS98] und Hauptmann [Hau00, 2.2.2]. Mikroskopische Modelle bilden jeden Gleisabschnitt ab, getrennt durch Signale, Weichen etc. Die Züge werden möglichst exakt anhand ihrer physikalischen Eigenschaften modelliert. Der Betriebsablauf des Systems orientiert sich am Fahrplan, berücksichtigt aber auch Störungen durch Unfälle, Witterung usw. Makroskopisch wird von vielem abstrahiert: Es gibt keine Gleisabschnitte, Signale, Waggons etc. Zum Teil gibt es in solchen Modellen nur Bahnhöfe und Abzweige als Knoten, zwischen denen Strecken als Kanten verlaufen, über die Züge als abstrakte Punkte fahren. Makroskopische Simulation hat den erheblichen Nachteil, dass die Ergebnisse beliebig ungenau sind. Deshalb wird heute fast ausschließlich mikroskopisch modelliert.

Die meisten Simulatoren arbeiten Ereignis-gesteuert, oft explizit mit einem Petri-Netz-Modell. Einen genaueren Einblick vor allem in hybride Petri-Netz-Modelle gibt Decknatel in [Dec99]. Der Text ist auch wegen der Gegenüberstellungen von Analyse und Simulation und von diskreter und kontinuierlicher Simulation im Schienenverkehr besonders interessant.

Sehr empfehlenswert ist in diesem Zusammenhang außerdem der Überblick

über formale Eisenbahn-Modelle von Cordeau, Toth und Vigo in [CTV98], unter anderem für Routing-Probleme, Rangier-Probleme, Zug-Zusammensetzung, Zug-Leitung, Güterzüge.

Es gibt auch erste Ansätze der Constraint-Modellierung von Eisenbahnen: Oliveira untersucht in [dO01][dOS01] das *Single-track Railway Scheduling Problem*, also eingleisige Strecken. Kreuger und andere untersuchen in [KCSÅ97] [KCSÅ01] Fahren auf Abstand. Beide Ansätze sind im Unterschied zu DRS keine generellen Eisenbahn-Simulationen.

Der einzige veröffentlichte Ansatz zur parallelen oder verteilten Eisenbahn-Simulation stammt von Klahn [Kla94]. Dieser Ansatz ist mehr parallel und arbeitet im Unterschied zu DRS mit zeit-synchronisierten Knoten. Dadurch hat er die in Abschnitt 4.1.3 [S. 40] genannten Probleme der eingeschränkten parallelen Möglichkeiten und der extensiven Kommunikation.

Aktuell entwickeln wir im Forschungsprojekt SIMONE [BM02][MB03] einen mikroskopischen Simulator, der sowohl synchrone als auch asynchrone Eisenbahn-Simulation beherrscht. Die vorliegende Arbeit zu DRS findet im Rahmen dieses Projekts statt. SIMONE basiert zu einem Großteil auf Erkenntnissen aus SABINE, siehe unten.

5.4 Anwendungen

5.4.1 Systeme

Im Folgenden beschreibe ich die wichtigsten im deutschsprachigen Raum eingesetzten Fahrplan-Simulations-Systeme und setze sie zueinander in Beziehung.

- FAKTUS (*Fahrplan-Konstruktion und Untersuchung*) [Brü96], RWTH Aachen. Auf der *Grundlage eines Spurplangraphen*, mit einer *integrierten Fahrzeitrechnung* [Brü96], können Belegungszeiten ermittelt werden. Das Verfahren ist zur Fahrplankonstruktion gemacht, kann aber (siehe oben) auch zur asynchronen Simulation eingesetzt werden.

- RWS-1 (*Railway-Simulation*) [Gig89]. Synchrone mikroskopische Simulation. Das System basiert auf einem Gleis-Graphen, Züge und Sicherungstechnik werden Ereignis-basiert berechnet. Ein recht frühes System, auf das sich viele später beziehen. Entstanden am Verkehrs-Institut IVT der Eidgenössichen Technischen Hochschule ETH Zürich.

- OpenTrack [Hür01][Hür]. Der Nachfolger von RWS. Eine Objekt-orientierte Neuentwicklung, ebenfalls vom IVT der ETH. Es benutzt eine neue Graph-Darstellung für Gleisnetze [Mon94]. OpenTrack simuliert synchron und mikroskopisch. OpenTrack hat eine besonders übersichtliche graphische Oberfläche incl. Gleisplan-Editor und ist v.a. für sehr genaue Simulationen kleinerer Netze gedacht, bei denen viele Alternativen getestet werden müssen.

- Simu VII [Kla94][Reh98]. Mikroskopische synchrone Simulation, laut [Reh98] für größere Anwendungen geeignet. Am Institut für Verkehrswesen, IVE, der Universität Hannover entwickelt. Simu VII basiert auf Simu V, das dort seit den siebziger Jahren entstand und genau wie Simu VII in Fortran realisiert ist.

- PC-Simu, UX-Simu, laut [Kla94, 2.6] ebenfalls basierend auf Simu V. Diese Verfahren wurden bis ca. 2000 von HaCon [HaC] vertrieben.
- RailSys/Simu++ [Rai][Hau00]. RailSys ist eine integrierte Software mit den Modulen *Infrastructure Management, Timetable Management* und *Simulation*. Das Simulations-Modul ist Simu++ [Hau00], eine Weiterentwicklung von Simu VII, ebenfalls vom IVE Hannover, jetzt basierend auf C++.
- SABINE. Laut [DS98] ein makroskopisches, synchrones Simulationsverfahren, basierend auf *zeitbewerteten hierarchischen Petri-Netzen* [DS98].
- TRANSIT [NR96][Reh98]. Eine interne Entwicklung von Siemens, zur Entwicklung und Bewertung eigener Schienenverkehrs-Systeme. TRANSIT ist laut [Reh98] Simu VII recht ähnlich, legt aber weniger Wert auf die mögliche Größe des Simulationsmodells, dafür mehr auf den Detaillierungsgrad der Simulation.
- FALKO (*Fahrplan-Validierung und -Konstruktion für spurgebundene Verkehrssysteme*) [ESS00]. Besitzt einen synchronen mikroskopischen Simulator für die Betriebssimulation. *Bei den Komponenten Fahrplan-Konstruktion und Umlaufplanung werden Methoden der diskreten Optimierung (Operations Research) eingesetzt* [ESS00]. FALKO basiert auf TRANSIT.

5.4.2 Andere Anwendungen

Im Schienenverkehr werden aber nicht nur die Fahrpläne simuliert bzw. der Fahrbetrieb. Es gibt hier eine ganze Reihe anderer Simulations-Anwendungen:

- Rangierbetrieb in Ablaufanlagen [Got01], z.B. das System HaCon RASIM [RAS].
- Leistungsverhalten unterschiedlicher Signaltechniken [Osb02].
- Sperrzeitverhalten von Bahnübergängen [BP02a].
- Stellwerk-Simulation, Elektronisches Stellwerk ESTW
  - System BEST [BPS01].
  - Ausbildung am ESTW [Har01].
  - Stellwerks- und Betriebssimulation für die Fahrdienstleiter-Schulung [BP02b].
  - Im Nahverkehr: Siemens SICAS [Lor01].
- Schotterbau [KPMS01]: *Mittels der Molekulardynamischen Simulation lässt sich beim Schottergleis jeder einzelne Schotterstein im Modell nachbilden.*
- Gleisanlagen-Instandhaltung [Krü01].
- Live-Cycle-Costs, LCC-Methode für Fahrbahnen [Kor00].
- Fahrzeug/Fahrweg-Interaktion [HGTZ02].
- Rad-Schiene-Rollkontakt per Finite-Element-Methoden [NZ01].

- Fahrzeugbau, Kollissionssicherheit: Projekt SAFETRAIN [Wol01b].
- Genaue Fahrdynamik zur Ermittlung des Energiebedarfs für Zugfahrten [Jen00].
- Analyse individueller Risiken an Bahnanlagen (v.a. Bahnübergänge) [BL00].
- Verschleiß von Rad- und Schienenprofilen [KMM03].
- Strategische Simulation für die strategische Planung
  - Verkehrsnachfragemodell AVENA [BKL03].
  - Risiko-Management [BS03].
  - Oberbauinstandhaltung [Jov03].
- Intermodaler Verkehr
  - Güterverkehrszentren [KK00].
  - Verkehrstelematik [BHK03].
  - Auswirkungen auf die Umwelt [GK00].

5.5 Einordnung

DRS realisiert eine mikroskopisch genaue Fahrplansimulation. Je nach Auslegung des lokalen Simulators ist diese (bzgl. Fahrplan-Simulation) asynchron bis vorausschauend synchron und kann daher auch für die Fahrplan-Konstruktion verwendet werden.

Bezüglich Eisenbahn-Simulation werden mit DRS zum einen mit der Constraint-basierten vorausschauenden Simulation und zum anderen mit der Verteilung neue Wege beschritten.

6 Synthetisches Beispiel

Zur Veranschaulichung folgt hier ein kleines künstliches Simulations-Beispiel. Dieses ist weit weniger komplex als das Fallbeispiel, das wir für die empirischen Bewertungen weiter unten verwenden und das auf realen Daten basiert. Das synthetische Beispiel leitet in den nächsten Teil über, in welchem es zur Veranschaulichung der formalen Konstruktionen der theoretischen Betrachtungen verwendet wird.

6.1 Infrastruktur

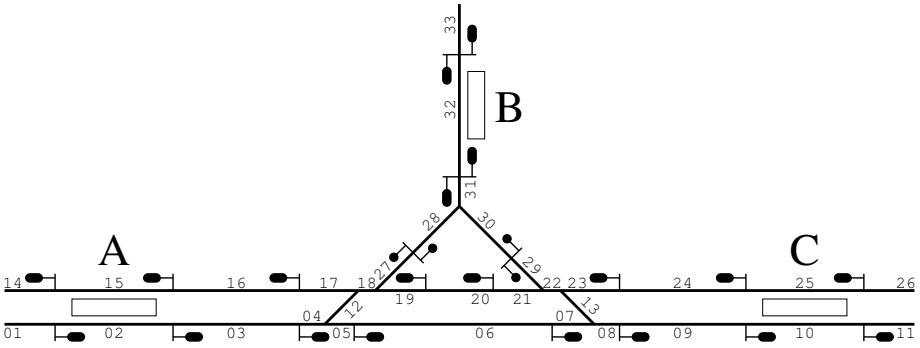


Abbildung 6.1: Synthetisches Beispiel.

Betrachte Abbildung 6.1. Dieses Beispiel soll uns im Folgenden begleiten und die einzelnen Sachverhalte illustrieren.

Das Bild beschreibt ein kleines Gleisnetz mit drei Bahnhöfen. Die Gleise sind durch dicke Linien dargestellt, die Bahnhöfe gekennzeichnet: A, B und C. Die Rechtecke zwischen den Gleisen symbolisieren die Bahnsteige, A und C haben zwei Gleise mit Bahnsteig, B nur eins. Die Signale an den Gleisen sind durch ausgefüllte Ovale dargestellt.

Zwischen den Bahnhöfen A und C ist die Strecke zweigleisig, auf dem oberen Gleis darf nur (in Richtung) von C nach A gefahren werden, unten nur von A nach C. Hier gilt also Rechtsverkehr. Die Signale stehen immer in Fahrtrichtung rechts neben dem Gleis. Der Standpunkt des Signals wird in der Abbildung durch eine dünne Verbindungslinie zwischen Signal und Gleis gekennzeichnet. Verzweigungen sind durch sich berührende Gleise dargestellt, an jeder Abzweigung befindet sich in der Realität eine Weiche. Alle Gleisabschnitte sind durchnummeriert. Jeder Gleisabschnitt ist ein Stück Gleis, das vorne und hinten durch ein Signal oder eine Weiche oder ein Gleisende begrenzt ist. Dadurch hat dieses Beispiel 33 Gleisabschnitte: 01-33.

Dieses Beispiel ist natürlich gegenüber der Realität vereinfacht. In Wirklichkeit gibt es zum Beispiel nicht nur eine Sorte Signale, weitere Sicherheitseinrichtungen wie Zugsschlussstellen, Geschwindigkeitsbegrenzungen und komplexe Gleisverbindungen. Für unsere Zwecke aber genügt diese Abstraktion.

Die für die Sicherung nötigen Block- oder Belegungsabschnitte ergeben sich jeweils aus zusammenhängenden Gleisabschnitten, die von Signalen begrenzt werden. Wir haben deshalb folgende Blockabschnitte: 01, 02, 03, 04/05, 06,

07/08, 09, 10, 11, 14, 15, 16, 17/18/19, 20, 21/22/23, 24, 25, 26, 04/12/18/27, 17/18/27, 08/13/22/29, 22/23/29, 28/31, 30/31, 32 und 33. Die Reihenfolge der Zahlen in den Bezeichnern für die Blockabschnitte spielt übrigens keine Rolle. Ich habe sie der besseren Lesbarkeit willen aufsteigend notiert.

6.2 Züge

Außerdem fahren drei Züge durch das Netz: AC von A nach C, CA von C nach A, und ABC von A über B nach C. AC benützt folgenden *Laufweg* (i.e. Folge von Belegungsabschnitten): 02 - 03 - 04/05 - 06 - 07/08 - 09 - 10. CA fährt über: 25 - 24 - 21/22/23 - 20 - 17/18/19 - 16 - 15. Und ABC fährt: 02 - 03 - 04/12/18/27 - 28/31 - 32 - 30/31 - 08/13/22/29 - 09 - 10. Hier spielt die Reihenfolge der Belegungsabschnitte eine Rolle: der Zug durchfährt sie in der angegebenen Reihenfolge.

AC und CA kommen sich nicht in die Quere. ABC aber kann mit AC und CA in Konflikt stehen: Wenn ABC den Block 04/12/18/27 befährt, darf kein anderer Zug in den Block einfahren, deshalb müssen dann sofort die Blöcke 04/05, 04/12/18/27, 17/18/19 und 17/18/27 für andere Züge gesperrt werden, indem jeweils das unmittelbar vorstehende Signal auf *Halt* geht.

Daraus lässt sich auch leicht folgern, warum die beiden Signale zwischen 21 und 19 sinnvoll sind: Wenn der Zug CA im Abschnitt 20 steht, kann der Zug ABC trotzdem sowohl über den Block 04/12/18/27 als auch über 08/13/22/29 fahren. Wenn das Signal zwischen 20 und 21 fehlte, könnte ABC nicht gleichzeitig über 08/13/22/29 fahren.

6.3 Sperrzeiten

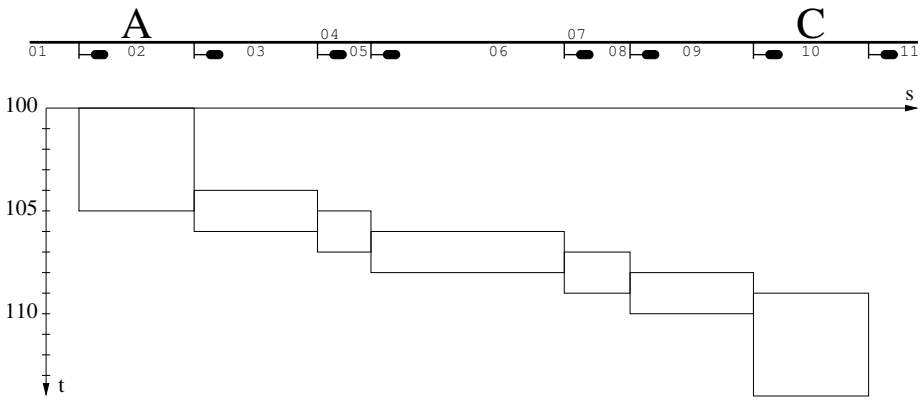


Abbildung 6.2: Sperrzeitentreppe für den Zug AC des synthetischen Beispiels.

Abbildung 6.2 [S. 48] zeigt eine mögliche Sperrzeitentreppe für den Zug AC. Die Treppe wird durch die graphische Darstellung der Belegungszeiten für alle Belegungsabschnitte gebildet. Wir nehmen hier an, dass der Zug durch jeden Abschnitt genau zwei Zeiteinheiten fährt und in jedem Bahnhof fünf Zeiteinheiten stehen bleibt. Die Belegungszeiten aufeinander folgender Blöcke überlappen

sich um jeweils genau eine Zeiteinheit. Letzteres bildet zum einen real existierende Vor- und Nachbelegungszeiten abstrakt nach. Andererseits verhindert die Überlappung in unserer Abstraktion, dass entgegenkommende Züge quasi *durcheinander durchfahren* können.

Der Zug AC steht also vom Zeitpunkt $t = 100$ bis $t = 104$ im Bahnhof A, verlässt dann den Abschnitt 02 und tritt in den Abschnitt 03 ein. Der Abschnitt 02 bleibt bis $t = 105$ belegt. Nachdem AC durch 03 gefahren ist, betritt er den Abschnitt 04/05, usw. Die Abschnitte bleiben jeweils für genau zwei Zeiteinheiten belegt. Schließlich kommt AC zu $t = 109$ in AC an und verschwindet dort nach weiteren fünf Zeiteinheiten. In Simulationen können Züge typischerweise einfach erscheinen oder verschwinden. In der Realität würde der Zug vielleicht weiterfahren oder auf ein Abstellgleis gestellt.

Übrigens: In der Sperrzeitentreppe bedeutet die Grundregel der Blocksicherung, dass sich nie zwei Belegungsblöcke auf ein und demselben *Gleisabschnitt* überlappen dürfen. Diese Eigenschaft werden wir gleich noch in der Theorie benötigen.

Teil II

Verteilter Algorithmus DRS

Dieser Teil beschreibt den verteilten Simulationsalgorithmus DRS und beleuchtet ihn theoretisch. Das ist damit der Kern dieser Arbeit.

Hier wird gezeigt, dass der verteilte Simulationsalgorithmus korrekt ist in dem Sinne, dass jedes Ergebnis, das er liefert, korrekt ist und dass er immer in endlicher Zeit terminiert. Die Schritte dahin sind folgende: ausgehend von einem beliebigen Simulationsproblem Π muss jede Simulation Σ in endlicher Zeit eine Lösung Solution liefern:

$$\Pi \xrightarrow{\Sigma} \text{Solution}$$

Dafür wird in Kapitel 8 [S. 54] definiert, was ein Simulationsproblem Π ist und in Kapitel 9 [S. 60], wie die Lösungsmenge Solution eines solchen Problems aussieht. In Kapitel 10 [S. 66] wird definiert, wie eine lokale Simulation Σ_p aussieht, und in Kapitel 11 [S. 71] die darauf aufbauende verteilte Simulation Σ beschrieben. Kapitel 12 [S. 76] definiert den Begriff der Konvergenz und wie Konvergenz von Simulationen zu Terminierung und Korrektheit führt. Die Kapitel 13 [S. 79] und 14 [S. 87] zeigen, warum Σ immer konvergiert. Kapitel 15 [S. 90] folgert daraus Korrektheit und Terminierung. Kapitel 16 [S. 92] schließlich fasst diesen Teil zusammen und diskutiert die theoretischen Ergebnisse.

7 Notation

In diesem Kapitel werden einige formale Schreibweisen definiert, insbesondere solche, die in der Literatur nicht einheitlich sind.

Definition 7.1 (Abbildung)
Eine *Abbildung* $f : A \to B$ ist hier immer *linkstotal* und *rechtseindeutig*:

(1) $f \subseteq A \times B$

(2) $\forall a \in A : \exists b \in B : (a, b) \in f$

(3) $\forall a \in A, b, b' \in B : ((a, b) \in f \wedge (a, b') \in f) \Rightarrow b = b'$

Für die Elemente der Abbildung schreibe ich äquivalent: $(a \mapsto b) \in f \Leftrightarrow (a, b) \in f$.

Definition 7.2 (Einschränkung einer Abbildung)
Wenn $f : A \to B$ eine Abbildung ist und $A' \subseteq A$, dann ist $f{\downarrow}A'$ die *Einschränkung* von f auf A':

(1) $f{\downarrow}A' : A' \to \{b \in B \mid \exists a \in A' : f(a) = b\}$

(2) $\forall a \in A' : f{\downarrow}A'(a) := f(a)$

Diese Definition der Einschränkung erhält die Bijektivität von Abbildungen!

Definition 7.3 (Umkehrabbildung)
Wenn $f : A \to B$ eine Abbildung ist, dann ist $f^{-1} : B \to A$ die *Umkehrabbildung*: $f^{-1} := \{(b, a) \mid (a, b) \in f\}$. Natürlich ist nicht für jedes f die Umkehrabbildung f^{-1} auch definiert, f^{-1} muss schließlich auch eine *Abbildung* (nach Definition 7.1, S. 52) sein.

Definition 7.4 (Mengenabbildung)
Wenn $f : A \to B$ eine Abbildung ist und $A' \subseteq A$, dann ist $f(A') := \{b \in B \mid \exists a \in A' : f(a) = b\}$.

Definition 7.5 (Bild einer Abbildung)
Wenn $f : A \to B$ eine Abbildung ist, dann ist $\mathrm{Bi}(f)$ das *Bild* von f: $\mathrm{Bi}(f) := f(A)$.

Definition 7.6 (Potenzmenge)
Wenn A eine Menge ist, dann ist 2^A die *Potenzmenge* von A: $2^A := \{A' \mid A' \subseteq A\}$.

Definition 7.7 (Tupel, Konkatenation, Projektion)
Ein *Tupel* ist eine indizierte Menge von Elementen. Formal ist ein Tupel $S = (a \in A)_{b \in B}$ eine Abbildung $S : B \to A$ mit $S(b) = a_b$. Tupel sind über beliebigen – auch unendlichen – Index- und Zielmengen definiert. Im Folgenden sind alle (potentiellen) Indexmengen disjunkt, sodass die Indizierung aus diesen Mengen eindeutig ist, zum Beispiel: $\mathsf{P} \cap \mathbb{N} = \emptyset, p \in \mathsf{P}, i \in \mathbb{N} \Rightarrow a_p \neq a_i$.

Für ein gegebenes Tupel $S : B \to A$ ist auch ein beliebiger Ausschnitt über einer Teilmenge $B' \subseteq B$ der Indexmenge als Tupel definiert, nämlich durch die

eingeschränkte Abbildung $S{\downarrow}B'$. Für zwei Ausschnitte B' und B'' mit $B' \cap B'' = \emptyset$ ist die *Konkatenation* $(a)_{b \in B'} \circ (a)_{b \in B''}$ definiert durch $S{\downarrow}B' \cup S{\downarrow}B''$.

Projektion ist die Anwendung der Funktion S auf ein Element. Ich schreibe dafür meist $a_b := S(b)$.

Die Zielmenge eines Tupels ist beliebig und kann das Kreuzprodukt einer beliebigen Anzahl beliebiger Mengen sein. Ich schreibe dann zum Beispiel: $(a_d, b_d, c_d)_{d \in D} := D \to A \times B \times C$.

Definition 7.8 (Menge der natürlichen Zahlen)
$\mathbb{N}$ ist die *Menge der natürlichen Zahlen* einschließlich der Null, $\mathbb{N}_+$ ohne die Null.

Bemerkung 7.9 (Allgemeines zur Notation)
Um eine gute Lesbarkeit der Formeln zu erreichen, beginnen Bezeichner für Mengen immer mit Großbuchstaben, genauso solche für Abbildungen, deren Ergebnis eine Menge ist (also $F : A \to 2^B$). Damit ist in folgenden Formeln immer $a \in A$ oder $f(a) \in F(b)$.

Für die im folgenden definierten Konstrukte verwende ich im wesentlichen eine eigene Schrifttype (zum Beispiel start_t, $\mathsf{S}_{t,p}$), vereinzelt auch Griechische Buchstaben (Simulationsprobleme Π und Simulationen Σ).

8 Simulationsprobleme

Hier wird definiert, wie ein Simulationsproblem Π aussieht: Es besteht im wesentlichen aus Zügen, Belegungsabschnitten, Problemteilen und Belegungszeiten. Außerdem wird am synthetischen Beispiel gezeigt, wie die abstrakten Konstruktionen konkret aussehen können.

Definition 8.1 (Simulationsproblem Π)
Ein *Simulationsproblem* Π – manchmal auch kurz *Problem* genannt – ist ein Tupel $(\mathsf{T}, \mathsf{S}, \mathsf{P}, \mathsf{tno}, \mathsf{part}, \mathsf{excl}, (\mathsf{S}_t, \mathsf{P}_t, \mathsf{sno}_t, \mathsf{pno}_t, \mathsf{mindur}_t, \mathsf{minend}_t)_{t \in \mathsf{T}})$ mit

(1) T
Menge der Züge (*Trains*), endlich.

(2) S
Menge der Belegungsabschnitte (*Sections*), endlich.

(3) P
Menge der Simulationsteile (*Parts*), endlich.

(4) $\mathsf{T}, \mathsf{S}, \mathsf{P}, \mathbb{N}$ paarweise disjunkt

(5) $\mathsf{tno} : \mathsf{T} \to \{1, ..., |\mathsf{T}|\} \subseteq \mathbb{N}$, bijektiv
Für jeden Zug eine eineindeutige Nummer.

(6) $\mathsf{part} : \mathsf{S} \to \mathsf{P}$
Für jeden Belegungsabschnitt den Simulationsteil.

(7) $\mathsf{excl} \subseteq \mathsf{S} \times \mathsf{S}$
excl beschreibt die sich ausschließenden Belegungsblöcke. Wie wir in Kapitel 6 [S. 47] gesehen haben, dürfen sich Belegungszeiten auf einem *Gleisabschnitt* nicht überlappen. Hier abstrahieren wir von Gleisabschnitten. excl beschreibt entsprechend die Belegungsabschnitte, die sich zeitlich nicht überlappen dürfen.

(8) $\mathsf{S}_t \subseteq \mathsf{S}, \mathsf{S}_t \neq \emptyset$
Für jeden Zug die Belegungsabschnitte, die er befährt.

(9) $\mathsf{P}_t := \mathsf{part}(\mathsf{S}_t)$
Für jeden Zug die Simulationsteile, die er passiert.

(10) $\mathsf{sno}_t : \mathsf{S}_t \to \{1, ..., |\mathsf{S}_t|\} \subseteq \mathbb{N}$, bijektiv
Pro Zug für jeden Belegungsabschnitt eine eineindeutige Nummer.

(11) $\mathsf{pno}_t : \mathsf{P}_t \to \{1, ..., |\mathsf{P}_t|\} \subseteq \mathbb{N}$, bijektiv
Pro Zug für jeden Simulationsteil eine eineindeutige Nummer.

(12) $\mathsf{mindur}_t : \mathsf{S}_t \to \mathbb{N}$
Pro Zug für jeden Belegungsabschnitt die *minimale* Dauer, die der Zug den Abschnitt belegen muss.

(13) $\mathsf{minend}_t : \mathsf{S}_t \to \mathbb{N}$
Der Fahrplan: Pro Zug für jeden Belegungsabschnitt der Zeitpunkt, zu dem der Zug dort abfahren soll.

(14) $\forall t \in \mathsf{T} : \forall s, s' \in \mathsf{S}_t : \mathsf{pno}_t(\mathsf{part}(s)) < \mathsf{pno}_t(\mathsf{part}(s')) \Rightarrow \mathsf{sno}_t(s) < \mathsf{sno}_t(s')$
Für beliebige $s, s' \in \mathsf{S}_t$ steigen oder fallen sno_t und pno_t monoton. Die Abbildungen sno_t und pno_t sind also quasi synchronisiert.

(15) $\forall t \in \mathsf{T} : \forall s, s' \in \mathsf{S}_t :$
$|\mathsf{sno}_t(s) - \mathsf{sno}_t(s')| = 1 \Rightarrow |\mathsf{pno}_t(\mathsf{part}(s)) - \mathsf{pno}_t(\mathsf{part}(s'))| \leq 1$
Aufeinander folgende Belegungsabschnitte gehören entweder zu demselben Simulationsteil oder zu aufeinander folgenden Simulationsteilen.

Bemerkung 8.2

- Ich benutze die paarweise disjunkten Mengen $\mathsf{T}, \mathsf{S}, \mathsf{P}, \mathbb{N}$ oft zur Indizierung. In der Regel geht aus dem Kontext klar hervor, aus welcher Menge der Index stammt und damit, was der Index bedeutet. Indem diese Indexmengen aber paarweise disjunkt sind, kann auch formal nichts schief gehen.
- Die Tatsache, dass part eine Abbildung ist, impliziert, dass die Aufteilung der Belegungsabschnitte auf die Simulationsteile eine Partition ist.
- sno_t und pno_t sind Bijektionen, deshalb sind jeweils die Umkehrfunktionen eindeutig definiert.
- In der Anwendung hat ein Fahrplan nicht für jeden Belegungsabschnitt einen Eintrag, sondern nur für ganz bestimmte (v.a. in Bahnhöfen). Für alle anderen sei $\mathsf{minend}_t(s) = 0$.
- Hier wird das Gleisnetz nicht explizit definiert, wir abstrahieren ja von den Gleisabschnitten.
- Der Fahrplan wird hier bezüglich der *Abfahrtzeit* der Züge, v.a. in den Bahnhöfen berücksichtigt (minend), nicht aber bezüglich der *Ankunftzeit*. Das kommt im wesentlichen davon, dass man nur bzgl. der Abfahrtzeit eine echt einzuhaltende Bedingung formulieren kann: Der Zug darf auf keinen Fall vor der planmäßigen Abfahrtzeit abfahren. Er darf aber natürlich früher ankommen als fahrplanmäßig vorgesehen und wird oft auch später ankommen. Die Ankunftzeit ist also *möglichst einzuhalten*. In unserem Modell der harten Bedingungen spielt die Ankunftzeit damit zunächst keine Rolle. Man kann sie in der Anwendung aber in ein Optimierungskriterium und die Simulationsheuristik einfließen lassen.
- Die Tatsache, dass sno_t eine Abbildung ist, impliziert, dass jeder Zug jeden Belegungsabschnitt höchstens einmal benutzt (s.a. Diskussion in Abschnitt 16.1 [S. 92]).
- Im Folgenden setze ich ein beliebiges Π meist als gegeben voraus. Wenn ich von Teilen der Struktur spreche, also zum Beispiel von $t \in \mathsf{T}, \mathsf{P}_t$, und kein spezielles Π (wie unten Π^{bsp}) angegeben ist, beziehen sich diese immer auf das gegebene Π.

Beispiel 8.3

Betrachten wir das synthetische Beispiel aus Kapitel 6 [S. 47] in einer leicht modifizierten Form: In Abbildung 8.1 [S. 56] ist dieses partitioniert, die Zerlegung in die drei Teile A, B und C ist mit gestrichelten Linien gekennzeichnet. Durch

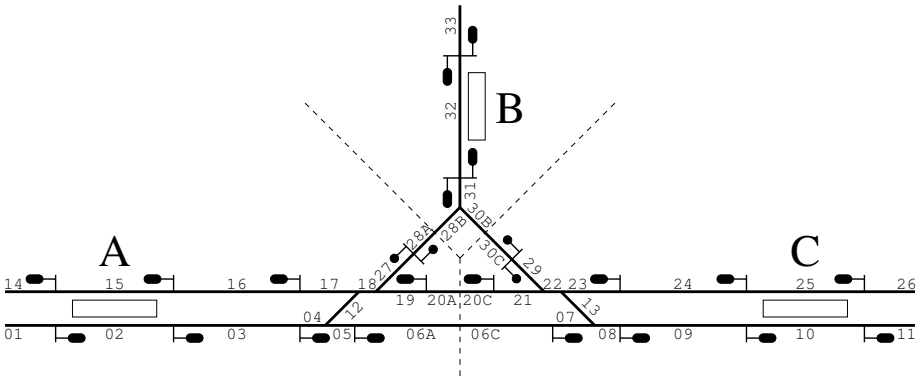


Abbildung 8.1: Nochmal das synthetische Beispiel aus Abbildung 6.1 [S. 47]. Hier aber mit der Partitionierung in die Teile A, B und C.

die Zerlegung werden die Gleisabschnitte 06, 21, 28 und 30 geteilt in jeweils zwei Teile. Das ist notwendig, weil die Belegungsabschnitte eindeutig auf die Teile verteilt werden müssen. Wie wir später noch sehen werden, fügt der verteilte Simulationsalgorithmus die zerteilten Abschnitte virtuell wieder zusammen, indem er die Belegungszeiten entsprechend unifiziert.

Damit sieht Π^{bsp} wie folgt aus:

$\mathsf{T} = \{\text{AC}, \text{CA}, \text{ABC}\}$

$\mathsf{S} = \{01, 02, 03, 04/05, 06\text{A}, 06\text{C}, 07/08, 09, 10, 11, 14, 15, 16, 17/18/19, 20\text{A}, 20\text{C}, 21/22/23, 24, 25, 26, 04/12/18/27, 17/18/27, 08/13/22/29, 22/23/29, 28\text{A}, 28\text{B}/31, 30\text{B}/31, 30\text{C}, 32, 33\}$

$\mathsf{P} = \{\text{A}, \text{B}, \text{C}\}$

$\mathsf{tno} = \{(\text{AC}, 1), (\text{CA}, 2), (\text{ABC}, 3)\}$

$\mathsf{part} = \{(01,\text{A}), (02,\text{A}), (03,\text{A}), (04/05,\text{A}), (06\text{A},\text{A}), (06\text{C},\text{C}), (07/08,\text{C}), (09,\text{C}), (10,\text{C}), (11,\text{C}), \ldots, (28\text{A},\text{A}), (28\text{B}/31,\text{B}), (30\text{B}/31,\text{B}), (30\text{C},\text{C}), (32,\text{B}), (33,\text{B})\}$

$\mathsf{excl} = \{(s,s)|s \in \mathsf{S}\} \cup E \cup \{(s,s')|(s',s) \in E\}$ mit $E := \{$
$(04/05,04/12/18/27), (17/18/19,04/12/18/27),$
$(17/18/27,04/12/18/27), (17/18/19,17/18/27),$
$(07/08,08/13/22/29), (21/22/23,08/13/22/29),$
$(22/23/29,08/13/22/29), (21/22/23,22/23/29),$
$(28\text{B}/31,30\text{B}/31)\ \}$

$\mathsf{S}_{\text{AC}} = \{02, 03, 04/05, 06\text{A}, 06\text{C}, 07/08, 09, 10\}$

$\mathsf{P}_{\text{AC}} = \{\text{A}, \text{C}\}$

$\mathsf{sno}_{\text{AC}} = \{(02,1),(03,2),(04/05,3),(06\text{A},4),(06\text{C},5),(07/08,6),(09,7),(10,8)\}$

$\mathsf{pno}_{\text{AC}} = \{(\text{A}, 1), (\text{C}, 2)\}$

$$\mathsf{mindur}_{\mathrm{AC}}(s) = \begin{cases} 5 & s \in \{02, 10\} \text{(im Bahnhof, s. Abschnitt 6.3, S. 48)} \\ 2 & \text{sonst} \end{cases}$$

$$\mathsf{minend}_{\mathrm{AC}}(s) = \begin{cases} 105 & s = 02 \\ 0 & \text{sonst} \end{cases}$$

Für den Endbahnhof (s=10) haben wir keine explizite Abfahrtzeit vorgegeben, der Zug AC verlässt hier die Simulation. Warum der Wert 105 für den Abfahrtsbahnhof gewählt wurde, werden wir später noch sehen (Abbildung 9.1, S. 62).

$\mathsf{S}_{\mathrm{CA}} = \{25, 24, 21/22/23, 20\mathrm{C}, 20\mathrm{A}, 17/18/19, 16, 15\}$

$\mathsf{P}_{\mathrm{CA}} = \{\mathrm{C}, \mathrm{A}\}$

$\mathsf{sno}_{\mathrm{CA}} = \{(25{,}1),(24{,}2),(21/22/23{,}3),(20\mathrm{C}{,}4),(20\mathrm{A}{,}5),(17/18/19{,}6),(16{,}7),(15{,}8)\}$

$\mathsf{pno}_{CA} = \{(C, 1), (A, 2)\}$

$$\mathsf{mindur}_{\mathrm{CA}}(s) = \begin{cases} 5 & s \in \{15, 25\} \\ 2 & \text{sonst} \end{cases}$$

$$\mathsf{minend}_{\mathrm{CA}}(s) = \begin{cases} 105 & s = 25 \\ 0 & \text{sonst} \end{cases}$$

$\mathsf{S}_{\mathrm{ABC}} = \{$ 02, 03, 04/12/18/27, 28A, 28B/31, 32, 30B/31, 30C, 08/13/22/29, 09, 10 $\}$

$\mathsf{P}_{\mathrm{ABC}} = \{\mathrm{A}, \mathrm{B}, \mathrm{C}\}$

$\mathsf{sno}_{\mathrm{ABC}} = \{$ (02,1), (03,2), (04/12/18/27,3), (28A,4), (28B/31,5), (32,6), (30B/31,7), (30C,8), (08/13/22/29,9), (09,10), (10,11) $\}$

$\mathsf{pno}_{\mathrm{ABC}} = \{(\mathrm{A}, 1), (\mathrm{B}, 2), (\mathrm{C}, 3)\}$

$$\mathsf{mindur}_{\mathrm{ABC}}(s) = \begin{cases} 5 & s \in \{02, 10, 32\} \\ 2 & \text{sonst} \end{cases}$$

$$\mathsf{minend}_{\mathrm{ABC}}(s) = \begin{cases} 120 & s = 02 \\ 127 & s = 32 \\ 0 & \text{sonst} \end{cases}$$

Definition 8.4 (first_t,last_t)
Ich definiere als abkürzende Schreibweisen first_t und last_t, jeweils den ersten und letzten Abschnitt eines Zuges in jedem Teil:

(1) $\mathsf{first}_t : \mathsf{P}_t \to \mathsf{S}_t, \mathsf{first}_t(p) := {\mathsf{sno}_{t,p}}^{-1}(\min(\mathrm{Bi}(\mathsf{sno}_{t,p})))$

(2) $\mathsf{last}_t : \mathsf{P}_t \to \mathsf{S}_t, \mathsf{last}_t(p) := {\mathsf{sno}_{t,p}}^{-1}(\max(\mathrm{Bi}(\mathsf{sno}_{t,p})))$

Beispiel 8.5
Für Π^{bsp} sehen diese Abbildungen wie folgt aus:

$\mathsf{first}_{\mathrm{AC}} = \{(\mathrm{A}, 02), (\mathrm{C}, 06\mathrm{C})\}$

$\mathsf{last}_{\mathrm{AC}} = \{(\mathrm{A}, 06\mathrm{A}), (\mathrm{C}, 10)\}$

$\mathsf{first}_{\mathrm{CA}} = \{(\mathrm{C}, 25), (\mathrm{A}, 20\mathrm{A})\}$

$\mathsf{last}_{\mathrm{CA}} = \{(\mathrm{C}, 20\mathrm{C}), (\mathrm{A}, 15)\}$

$\mathsf{first}_{\mathrm{ABC}} = \{(\mathrm{A}, 02), (\mathrm{B}, 28\mathrm{B}), (\mathrm{C}, 30\mathrm{C})\}$

$\mathsf{last}_{\mathrm{ABC}} = \{(\mathrm{A}, 28\mathrm{A}), (\mathrm{B}, 30\mathrm{B}), (\mathrm{C}, 10)\}$

Lemma 8.6
Wie wir in Bemerkung 8.2 [S. 55] bereits gesehen haben, benutzt jeder Zug jeden Belegungsabschnitt höchstens einmal. Genauso benutzt jeder Zug jeden Teil höchstens einmal.

Beweis: Zu zeigen ist, dass kein Zug t auf seiner Strecke einen Teil verlässt und ihn später wieder betritt. Gegeben $s, s', s'' \in \mathsf{S}_t, s \neq s' \neq s'' \neq s, p, p' \in \mathsf{P}_t, \mathsf{part}(s) = p, \mathsf{part}(s') = p', \mathsf{part}(s'') = p$. O.B.d.A. sei $\mathsf{sno}_t(s) \leq \mathsf{sno}_t(s') \leq \mathsf{sno}_t(s'')$. t durchfährt also auf seinem Weg über s, s', s'' die Teile p, p', p. Es folgt mit Definition 8.1 (14) [S. 55]: $\mathsf{pno}_t(p) \leq \mathsf{pno}_t(p') \leq \mathsf{pno}_t(p) \Rightarrow \mathsf{pno}_t(p) = \mathsf{pno}_t(p') \Rightarrow p = p'$. □

Definition 8.7 (Problem-Teile P)
Ich definiere abkürzende Schreibweisen für die Teile $p \in \mathsf{P}$ eines Problems Π:

(1) $\mathsf{T}_p := \{t \in \mathsf{T} \mid p \in \mathsf{P}_t\}$

(2) $\mathsf{S}_p := \{s \in \mathsf{S} \mid \mathsf{part}(s) = p\}$

(3) $\mathsf{S}_{t,p} := \mathsf{S}_t \cap \mathsf{S}_p$

(4) $\mathsf{sno}_{t,p} := \mathsf{sno}_t{\downarrow}\mathsf{S}_{t,p}$

(5) $\mathsf{mindur}_{t,p} := \mathsf{mindur}_t{\downarrow}\mathsf{S}_{t,p}$

(6) $\mathsf{minend}_{t,p} := \mathsf{minend}_t{\downarrow}\mathsf{S}_{t,p}$

Beispiel 8.8
Für Π^{bsp} und $p = \mathrm{A}$ sehen diese wie folgt aus:

$\mathsf{T}_{\mathrm{A}} = \{\mathrm{AC}, \mathrm{CA}, \mathrm{ABC}\}$

$\mathsf{S}_{\mathrm{A}} = \{01, 02, 03, 04/05, 06\mathrm{A}, 14, 15, 16, 17/18/19, 20\mathrm{A}, 04/12/18/27, 17/18/27, 28\mathrm{A}\ \}$

$\mathsf{S}_{\mathrm{AC,A}} = \{02,\ 03,\ 04/05,\ 06\mathrm{A}\}$

$\mathsf{sno}_{\mathrm{AC,A}} = \{(02{,}1),(03{,}2),(04/05{,}3),(06\mathrm{A}{,}4)\}$

$\mathsf{mindur}_{\mathrm{AC,A}} = \{(02{,}5),(03{,}2),(04/05{,}2),(06\mathrm{A}{,}2)\}$

$\mathsf{minend}_{\mathrm{AC,A}} = \{(02{,}105),(03{,}0),(04/05{,}0),(06\mathrm{A}{,}0)\}$

$\mathsf{S}_{\mathrm{CA,A}} = \{15,\ 16,\ 17/18/19,\ 20\mathrm{A}\}$

$\mathsf{sno}_{\mathrm{CA,A}} = \{(20\mathrm{A}{,}5),(17/18/19{,}6),(16{,}7),(15{,}8)\}$

$\mathsf{mindur}_{\mathrm{CA,A}} = \{(20\mathrm{A}{,}2),(17/18/19{,}2),(16{,}2),(15{,}5)\}$

$\mathsf{minend}_{\mathrm{CA,A}} = \{(20\mathrm{A},0),(17/18/19,0),(16,0),(15,0)\}$

$\mathsf{S}_{\mathrm{ABC,A}} = \{02,\ 03,\ 04/12/18/27,\ 28\mathrm{A}\}$

$\mathsf{sno}_{\mathrm{ABC,A}} = \{(02,1),\ (03,2),\ (04/12/18/27,3),\ (28\mathrm{A},4)\}$

$\mathsf{mindur}_{\mathrm{ABC,A}} = \{(02,5),\ (03,2),\ (04/12/18/27,2),\ (28\mathrm{A},2)\}$

$\mathsf{minend}_{\mathrm{ABC,A}} = \{(02,120),\ (03,0),\ (04/12/18/27,0),\ (28\mathrm{A},0)\}$

Lemma 8.9
Laut Definition 8.1 (10) [S. 54] nummeriert sno_t alle Abschnitte durchgehend. Dasselbe gilt in jedem Teil $p \in \mathsf{P}$: $\forall t \in \mathsf{T}_p : \forall i \in \mathbb{N}, \min(\mathrm{Bi}(\mathsf{sno}_{t,p})) \leq i \leq \max(\mathrm{Bi}(\mathsf{sno}_{t,p})) : \exists s \in \mathsf{S}_{t,p} : \mathsf{sno}_{t,p}(s) = i$.

Beweis: sno_t nummeriert durchgehend. Es ist also zu zeigen, dass $\mathsf{sno}_{t,p} = \mathsf{sno}_t{\downarrow}\mathsf{S}_{t,p}$ in der Nummerierung keine Lücken aufweist. Das aber folgt direkt aus Lemma 8.6 [S. 58]. □

Definition 8.10 (Grenzabschnitte $\mathsf{Border}_{t,p}$, Übergänge $\mathsf{crossing}_{t,p}$)
Für $t \in \mathsf{T}, p \in \mathsf{P}_t$ sind die *Grenzübergänge* gegeben durch $\mathsf{crossing}_{t,p} := \{(s,s') \in \mathsf{S}_{t,p} \times (\mathsf{S}_t \setminus \mathsf{S}_{t,p}) \mid 1 = |\mathsf{sno}_t(s) - \mathsf{sno}_t(s')|\}$.

Die Menge der *Grenzabschnitte* ist gegeben durch $\mathsf{Border}_{t,p} := \{s \mid \exists s' : (s,s') \in \mathsf{crossing}_{t,p}\}$.

Beispiel 8.11
Diese Relationen sehen für Π^{bsp} folgendermaßen aus:

$\mathsf{crossing}_{\mathrm{AC,A}} = \{(06\mathrm{A}, 06\mathrm{C})\}$, $\mathsf{crossing}_{\mathrm{AC,B}} = \emptyset$, $\mathsf{crossing}_{\mathrm{AC,C}} = \{(06\mathrm{C}, 06\mathrm{A})\}$

$\mathsf{crossing}_{\mathrm{CA,A}} = \{(20\mathrm{A}, 20\mathrm{C})\}$, $\mathsf{crossing}_{\mathrm{CA,B}} = \emptyset$, $\mathsf{crossing}_{\mathrm{CA,C}} = \{(20\mathrm{C}, 20\mathrm{A})\}$

$\mathsf{crossing}_{\mathrm{ABC,A}} = \{(28\mathrm{A}, 28\mathrm{B}/31)\}$,
$\mathsf{crossing}_{\mathrm{ABC,B}} = \{(28\mathrm{B}/31, 28\mathrm{A}), (30\mathrm{B}/31, 30\mathrm{C})\}$,
$\mathsf{crossing}_{\mathrm{ABC,C}} = \{(30\mathrm{C}, 30\mathrm{B}/31)\}$

Lemma 8.12
$\forall t \in \mathsf{T} : \forall p \in \mathsf{P}_t : \forall (s,s') \in \mathsf{crossing}_{t,p} : (s',s) \in \mathsf{crossing}_{t,\mathsf{part}(s')}$.

Beweis: Folgt aus Symmetrie-Gründen direkt aus Definition 8.10 [S. 59]. □

Lemma 8.13
Die Grenzabschnitte haben im jeweiligen Teil als Nummer den jeweils kleinsten oder größten Wert: $\forall t \in \mathsf{T} : \forall p \in \mathsf{P}_t : s \in \mathsf{Border}_{t,p} \Rightarrow \mathsf{sno}_{t,p}(s) \in \{\min(\mathrm{Bi}(\mathsf{sno}_{t,p})), \max(\mathrm{Bi}(\mathsf{sno}_{t,p}))\}$.

Beweis: Lemma 8.9 [S. 59] besagt, dass alle Abschnitte in $\mathsf{S}_{t,p}$ durchgehend nummeriert sind. Außerdem sind Grenzabschnitte $s \in \mathsf{Border}_{t,p}$ nach Definition 8.10 [S. 59] ja gerade so definiert, dass ihre Nummer $\mathsf{sno}_{t,p}(s)$ von der des Nachbarn $s' : (s,s') \in \mathsf{crossing}_{t,p}$ um genau 1 verschieden ist. Deshalb kann ein Grenzabschnitt als Nummer nur entweder das Minimum oder das Maximum von $\mathrm{Bi}(\mathsf{sno}_{t,p})$ haben. □

Korollar 8.14
Jeder Zug hat in jedem Teil höchstens zwei Grenzabschnitte: $\forall t \in \mathsf{T} : \forall p \in \mathsf{P}_t : |\mathsf{Border}_{t,p}| \leq 2$.

9 Lösungen

Hier wird beschrieben, wie die Lösung eines Simulationsproblems aussieht: zuerst Belegungen für Teilprobleme, dann vollständige Belegungen, dann, wodurch vollständige Belegungen inkonsistent sein können, und schließlich nicht-inkonsistente (korrekte) Belegungen oder Lösungen.

Definition 9.1 (Teil-Belegungen Assign_p)
Gegeben Π und hier ein konkreter Teil $p \in \mathsf{P}$. Hierfür sei $\mathsf{Assign}_p := \{((\mathsf{start}_{t,p}, \mathsf{dur}_{t,p})_{t \in \mathsf{T}_p})\}$ die *Menge aller Teil-Belegungen* von Π bezüglich p.

Dabei ist *eine Teil-Belegung* ein Tupel von Paaren $((\mathsf{start}_{t,p}, \mathsf{dur}_{t,p})_{t \in \mathsf{T}_p})$ mit

(1) $\mathsf{start}_{t,p} : \mathsf{S}_{t,p} \to \mathbb{N}$
Für jeden Zug und jeden Belegungsabschnitt der Belegungsbeginn.

(2) $\mathsf{dur}_{t,p} : \mathsf{S}_{t,p} \to \mathbb{N}$
Für jeden Zug und jeden Belegungsabschnitt die Belegungsdauer.

(3) $\forall t \in t : \forall s \in \mathsf{S}_{t,p} : \mathsf{dur}_{t,p}(s) \geq \mathsf{mindur}_{t,p}(s)$
Die tatsächliche Belegungsdauer darf nicht kleiner sein als die minimale Belegungsdauer.

(4) $\forall t \in t : \forall s \in \mathsf{S}_{t,p} : \mathsf{start}_{t,p}(s) + \mathsf{dur}_{t,p}(s) \geq \mathsf{minend}_{t,p}(s)$
Fahrplanbedingung: Kein Zug darf irgendeinen Belegungsabschnitt vor der fahrplanmäßigen Abfahrtzeit *verlassen*.

(5) $\forall t \in \mathsf{T}_p : \forall s, s' \in \mathsf{S}_{t,p} :$
$\mathsf{sno}_{t,p}(s) + 1 = \mathsf{sno}_{t,p}(s') \Rightarrow \mathsf{start}_{t,p}(s) + \mathsf{dur}_{t,p}(s) - 1 = \mathsf{start}_{t,p}(s')$
Lokale Konsistenz: Für jeden Zug gilt, für zwei aufsteigend aufeinander folgende Belegungsabschnitte (Vorgänger, Nachfolger) beginnt die Belegung des Nachfolgers mit dem Belegungsende des Vorgängers, minus einer konstanten Überlappung (siehe Abschnitt 6.3, S. 48) von einer Zeiteinheit.

(6) $\forall (s, s') \in \mathsf{excl} : \forall t, t' \in \mathsf{T}_p : t \neq t' \wedge s \in \mathsf{S}_{t,p} \wedge s' \in \mathsf{S}_{t',p} \Rightarrow$
$\mathsf{start}_{t,p}(s) + \mathsf{dur}_{t,p}(s) \leq \mathsf{start}_{t',p}(s) \vee \mathsf{start}_{t',p} + \mathsf{dur}_{t',p} \leq \mathsf{start}_{t,p}(s)$
Die Belegungszeiten (vom Start $\mathsf{start}_{t,p}(s)$ bis zum Ende $\mathsf{start}_{t,p}(s) + \mathsf{dur}_{t,p}(s)$) beliebiger Züge auf sich ausschließenden Belegungsabschnitten überlappen sich nicht.
Diese Bedingung heißt auch *Blockausschlussbedingung* und realisiert die Grundregel der Blocksicherung (Abschnitt 5.1, S. 42): Zu keinem Zeitpunkt ist irgendein Blockabschnitt von mehr als einem Zug belegt.

Definition 9.2 (Belegungen Assign)
Eine *(vollständige, globale) Belegung* setzt sich zusammen aus Teil-Belegungen für alle Teile $p \in \mathsf{P}$. Damit ist eine vollständige Belegung eine Familie von Teil-Belegungen: $(((\mathsf{start}_{t,p}, \mathsf{dur}_{t,p})_{t \in \mathsf{T}_p})_{p \in \mathsf{P}})$.

Weiterhin sei $\mathsf{Assign} := \{(((\mathsf{start}_{t,p}, \mathsf{dur}_{t,p})_{t \in \mathsf{T}_p})_{p \in \mathsf{P}})\}$ die *Menge aller (vollständigen) Belegungen* für das gegebene Π.

Definition 9.3 ($\mathsf{start}_t, \mathsf{dur}_t$)
Für ein gegebenes $(((\mathsf{start}_{t,p}, \mathsf{dur}_{t,p})_{t \in \mathsf{T}_p})_{p \in \mathsf{P}}) \in \mathsf{Assign}$ seien folgende abkürzende Schreibweisen definiert:

(1) $\mathsf{start}_t : \mathsf{S}_t \to \mathbb{N}$, $\mathsf{start}_t(s) := \mathsf{start}_{t,\mathsf{part}(s)}(s)$

(2) $\mathsf{dur}_t : \mathsf{S}_t \to \mathbb{N}$, $\mathsf{dur}_t(s) := \mathsf{dur}_{t,\mathsf{part}(s)}(s)$

Korollar 9.4
$\mathsf{Assign} = \{((\mathsf{start}_t, \mathsf{dur}_t)_{t\in\mathsf{T}})\}$.

Beispiel 9.5
Ich gebe wieder für das synthetische Beispiel Π^{bsp} *eine* Belegung an: siehe Abbildung 9.1 [S. 62].

$\forall p \in P : \forall t \in \mathsf{T}_p : \mathsf{dur}_{t,p} = \mathsf{mindur}_{t,p}$

$\mathsf{start}_{\mathrm{AC,A}} = \{(02{,}100),(03{,}104),(04/05{,}105),(06\mathrm{A},106)\}$

$\mathsf{start}_{\mathrm{AC,C}} = \{(06\mathrm{C},106),(07/08{,}107),(09{,}108),(10{,}109)\}$

$\mathsf{start}_{\mathrm{CA,C}} = \{(25{,}100),(24{,}104),(21/22/23{,}105),(20\mathrm{C},106)\}$

$\mathsf{start}_{\mathrm{CA,A}} = \{(20\mathrm{A},106),(17/18/19{,}107),(16{,}108),(15{,}109)\}$

$\mathsf{start}_{\mathrm{ABC,A}} = \{(02{,}115),(03{,}119),(04/12/18/27{,}120),(28\mathrm{A},121)\}$

$\mathsf{start}_{\mathrm{ABC,B}} = \{(28\mathrm{B}/31{,}121),(32{,}122),(30\mathrm{B}/31{,}126)\}$

$\mathsf{start}_{\mathrm{ABC,C}} = \{(30\mathrm{C},126),(08/13/22/29{,}127),(09{,}128),(10{,}129)\}$

Bemerkung 9.6
An diesem Beispiel kann man auch schön sehen, dass die Assign_p und damit Assign nicht endlich sind: u.a. können die Belegungszeiten der Züge beliebig in die Zukunft verschoben werden.

Definition 9.7 (Inkonsistente Teile Incons)
Die Menge der *inkonsistenten Teile* einer Lösung ist gegeben durch Incons:

(1) $\mathsf{Incons} : \mathsf{Assign} \to 2^{\mathsf{P}}$

(2) $\mathsf{Incons}((((\mathsf{start}_{t,p}, \mathsf{dur}_{t,p})_{t\in\mathsf{T}_p})_{p\in\mathsf{P}})) := \{p \in \mathsf{P} \mid \exists t \in \mathsf{T}_p, (s,s') \in \mathsf{crossing}_{t,p} : \mathsf{start}_{t,p}(s) \neq \mathsf{start}_{t,\mathsf{part}(s')}(s') \vee \mathsf{dur}_{t,p}(s) \neq \mathsf{dur}_{t,\mathsf{part}(s')}(s')\}$

In einer Belegung sind also diejenigen Teile inkonsistent, deren Belegungszeiten ($\mathsf{start}_{t,p}$ oder $\mathsf{dur}_{t,p}$) an den Grenzen (s) nicht identisch sind mit denen der zugehörigen Nachbarabschnitte ($s', (s,s') \in \mathsf{crossing}_{t,p}$).

Definition 9.8 (Konsistente Belegung, Lösung)
Eine *Belegung* $a \in \mathsf{Assign}$ heißt *konsistent* wenn $\mathsf{Incons}(a) = \emptyset$. Eine solche Belegung ist eine *Lösung*.

Korollar 9.9
Für jede Lösung $(((\mathsf{start}_{t,p}, \mathsf{dur}_{t,p})_{t\in\mathsf{T}_p})_{p\in\mathsf{P}}) \in \mathsf{Assign}$ gilt:

(1) $\forall p \in \mathsf{P} : \forall t \in \mathsf{T}_p : \forall (s,s') \in \mathsf{crossing}_{t,p} : \mathsf{start}_t(s) = \mathsf{start}_t(s') \wedge \mathsf{dur}_t(s) = \mathsf{dur}_t(s')$

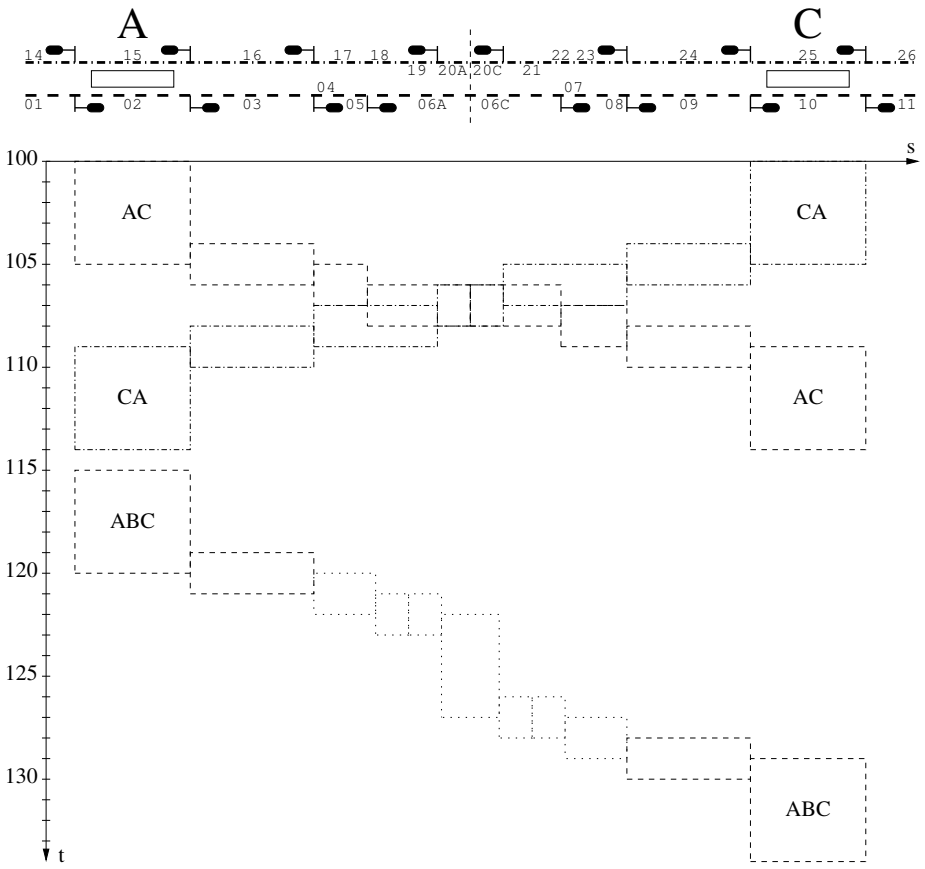


Abbildung 9.1: *Eine* vollständige Belegung für das synthetische Beispiel Π^{bsp}. Die Rechtecke beschreiben Belegungszeiten für die unterschiedlichen Belegungsabschnitte (die jeweils obere waagrechte Linie steht für den Belegungsbeginn start, die Dauer dur wird durch die Höhe des Rechtecks symbolisiert). Die Abschnitte, die sich auf das obere Gleis beziehen (von C nach A), sind wie das Gleis mit einer Strich-Punkt-Linie umschrieben, die unteren (von A nach C) mit einer gestrichelten. Die gepunkteten Blöcke beziehen sich auf Abschnitte, die der Übersichtlichkeit halber hier nicht dargestellt sind: 12,27,28,31,32,30,29,13 (siehe Abbildung 8.1, S. 56). Zeitlich zusammenhängende Blöcke beschreiben den Lauf jeweils eines Zugs (wie in Abbildung 6.2, S. 48): AC, CA und ABC.

Definition 9.10 (Menge aller Lösungen Solution)
Für ein gegebenes Π ist die Menge aller Lösungen:

(1) $\mathsf{Solution} := \{a \in \mathsf{Assign} | \mathsf{Incons}(a) = \emptyset\}$

Genau wie Assign ist auch Solution nicht endlich.

Beispiel 9.11 (Lösung, inkonsistente Belegung)
Die Belegung aus Beispiel 9.5 [S. 61] ist konsistent, also eine Lösung.

Die Belegung a in Abbildung 9.2 [S. 64] hingegen ist nicht konsistent, es gilt: $\mathsf{Incons}(a) = \{\mathrm{B}, \mathrm{C}\}$.

Beispiel 9.12 (Blockausschluss)
Abbildung 9.3 [S. 65] zeigt exemplarisch, warum einige Belegungen zu Π^{bsp} wegen der Blockausschlussbedingung (Definition 9.1 (6), S. 60) nicht möglich sind.

Bemerkung 9.13
In einer (konsistenten) Lösung werden zusammenhängende innere Abschnitte anders behandelt als die an den Grenzen. Erstere folgen quasi zeitlich aufeinander ($\mathsf{start}_{t,p}(s) + \mathsf{dur}_{t,p}(s) - 1 = \mathsf{start}_{t,p}(s')$), während letztere zeitgleich belegt werden ($\mathsf{start}_t(s) = \mathsf{start}_t(s') \wedge \mathsf{dur}_t(s) = \mathsf{dur}_t(s')$). Das suggeriert, dass ein Problem durch unterschiedliche Zerlegungen unterschiedliche Lösungen hat. In der Anwendung aber werden die Belegungsabschnitte in Abhängigkeit der Zerlegung so definiert, dass die Lösungen unterschiedlicher Zerlegungen einander entsprechen!

In den Beispielen haben wir auch schon gesehen, wie das geht: *Reale* Belegungsabschnitte werden durch die Problem-Zerlegung so geteilt, dass sie in einer Lösung durch die Unifikation der Werte $\mathsf{start}_{t,p}$ und $\mathsf{dur}_{t,p}$ wieder zusammengefügt sind.

Definition 9.14 (Leere Belegung assign^0)
Für die folgenden Definitionen benötige ich noch die *leere Belegung*, die nach Definition 9.2 [S. 60] keine Belegung ist. Sie hat zwar die Struktur einer Belegung, erfüllt aber die weiteren Bedingungen nicht:

(1) $\mathsf{assign}^0 := ((\mathsf{start}_{t,p} : \mathsf{S}_{t,p} \to \{0\}, \mathsf{dur}_{t,p} : \mathsf{S}_{t,p} \to \{0\})_{t \in \mathsf{T}_p})_{p \in \mathsf{P}}$

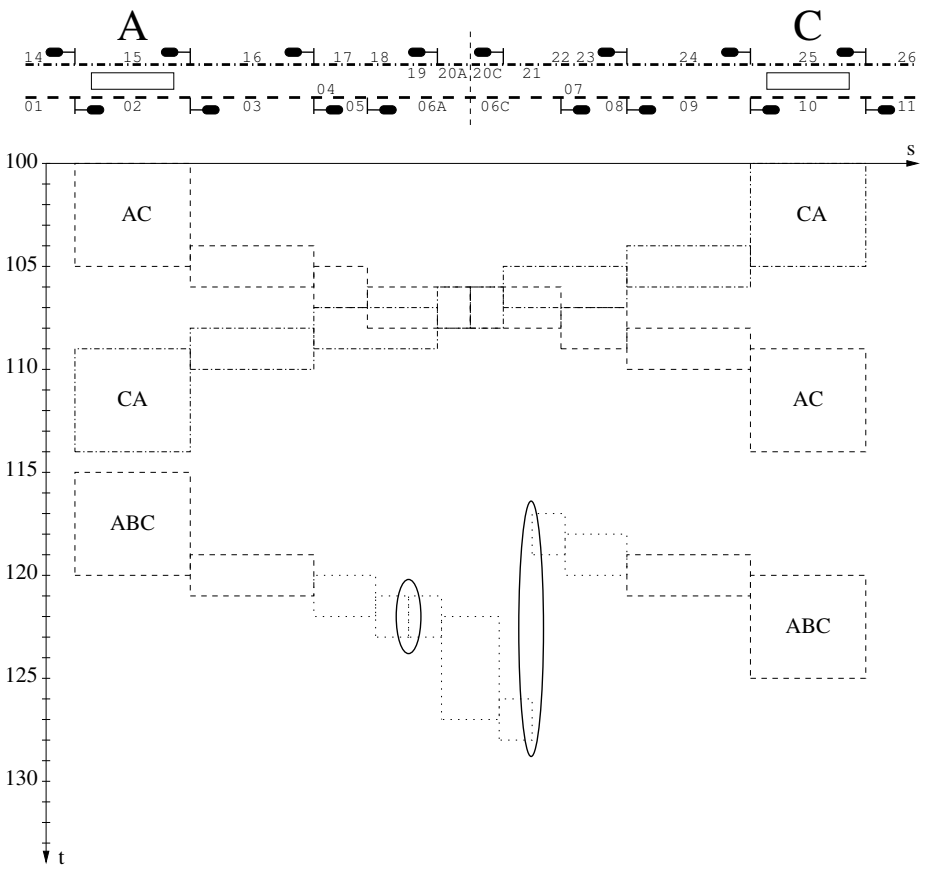


Abbildung 9.2: Beispiel für eine inkonsistente Belegung bezüglich Zug ABC zwischen den Teilen B und C. Die gepunktet umrandeten Rechtecke beziehen sich wie in Abbildung 9.1 [S. 62] auf in der Infrastruktur oben nicht dargestellte Abschnitte. Das rechte Oval kennzeichnet die inkonsistenten Zeiten zwischen B und C, im linken Oval kann man sehen, dass die Zeiten zwischen A und B konsistent sind.
Dieses Beispiel zeigt übrigens nicht zuletzt deshalb eine Belegung, weil die Fahrplanbedingungen gelten: ABC verlässt sowohl den Abschnitt 02 als auch 32 nicht früher als durch minend festgelegt: $\mathsf{minend}_{\mathrm{ABC}}(02) = 120, \mathsf{minend}_{\mathrm{ABC}}(32) = 127$. Im Teil bzw. Bahnhof C gibt $\mathsf{minend}_{\mathrm{ABC}}$ keine Abfahrtzeit vor (bzw. überall 0), sodass für eine gültige Belegung die Sperrzeitentreppe dort auch wie gezeigt liegen kann. Das ist also eine (gültige) Belegung, aber eben keine konsistente.

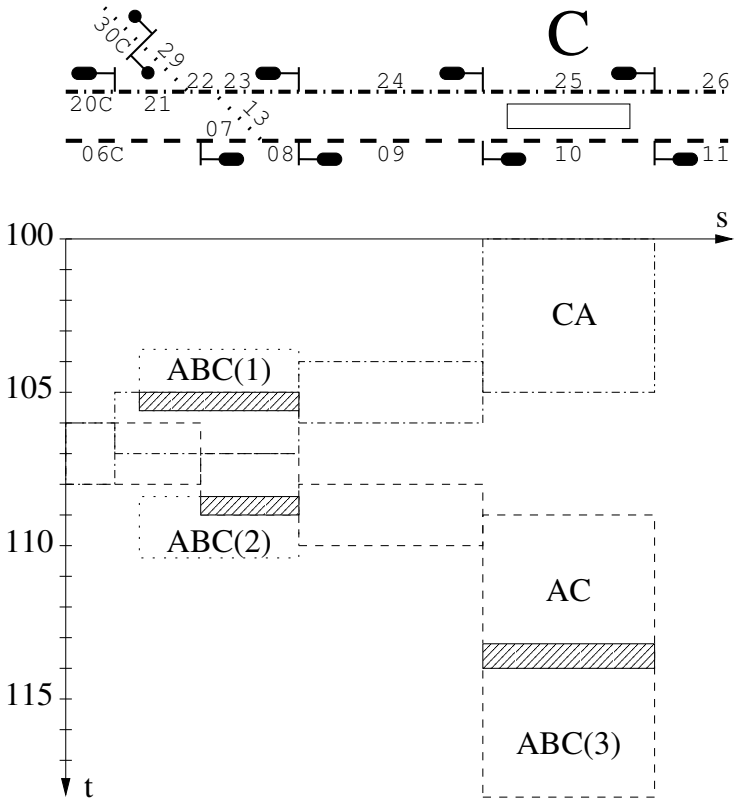


Abbildung 9.3: Belegungen zu Π^{bsp}, die die Blockausschluss-Bedingung nicht erfüllen und deshalb ungültig sind. Hier sind wieder die Zeiten der Belegungsabschnitte des unteren Gleises gestrichelt umrandet, die des oberen per Strich-Punkt-Linie und die der quer dazu laufenden Gleise per Punkt-Linie. Für den Zug ABC sind drei Blöcke herausgegriffen (die beiden linken stellen Alternativen für denselben Belegungsabschnitt dar), die sich jeweils mit einem Block eines anderen Zuges überlappen. Diese Überlappung ist nach Definition 9.1 (6) [S. 60] verboten. Die Überlappungen sind definiert durch die Relation excl und hier schraffiert hervorgehoben. Rechts unten überlappt der Block ABC(3) mit dem letzten des Zuges ABC auf demselben Belegungsabschnitt 10 ($(10, 10) \in$ excl). Links unten überlappen ABC(2) auf 08/13/22/29 und der Block 07/08 des Zuges AC ($(08/13/22/29, 07/08) \in$ excl). Und links oben überlappt ABC(1) auf 08/13/22/29 mit CA auf 21/22/23 ($(08/13/22/29, 21/22/23) \in$ excl).

10 Lokale Simulation

In diesem Kapitel wird gezeigt, wie eine lokale Simulation Σ_p eine Teilbelegung berechnet und warum die Ergebnisse einer lokalen Simulation immer lokal korrekt sind. Außerdem sehen wir, warum lokale Simulationen in diesem Zusammenhang besonders einfach unter Verwendung von Constraint-Propagation umgesetzt werden können.

Definition 10.1 (Lokale Simulation Σ_p)
Eine lokale Simulation überführt eine vollständige, globale Belegung in eine neue *lokale* Belegung indem sie an den Belegungszeiten *eines Teils* Änderungen vornimmt.

Gegeben ein Simulationsproblem Π und ein $p \in \mathsf{P}$. Eine lokale Simulation Σ_p ist gegeben durch

(1) $\Sigma_p : \mathsf{Assign} \cup \{\mathsf{assign}^0\} \rightarrow \mathsf{Assign}_p$

(2) $\Sigma_p(a) = a'_p$
$a = ((\mathsf{start}_{t,q}, \mathsf{dur}_{t,q})_{t \in \mathsf{T}_q})_{q \in \mathsf{P}}$
$a'_p = (\mathsf{start}'_{t,p}, \mathsf{dur}'_{t,p})_{t \in \mathsf{T}_p}$

(3) $\forall t \in t : \forall s \in \mathsf{S}_{t,p} : \mathsf{dur}'_{t,p}(s) \geq \mathsf{mindur}_{t,p}(s)$
Die tatsächliche Belegungsdauer in jedem Belegungsabschnitt ist größer gleich der Mindest-Belegungsdauer.

(4) $\forall t \in t : \forall s \in \mathsf{S}_{t,p} : \mathsf{start}'_{t,p}(s) + \mathsf{dur}'_{t,p}(s) \geq \mathsf{minend}_{t,p}(s)$
Die Belegung endet nicht vor dem Fahrplan-Eintrag, d.h. der Zug fährt nicht früher als Fahrplan-gemäß ab.

(5) $\forall t \in \mathsf{T}_p : \forall s, s' \in \mathsf{S}_{t,p} : \mathsf{sno}_{t,p}(s)+1 = \mathsf{sno}_{t,p}(s') \Rightarrow \mathsf{start}'_{t,p}(s)+\mathsf{dur}'_{t,p}(s)-1 = \mathsf{start}'_{t,p}(s')$
Für aufeinander folgende Abschnitte gilt: der Belegungsbeginn des nachfolgenden Abschnitts ist gleich dem Belegungsende des vorhergehenden Abschnitts minus einer konstanten Überlappung von einer Zeiteinheit.

(6) $\forall (s, s') \in \mathsf{excl} : \forall t, t' \in \mathsf{T}_p : \mathsf{tno}(t') < \mathsf{tno}(t) \wedge s \in \mathsf{S}_{t,p} \wedge s' \in \mathsf{S}_{t',p} \Rightarrow \mathsf{start}'_{t,p}(s) + \mathsf{dur}'_{t,p}(s) \leq \mathsf{start}'_{t',p} \vee \mathsf{start}'_{t',p} + \mathsf{dur}'_{t',p} \leq \mathsf{start}'_{t,p}(s)$
Lokal, im Teil p, gilt der Blockausschluss (Definition 9.1 (6), S. 60).

(7) $\forall t \in \mathsf{T}_p : \forall (s, s') \in \mathsf{crossing}_{t,p} : \mathsf{start}'_{t,p}(s) \geq \mathsf{start}_{t,p}(s) \wedge \mathsf{dur}'_{t,p}(s) \geq \mathsf{dur}_{t,p}(s)$
Lokale Geschichte: An den Grenzen müssen Belegungs-Start und -Dauer größer gleich Start bzw. Dauer der bisherigen Belegung sein.

(8) $\forall t \in \mathsf{T}_p : \forall (s, s') \in \mathsf{crossing}_{t,p} : \mathsf{start}'_{t,p}(s) \geq \mathsf{start}_{t,\mathsf{part}(s')}(s') \wedge \mathsf{dur}'_{t,p}(s) \geq \mathsf{dur}_{t,\mathsf{part}(s')}(s')$
Für benachbarte Teile gilt: an den Grenzen ist im Ergebnis der lokalen Simulation der Belegungs-Start größer gleich der Startzeit des entsprechenden Nachbarn. Genauso ist die Dauer größer gleich der Dauer des entsprechenden Nachbarn.

(9) Den konkreten lokalen Belegungs-Algorithmus:

for $i := 1$ **to** $|\mathsf{T}|$ **do**
  $t := \mathsf{tno}^{-1}(i)$
  $s := \mathsf{last}_t(p)$
  setze $e := \mathsf{start}'_{t,p}(s) + \mathsf{dur}'_{t,p}(s)$ minimal unter geg. Bedingungen
  for $j := \max(\mathrm{Bi}(\mathsf{sno}_{t,p}))$ **to** $\min(\mathrm{Bi}(\mathsf{sno}_{t,p}))$ **do**
    $s := \mathsf{sno}_{t,p}{}^{-1}(j)$
    setze $\mathsf{dur}'_{t,p}(s)$ minimal unter gegebenen Bedingungen
  end
end

Bemerkung 10.2
Die Definitions-Teile (2) bis (5) entsprechen Definition 9.1 [S. 60]. 10.1 (6) hingegen ist anders definiert als 9.1 (6). Die Definition hier spiegelt den Algorithmus wieder, mit dem konkret die Werte von $\mathsf{start}'_{t,p}(s)$ und $\mathsf{dur}'_{t,p}(s)$ berechnet werden (9): Beginnend mit dem *kleinsten Zug* t werden für alle Züge nacheinander die konkreten Werte für $\mathsf{start}'_{t,p}(s)$ und $\mathsf{dur}'_{t,p}(s)$ bestimmt und zwar so, dass sie jeweils lokal konsistent sind und überlappungsfrei zu allen bereits berechneten Zügen.

(7) ist notwendig für Terminierung: indem lokal jeder Zug immer nur hinaus geschoben wird, wird global nicht immer wieder dieselbe Belegung versucht. (8) synchronisiert Start und Ende jedes Zuges in einem Abschnitt mit den Nachbarn: sie dürfen jeweils nicht früher beginnen als im Nachbarabschnitt.

(9) schließlich gibt den genauen lokalen Belegungsalgorithmus an, der sich auf die anderen Bedingungen bezieht: Beginnend beim kleinsten Zug (bzgl. tno) aufsteigend zum größten Zug wird für jeden Zug zunächst die Ausfahrzeit $\mathsf{start}'_{t,p}(s) + \mathsf{dur}'_{t,p}(s)$ bezüglich aller Bedingungen minimal gesetzt. Dann wird beginnend beim lokal letzten Abschnitt (s.a. Definition 8.4, S. 57) absteigend zum ersten Abschnitt nur noch die Dauer minimal gesetzt. Die Startzeit ergibt sich dann jeweils aus den anderen Bedingungen (oder *Constraints*).

Unter gegebenen Bedingungen bedeutet jeweils, unter *allen bis hierher geltenden Bedingungen*! Das heißt zum Beispiel für $i = 2$ und den entsprechenden Zug $t = \mathsf{tno}^{-1}(2)$ dass die Belegungszeiten des Zuges $t' = \mathsf{tno}^{-1}(1)$ (der in der Schleife ja vor t behandelt wurde) berücksichtigt werden müssen, vor allem für die Gleisausschlussbedingungen.

Bemerkung 10.3 (CP)
Offenbar kann man das angegebene lokale Simulationsproblem unmittelbar als CSP formulieren (alle Bedingungen werden durch entsprechende Constraints ausgedrückt) und mit dem Belegungsalgorithmus lösen. Und genau so wird es auch in der Implementierung DRS gemacht: siehe Teil III [S. 97].

Beim Belegungsalgorithmus Definition 10.1 (9) [S. 67] sind keine Suchschritte angegeben. Theoretisch kann man hier vollständige Propagation (oder starke k-Konsistenz, siehe Abschnitt 3.4.5, S. 26) annehmen, mit der ohne Suche immer eine Lösung gefunden wird. Wenn diese (wie in der Praxis meistens) nicht gegeben ist, kann man zum Beispiel durch ein einfaches Suchverfahren (kombiniert mit der Propagation des Constraint-Solvers) per Backtracking eine entsprechende Lösung erhalten.

Lemma 10.4 (Σ_p ist wohldefiniert)
Zu zeigen ist, dass $\Sigma_p(a) \in \mathsf{Assign}_p$ für alle $a \in \mathsf{Assign} \cup \{\mathsf{assign}^0\}$.

Beweis: Ich zeige zunächst, dass Σ_p immer eine Lösung findet: Dazu genügt zu zeigen, dass die Bedingungen (3) bis (8) widerspruchsfrei sind, weil dann der Algorithmus (9) für jede Kombination dieser Bedingungen eine Lösung findet. Das aber gilt: (3) begrenzt die Dauern von unten, (4) die Startzeiten, (5) verbindet einige davon widerspruchsfrei, (6) schließt verschiedene Belegungsüberlappungen aus und erfordert damit, dass eine Belegung vollständig vor oder nach der anderen kommt, (7) und (8) begrenzen wiederum einige Startzeiten und Dauern von unten. Intuitiv kann man sagen, dass keine der Bedingungen einen Startwert absolut von oben begrenzt und damit der Zeithorizont nach oben hin offen ist, also im Prinzip jeder Zug beliebig verzögert werden kann und deshalb immer eine Lösung existiert.

Außerdem ist zu zeigen, dass $\Sigma_p(a)$ alle Bedingungen erfüllt, um Element von Assign_p zu sein. Wie wir bereits gesehen haben, entsprechen 10.1 (1) bis (5) denen in Definition 9.1 [S. 60], (7) bis (9) sind zusätzliche Bedingungen. Zu zeigen ist also nur noch, dass 10.1 (6) äquivalent ist mit 9.1 (6):

$$\forall (s,s') \in \mathsf{excl} : \forall t,t' \in \mathsf{T}_p : \mathsf{tno}(t') < \mathsf{tno}(t) \wedge s \in \mathsf{S}_{t,p} \wedge s' \in \mathsf{S}_{t',p} \Rightarrow$$
$$\mathsf{start}'_{t,p}(s) + \mathsf{dur}'_{t,p}(s) \leq \mathsf{start}'_{t',p} \vee \mathsf{start}'_{t',p} + \mathsf{dur}'_{t',p} \leq \mathsf{start}'_{t,p}(s)$$
$$\text{gdw.(Symmetrie: } t \text{ und } t' \text{ austauschbar)}$$
$$\forall (s,s') \in \mathsf{excl} : \forall t,t' \in \mathsf{T}_p : t \neq t' \wedge s \in \mathsf{S}_{t,p} \wedge s' \in \mathsf{S}_{t',p} \Rightarrow$$
$$\mathsf{start}_{t,p}(s) + \mathsf{dur}_{t,p}(s) \leq \mathsf{start}_{t',p}(s) \vee \mathsf{start}_{t',p} + \mathsf{dur}_{t',p} \leq \mathsf{start}_{t,p}(s)$$

□

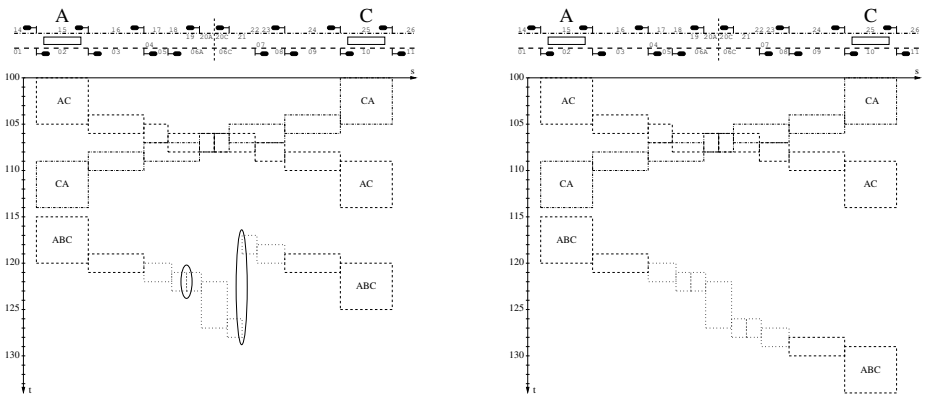
Bemerkung 10.5
In Definition 10.1 [S. 66] wird die Beziehung zwischen einer alten Belegung a und einer neuen a' in den Punkten (7) und (8) definiert. Beide benutzen die Abbildungen aus a ausschließlich als *untere Schranken* für die entsprechenden in a'. Deshalb ist Σ_p wohldefiniert auch für $a = \mathsf{assign}^0$, bei der ja alle Abbildungen konstant nach 0 gehen.

Korollar 10.6 (Determinismus)
Σ_p ist deterministisch. Wenn von außen untere Schranken für start und dur vorgegeben sind, wird ein eindeutig bestimmtes Ergebnis berechnet, das die Überlappungs-, Zusammenhangs- und Fahrplan-Bedingungen erfüllt. Das Ergebnis ist für jeden Zug (in der Reihenfolge seiner Nummer) bezüglich der Ausfahrzeit und der Durchfahrzeit minimal.

Korollar 10.7 (Korrektheit von Σ_p)
Der gegebene Algorithmus berechnet für jeden Zug *eine mögliche Fahrt* (s.a. Kapitel 4, S. 38) durch den gegebenen Teil.

Der angegebene Algorithmus berechnet auf der durch die $\mathsf{start}_{t,p}$, $\mathsf{dur}_{t,p}$, usw. gegebenen *Datenstruktur* eine einfache Version einer Simulation. Die tatsächliche lokale Simulation, die beliebig genau sein kann, ist nicht Gegenstand dieser Arbeit. Der im Projekt SIMONE entwickelte lokale Simulator (siehe Kapitel 22, S. 146) basiert auf einer Datenstruktur die unserer sehr ähnlich ist, berechnet aber vor allem die Belegungszeiten genauer und macht sie nicht einfach abhängig von minimalen Belegungszeiten, sondern auch von der Fahrgeschwindigkeit des Zuges und ähnlichem. Er berücksichtigt aber insbesondere wie hier gefordert die globale Reihenfolge der Züge.

Wir abstrahieren hier also von Details, die zwar für die lokale Simulation wichtig sind, damit dieser möglichst wirklichkeitsgetreue Ergebnisse liefert, für den globalen verteilten Algorithmus aber unwichtig. Und nur um letzteren geht es hier ja.



(1) (2)

Abbildung 10.1: Anwendung von Σ_p: (1) zeigt die inkonsistente Belegung aus Beispiel 9.11 [S. 63], (2) das Ergebnis aus Beispiel 9.5 [S. 61].

Beispiel 10.8
Ich zeige exemplarisch die Anwendung von Σ_p auf eine inkonsistente Belegung: $\Sigma^{\mathrm{C}}(a) = a'_{\mathrm{C}}$. Siehe auch Abbildung 10.1 [S. 69].
$a = ((\mathsf{start}_{t,q}, \mathsf{dur}_{t,q})_{t \in \mathsf{T}_q})_{q \in \mathsf{P}}$:

$\forall p \in P : \forall t \in \mathsf{T}_p : \mathsf{dur}_{t,p} = \mathsf{mindur}_{t,p}$

$\mathsf{start}_{\mathrm{AC,A}} = \{(02{,}100),(03{,}104),(04/05{,}105),(06\mathrm{A}{,}106)\}$

$\mathsf{start}_{\mathrm{AC,C}} = \{(06\mathrm{C}{,}106),(07/08{,}107),(09{,}108),(10{,}109)\}$

$\mathsf{start}_{\mathrm{CA,C}} = \{(25{,}100),(24{,}104),(21/22/23{,}105),(20\mathrm{C}{,}106)\}$

$\mathsf{start}_{\mathrm{CA,A}} = \{(20\mathrm{A}{,}106),(17/18/19{,}107),(16{,}108),(15{,}109)\}$

$\mathsf{start}_{\mathrm{ABC,A}} = \{(02{,}115),(03{,}119),(04/12/18/27{,}120),(28\mathrm{A}{,}121)\}$

$\mathsf{start}_{\mathrm{ABC,B}} = \{(28\mathrm{B}/31{,}121),(32{,}122),(30\mathrm{B}/31{,}126)\}$

$\mathsf{start}_{\mathrm{ABC,C}} = \{(30\mathrm{C}{,}117),(08/13/22/29{,}118),(09{,}119),(10{,}120)\}$

$a'_{\mathrm{C}} = (\mathsf{start}'_{t,\mathrm{C}}, \mathsf{dur}'_{t,\mathrm{C}})_{t \in \mathsf{T}_{\mathrm{C}}}$:

$\forall t \in \mathsf{T}_{\mathrm{C}} : \mathsf{dur}'_{t,\mathrm{C}} = \mathsf{mindur}_{t,\mathrm{C}}$

$\mathsf{start}'_{\mathrm{AC,C}} = \{(06\mathrm{C}{,}106),(07/08{,}107),(09{,}108),(10{,}109)\}$

$\mathsf{start}'_{\mathrm{CA,C}} = \{(25{,}100),(24{,}104),(21/22/23{,}105),(20\mathrm{C}{,}106)\}$

$\mathsf{start}'_{\mathrm{ABC,C}} = \{(30\mathrm{C}{,}126),(08/13/22/29{,}127),(09{,}128),(10{,}129)\}$

Wie oben festgestellt, sind in a die Teile B und C inkonsistent, dort insbesondere der Zug ABC: $(30\mathrm{C}, 30\mathrm{B}/31) \in \mathsf{crossing}_{\mathrm{ABC,C}}$ (s. Beispiel 8.11, S. 59), aber $\mathsf{start}_{\mathrm{ABC,C}}(30\mathrm{C}) < \mathsf{start}_{\mathrm{ABC,B}}(30\mathrm{B}/31)$.

In der Anwendung von Σ_p auf a führt genau diese Ungleichheit (wegen Definition 10.1 (8), S. 66) dazu, dass der Zug ABC in die Zukunft verschoben wird. Der Belegungsalgorithmus (Definition 10.1 (9), S. 67) verschiebt ihn dann genau so weit, dass er konsistent zu B wird.

Definition 10.9 (Einbettung embed)
Eine Einbettung embed erzeugt aus einer gegebenen vollständigen Belegung und Ergebnissen von Σ_p für einige Teile $Q \subseteq \mathsf{P}$ eine neue vollständige Belegung:

(1) $\mathsf{embed} : 2^{\mathsf{P}} \times (\mathsf{Assign} \cup \{\mathsf{assign}^0\}) \to \mathsf{Assign}$

(2) $\mathsf{embed}(Q, a) := (\Sigma_q(a))_{q \in Q} \circ (a_q)_{\mathsf{P} \ni q \notin Q}$
Die neue Belegung setzt sich zusammen aus allen neu berechneten Teil-Belegungen bzgl. Q und den nicht neu berechneten aus a.

Definition 10.10 (Initiale Belegung assign^1)
Mit embed kann man auch die Anwendung von Σ_p auf *alle Teile* P darstellen. Angewandt auf die leere Belegung assign^0 sei dies die *initiale Belegung* assign^1:

(1) $\mathsf{assign}^1 := \mathsf{embed}(\mathsf{P}, \mathsf{assign}^0) \in \mathsf{Assign}$

11 Verteilte Simulation

Dieses Kapitel beschreibt den verteilten Teil Σ des gesamten Algorithmus DRS: Σ veranlasst zu Beginn alle lokalen Simulationen, initiale Belegungen zu berechnen. Die Ergebnisse der lokalen Simulationen werden dann an den Schnittstellen der benachbarten Teile verglichen. Solange dort Inkonsistenzen bestehen, werden die lokalen Simulationen mit den Nachbardaten als zusätzliche neue Constraints iteriert. Die wiederholte Ausführung der lokalen Berechnungen muss *fair* ablaufen, damit sichergestellt ist, dass jede zur Berechnung anstehende lokale Simulation nach endlich vielen Schritten auch wirklich berechnet wird.

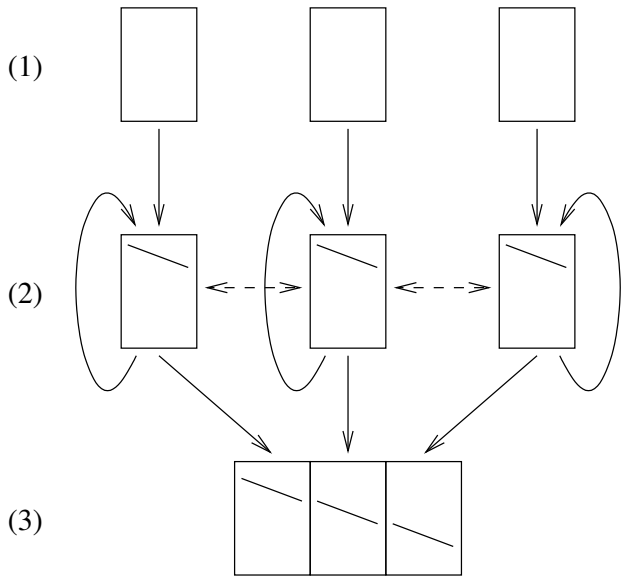


Abbildung 11.1: Skizze des verteilten Algorithmus. Durchgezogene Pfeile stehen für Anwendungen von Σ_p, gestrichelte für Kommunikation zwischen Teilen. Die Rechtecke symbolisieren Simulationsteile, die schrägen Linien darin Sperrzeitentreppen (siehe Abbildung 6.2, S. 48).

Abbildung 11.1 [S. 71] gibt eine Skizze des verteilten Algorithmus: Ausgehend von einer leeren Belegung (1) werden für alle Teile per Σ_p korrekte Teil-Belegungen berechnet. Von den Teil-Belegungen werden die Belegungszeiten der Züge an den Grenzen (crossing) an die Nachbarn kommuniziert (2). Sofern die Belegungszeiten inkonsistent sind, werden die inkonsistenten Teile neu berechnet (2). Dieser Prozess wird so lange wiederholt, bis keine Teile mehr inkonsistent sind und (Definition 9.8, S. 61) eine konsistente Lösung vorliegt (3).

Abbildung 11.2 [S. 72] zeigt skizzenhaft die Anwendung auf Π^{bsp}: Im Schritt (1) werden per Σ_p Teil-Belegungen berechnet, die zusammengenommen sehr ähnlich der in Beispiel 9.11 [S. 63] gegebenen Belegung sind. Tatsächlich ist nur $\mathsf{start}_{\text{ABC,C}}$ etwas verschieden davon: der Zug liegt tatsächlich früher, nämlich zum für ihn frühest möglichen Zeitpunkt, also nur durch die Konflikte mit den anderen Zügen bestimmt. In (2) kommunizieren alle Teile die Belegungszeiten

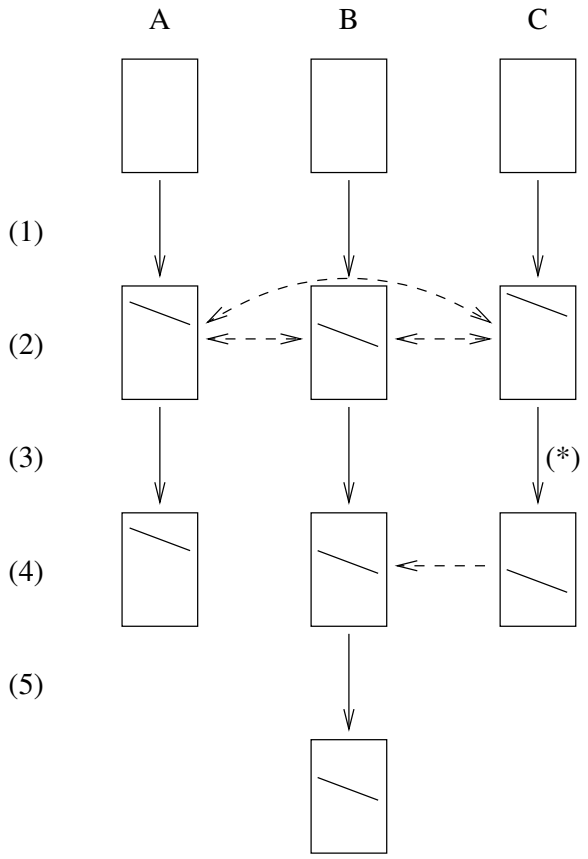


Abbildung 11.2: Anwendung des verteilten Algorithmus auf Π^{bsp}. Die schrägen Linien in den Teilen stehen für die Lage der Sperrzeitentreppe des Zugs ABC. (*): siehe Beispiel 10.8 [S. 69].

ihrer Züge an den Grenzen zu ihren Nachbarn (gestrichelte Linien). Weil also jeder Teil solche *Änderungen* bekommt, muss auch jeder Teil neu berechnet werden (3). Die Berechnung von C (*) wird übrigens im Detail in Beispiel 10.8 [S. 69] beschrieben. In (4) wird wieder kommuniziert: Neuberechnung von A und B ergibt keine Änderung zum vorhergehenden Ergebnis, deshalb werden hier keine Änderungen kommuniziert, C aber teilt B die spätere Anfangszeit von ABC mit. Deshalb wird B nochmals neu berechnet (5). Weil B bei der Neuberechnung keine Differenz zum bisherigen Ergebnis feststellt, muss keine Änderung mehr kommuniziert werden, das Ergebnis ist global konsistent.

Definition 11.1 (Verteilte Simulation Σ)
Für ein Problem Π ist eine *verteilte Simulation* Σ gegeben durch ein Tupel $(\Sigma_p, \mathsf{Sched}, \mathsf{step})$ mit

(1) $\Sigma_p : \mathsf{Assign} \cup \{\mathsf{assign}^0\} \to \mathsf{Assign}_p$
Lokale Simulation, siehe Definition 10.1 [S. 66].

(2) $\mathsf{Sched} : \mathbb{N} \times 2^{\mathsf{P}} \to 2^{\mathsf{P}}$
Das Scheduling der lokalen Simulationen: wann welche Teile neu berechnet werden, in Abhängigkeit der jeweils neu zu berechnenden (in der Regel die inkonsistenten).

(3) $\forall i \in \mathbb{N} : \forall Q \subseteq \mathsf{P} : \mathsf{Sched}(i, Q) \subseteq Q \wedge (Q \neq \emptyset \Rightarrow \mathsf{Sched}(i, Q) \neq \emptyset)$
Q ist die Menge der neu zu berechnenden Teile, und nur davon sollen auch wirklich welche neu berechnet werden.

(4) $\forall (F : \mathbb{N} \to 2^{\mathsf{P}}) :$
$(\exists n \in \mathbb{N}, p \in \mathsf{P} : \forall i \geq n : p \in f(i)) \Rightarrow \exists m \geq n : p \in \mathsf{Sched}(m, F(m))$
Intuitiv: Wenn ab einem Simulationsschritt n ein Teil p für immer zur Berechnung ansteht (inkonsistent ist), soll er auch irgendwann neu berechnet werden. Genauer: siehe Bemerkung 11.2 [S. 73].

(5) $\mathsf{step} : \mathbb{N} \times \mathsf{Assign} \to \mathsf{Assign}$
Ein Simulationsschritt.

(6) $\forall a \in \mathsf{Assign} : \mathsf{step}(0, a) := \mathsf{assign}^1$
Im *0-ten Schritt* ist das Ergebnis gleich assign^1.

(7) $\forall a \in \mathsf{Assign} : \forall i > 0 : \mathsf{step}(i, a) := \mathsf{embed}(\mathsf{Sched}(i, \mathsf{Incons}(a)), a)$
Im i-ten Schritt wählt Sched aus den inkonsistenten Teilen $\mathsf{Incons}(a)$ neu zu berechnende aus. Genau diese werden neu berechnet.

Bemerkung 11.2 (Fairness, Endliche Inkonsistenz)
Eine wichtige Eigenschaft von Definition 11.1 [S. 73] ist (4) – in der Literatur auch *Fairness* genannt. Sie garantiert, dass eine Aufgabe, die unendlich oft (hier sogar hintereinander) ansteht, auch irgendwann zur Ausführung kommt. Tatsächlich impliziert diese Definition, dass diese Aufgabe dann auch unendlich oft ausgeführt wird. Wichtig aber ist die Umkehrung: keine Aufgabe wird unendlich lange verzögert. Oder anders gesagt: Inkonsistenzen werden irgendwann (i.e. nach endlich vielen Schritten) auch wirklich behandelt, Σ verharrt also nie in einem inkonsistenten Zustand.

Formal funktioniert die Definition folgendermaßen: Sched – d.h. die Eigenschaft (4) – gilt für alle $F : \mathbb{N} \to 2^{\mathsf{P}}$ und damit für jede mögliche Folge

$(a_i \in 2^{\mathsf{P}})_{i\in\mathbb{N}}$. Und die $\mathsf{Incons}(a)$ ergeben für die $(\mathsf{step}(i,a))_{i\in\mathbb{N}}$ eine solche Folge. Betrachten wir als Beispiel folgende unendliche Folge:

i	j	$j+1$	$j+2$	$j+3$	$j+4$	$j+5$	$\cdots$
Q	$\{p,p'\}$	$\{p,p''\}$	$\{p,p'\}$	$\{p,p''\}$	$\{p,p'\}$	$\{p,p''\}$	$\cdots$
$\mathsf{Sched}(i,Q)$	$\{p'\}$	$\{p''\}$	$\{p'\}$	$\{p''\}$	$\{p'\}$	$\{p''\}$	$\cdots$

Hier würde der Task oder Teil p nie ausgeführt – man sagt auch *ausgehungert* – obwohl er für immer zur Berechnung ansteht. Die Definition von Sched verhindert genau solche Folgen für die Abbildung step.

Man könnte Sched natürlich auch basierend auf step definieren, dann aber wären Sched und step kompliziert verknüpft. So aber definiere ich eine allgemeine, von step unabhängige Eigenschaft von Sched, die ich auch für step verwenden kann. Tatsächlich ist Sched eine allgemeine Definition für ein faires Scheduling von beliebigen Tasks P!

Das Scheduling Sched wird in der Praxis meist nicht direkt vom Algorithmus vorgegeben, sondern von der Laufzeit-Umgebung (Prozessor, Betriebssystem, Netzwerk, etc.) zusammen mit der Implementierung.

Bemerkung 11.3 (Simulation, Menge von Simulationen)
Anders als Σ_p ist Σ für ein gegebenes Π nicht eindeutig definiert: Für ein gegebenes Π gibt es viele, in der Regel sogar unendlich viele Möglichkeiten für Sched. Für ein gegebenes Sched ist aber step wieder eindeutig definiert. Das Scheduling Sched bestimmt also eindeutig Σ.

Sched reflektiert damit den Nicht-Determinismus des verteilten Algorithmus: Indem die Laufzeit-Umgebung die konkrete Reihenfolge der Arbeitsschritte bestimmt, ist diese nicht von vornherein festgelegt oder für alle Abläufe gleich.

Im Folgenden geht es uns aber um Eigenschaften, die für alle konkreten Abläufe gelten müssen: um Terminierung und Korrektheit. Wir müssen also über alle Abläufe sprechen, über die Menge der möglichen Abläufe. Diese ist induziert von der Menge der Σ zum gegebenen Π.

Definition 11.4 (Mögliche Ausführungen Exec)
Gegeben Π, aber kein spezielles Σ! Die jeweilige *Menge der möglichen Ausführungen* Exec ist in jedem Schritt $i \in \mathbb{N}$ gegeben durch die Menge *aller möglichen Belegungen*:

(1) $\mathsf{Exec} : \mathbb{N} \to 2^{\mathsf{Assign}}$

(2) $\mathsf{Exec}(0) := \{\mathsf{assign}^1\}$

(3) $\forall i > 0 : \mathsf{Exec}(i) := \{\mathsf{step}(i,a) \mid \exists \Sigma = (\Sigma_p, \mathsf{Sched}, \mathsf{step}) \wedge a \in \mathsf{Exec}(i-1)\}$

Bemerkung 11.5
Wir sprechen im Folgenden über alle möglichen Σ zu einem speziellen Π. Ich definiere die Begriffe Konvergenz und Terminierung anhand der Belegungen, die durch die unterschiedlichen Ausführungen von Σ erzeugt werden. Dabei kann jede Belegung durch verschiedene Ausführungen erreicht werden: siehe Abbildung 11.3 [S. 75].

Hier kann $a'' \in \mathsf{Exec}(i+2)$ sowohl von einem Σ^a von a über a' erzeugt sein, als auch von einem anderen Σ^b von b über b', als auch von einem Σ^c von c über b'. In Exec wird also nicht jeder mögliche Ablauf explizit repräsentiert, aber alle

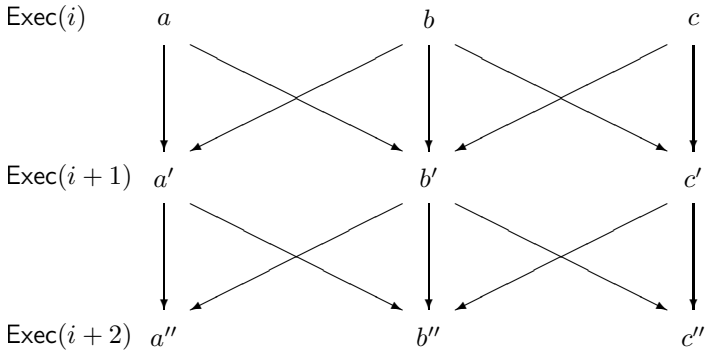


Abbildung 11.3: Mögliche Ausführungsreihenfolgen.

möglichen sind implizit durch die möglichen Belegungen und die entsprechenden Übergänge gegeben.

Es kann freilich nicht jeder beliebige Zustand von jedem beliebigen anderen erreicht werden: beispielsweise ist Σ so konstruiert, dass kein Zug *in die Vergangenheit* verschoben werden kann. Wenn also in einer Belegung a ein Zug *echt früher* liegt als in einer anderen Belegung a', dann kann a niemals von a' aus erreicht werden, möglicherweise aber umgekehrt a' von a aus.

12 Konvergenz

Hier wird der Begriff der *Konvergenz* als Voraussetzung für endliche Terminierung des Algorithmus DRS definiert: Simulationen konvergieren, wenn nach endlich vielen Schritten ein globales Ergebnis erreicht ist (die *Konvergente*), das auch durch folgende Anwendungen von Σ nicht mehr verändert wird. Für ein gegebenes Simulationsproblem gibt es meist mehr als eine Konvergente. Teilbelegungen, die Teil einer Konvergente sind, werden durch die Anwendung von Σ_p nicht verändert. Umgekehrt nimmt Σ_p unter bestimmten Voraussetzungen an einem noch inkonsistenten Teilergebnis immer Änderungen vor. Am Schluss des Kapitels wird hergeleitet, dass DRS immer konvergiert, wenn das Simulationsproblem aus *nur einem Teil* besteht.

Definition 12.1 (Konvergenz)
Die Σ zu einem gegebenen Π heißen *konvergierend*, wenn gilt:

(1) $\exists n \in \mathbb{N} : \forall i >= n : \forall a \in \mathsf{Exec}(i) : \forall \Sigma = (\Sigma_p, \mathsf{Sched}, \mathsf{step}) : \mathsf{step}(i, a) = a$

Wenn die Σ konvergieren, gibt es also eine Menge von Belegungen, deren Elemente durch beliebige Anwendungen von step nicht mehr verändert werden. Eine solche Belegung wird dann immer nach endlich vielen Schritten erreicht.

Lemma 12.2
Wenn die Σ konvergieren, gilt: $\exists n \in \mathbb{N} : \forall i >= n : \mathsf{Exec}(i) = \mathsf{Exec}(n)$

Beweis: Wenn wie in Definition 12.1 [S. 76] gegeben, ab diesem n keine mögliche Simulation Σ an keinem der $a \in \mathsf{Exec}(n)$ mehr etwas verändert, dann ist mit Definition 11.4 [S. 74] $\mathsf{Exec}(n) = \mathsf{Exec}(n+1)$ und identisch allen folgenden. □

Definition 12.3 (Konvergente Conv)
Die Menge $\mathsf{Exec}(n)$ aus Lemma 12.2 [S. 76] heisse *Konvergente* Conv der Simulation:

(1) $\mathsf{Conv} := (\mathsf{Exec}(n) : \forall i >= n : \mathsf{Exec}(i) = \mathsf{Exec}(n))$

Conv ist eine Menge von Belegungen, und jedes konkrete Σ konvergiert gegen ein Element aus Conv.

Bemerkung 12.4 (Eindeutigkeit)
Conv ist für ein gegebenes Π tatsächlich eine *Menge*. Es gilt für den angegebenen Algorithmus nicht, dass jede beliebige (globale) Simulation zu einem gegebenen Simulationsproblem Π immer in derselben Lösung konvergiert. Anders gesagt: Unterschiedliche Abläufe von Σ können unterschiedliche Ergebnisse produzieren.

In Bemerkung 11.3 [S. 74] hatten wir schon festgestellt, dass die Reihenfolge der lokalen Simulationen in verschiedenen konkreten Abläufen unterschiedlich sein kann. Das führt dazu, dass die Endergebnisse unterschiedlich sein können.

Das zu Beweisen ist sehr einfach: es genügt, zwei unterschiedliche Läufe anzugeben, die zu unterschiedlichen Ergebnissen führen. Im Detail möchte ich das hier aber nicht machen. Die Tests mit der Implementierung aber zeigen genau dieses Verhalten, siehe Kapitel 25 [S. 160].

Definition 12.5 (Konsistente Simulation)
Die Σ *konvergieren konsistent*, wenn die Konvergente aus konsistenten Belegungen besteht (siehe auch Definition 9.10, S. 63):

(1) $\mathsf{Conv} \subseteq \mathsf{Solution}$

Lemma 12.6 (Konsistentes a ist Konvergente)
$\mathsf{Incons}(a) = \emptyset \Rightarrow \forall i > 0, \Sigma : \mathsf{step}(i, a) = a$ und damit ist $a \in \mathsf{Conv}$.
Beweis:

$$\begin{gathered}\mathsf{Incons}(a) = \emptyset \\ \text{impl.(Definition 11.1, S. 73)} \\ \mathsf{Sched}(i, \mathsf{Incons}(a)) = \emptyset \\ \text{impl.(Definition 11.1, S. 73)} \\ \mathsf{step}(i, a) = \mathsf{embed}(\mathsf{Sched}(i, \mathsf{Incons}(a)), a) = \mathsf{embed}(\emptyset, a) \overset{\text{Def. 10.9 [S. 70]}}{=} a\end{gathered}$$

□

Definition 12.7 (Later)
Ich definiere die Abbildung Later ähnlich Incons (Definition 9.7, S. 61), allerdings mit dem Unterschied, dass hier nur diejenigen inkonsistenten Teile erfasst werden, deren Zeiten *größer als* (anstatt ungleich) die des Nachbarn sind.

(1) $\mathsf{Later} : \mathsf{Assign} \to 2^{\mathsf{P}}$

(2) $\mathsf{Later}((((\mathsf{start}_{t,p}, \mathsf{dur}_{t,p})_{t \in \mathsf{T}_p})_{p \in \mathsf{P}})) := \{p \in \mathsf{P} \mid \exists t \in \mathsf{T}_p, (s, s') \in \mathsf{crossing}_{t,p} : \mathsf{start}_{t,p}(s) < \mathsf{start}_{t,\mathsf{part}(s')}(s') \vee \mathsf{dur}_{t,p}(s) < \mathsf{dur}_{t,\mathsf{part}(s')}(s')\}$

Korollar 12.8
$\forall a \in \mathsf{Assign} : \mathsf{Later}(a) \subseteq \mathsf{Incons}(a)$.

Lemma 12.9
$\forall a \in \mathsf{Assign} : \mathsf{Later}(a) = \emptyset \Rightarrow \mathsf{Incons}(a) = \emptyset$.
Beweis: Mit $a = (((\mathsf{start}_{t,p}, \mathsf{dur}_{t,p})_{t \in \mathsf{T}_p})_{p \in \mathsf{P}})$ gilt:

$$\begin{gathered}\mathsf{Later}(a) = \emptyset \\ \text{impl.(Definition 12.7, S. 77)} \\ \nexists p \in \mathsf{P} : \exists t \in \mathsf{T}_p, (s, s') \in \mathsf{crossing}_{t,p} : \\ \mathsf{start}_{t,p}(s) < \mathsf{start}_{t,\mathsf{part}(s')}(s') \vee \mathsf{dur}_{t,p}(s) < \mathsf{dur}_{t,\mathsf{part}(s')}(s') \\ \text{impl.(Symmetrie)} \\ \nexists p \in \mathsf{P} : \exists t \in \mathsf{T}_p, (s, s') \in \mathsf{crossing}_{t,p} : \\ \mathsf{start}_{t,p}(s) \neq \mathsf{start}_{t,\mathsf{part}(s')}(s') \vee \mathsf{dur}_{t,p}(s) \neq \mathsf{dur}_{t,\mathsf{part}(s')}(s') \\ \text{impl.(Definition 9.7, S. 61)} \\ \mathsf{Incons}(a) = \emptyset\end{gathered}$$

□

Lemma 12.10 (Idempotenz von Σ_p)
$\forall a = (a_q)_{q \in \mathsf{P}} \in \mathsf{Assign} : \forall p \in \mathsf{P} : p \notin \mathsf{Later}(a) \Rightarrow \Sigma_p(a) = a_p$.

Beweis: Wenn p nicht inkonsistent ist, ergibt die Anwendung von Σ_p auf a keine Veränderung in a_p weil Σ_p alle Züge in p deterministisch (Korollar 10.6, S. 68) zu den minimal möglichen Zeiten einlegt (nach Definition 10.1 (9), S. 67), die außer von der lokalen Geschichte (Definition 10.1 (7), S. 66), also dem Maximum der Zeiten der bisherigen Lösungen, nur von den Zeiten der Nachbarn abhängt und auch das nur, wenn diese größer sind als lokal gegeben (Definition 10.1 (8), S. 66). □

Lemma 12.11 (Fortschritt von Σ_p)
$\forall a = (a_q)_{q \in \mathsf{P}} \in \mathsf{Assign} : \forall p \in \mathsf{P} : p \in \mathsf{Later}(a) \Rightarrow \Sigma_p(a) \neq a_p$.

Das ist die Umkehrung von Lemma 12.10 [S. 77].

Beweis: Wenn p inkonsistent ist, existieren an den Grenzen Zeiten der Nachbarn, die größer sind als die lokal gegebenen. Dann aber passt Σ_p genau diese den Nachbarn, wodurch a_p verändert wird, also $\Sigma_p(a) \neq a_p$.

Indem die neuen Werte nie kleiner, meist aber echt größer sind als die bisherigen, kann man sagen, dass die Züge immer weiter hinausgeschoben werden, oder quasi gilt: $\Sigma_p(a) > a_p$. □

Lemma 12.12 (Isomorphie von Incons und Later bzgl. Konvergenz)
Aus Lemma 12.9 [S. 77] und Lemma 12.10 [S. 77] folgt unmittelbar Isomorphie von Incons und Later bezüglich Konvergenz:

Inkonsistente Teile $p \in \mathsf{Incons}(a)$, die nicht in $\mathsf{Later}(a)$ liegen, werden durch wiederholte Berechnungen von Σ_p nicht verändert. Alle anderen Teile aus $\mathsf{Later}(a)$ sind auch in $\mathsf{Incons}(a)$ und werden deshalb und wegen Definition 11.1 (4) [S. 73] nach endlich vielen Schritten neu berechnet, unabhängig von den Teilen in $\mathsf{Incons}(a) \setminus \mathsf{Later}(a)$. Deshalb ist es für die Terminierung von Σ unerheblich, ob wir jeweils alle Teile aus $\mathsf{Incons}(a)$ betrachten oder nur die aus $\mathsf{Later}(a)$. Ich werde mich im Folgenden meist auf $\mathsf{Later}(a)$ beschränken.

Bemerkung 12.13
Ich zeige im Folgenden, dass jedes Σ konvergiert und die Konvergente konsistent ist, also eine Lösung.

Dabei unterscheide ich die wesentlichen Fälle: ein Teil ($|\mathsf{P}| = 1$) und mehrere Teile ($|\mathsf{P}| > 1$), ein Zug ($|\mathsf{T}| = 1$) und mehrere Züge ($|\mathsf{T}| > 1$).

Lemma 12.14 (Konvergenz für $|\mathsf{P}| = 1$)
Die Σ konvergieren für alle Probleme Π mit $|\mathsf{P}| = 1$ und T beliebig.

Beweis: $\forall a, \Sigma : \mathsf{step}(0, a) = \mathsf{assign}^1$ (Definition 11.1 (6), S. 73). Mit Lemma 12.6 [S. 77] ist nur noch zu zeigen, dass $\mathsf{Incons}(\mathsf{assign}^1) = \emptyset$:

$$\forall t \in \mathsf{T}, p \in \{p\} = \mathsf{P} : \mathsf{crossing}_{t,p} = \emptyset$$
$$\text{impl.(Definition 9.7, S. 61)}$$
$$\mathsf{Incons}(\mathsf{assign}^1) = \emptyset$$

Deshalb ist $\mathsf{Exec}(i) = \{\mathsf{assign}^1\}$ für alle i und damit konvergieren alle Σ mit $\mathsf{Conv} = \{\mathsf{assign}^1\}$. □

Bemerkung 12.15
Im Folgenden gilt $|\mathsf{P}| > 1$.

13 Konvergenz für einen Zug

In diesem Kapitel wird gezeigt, dass DRS immer endlich konvergiert, wenn das gesamte Problem aus *mehr als einem Teil, aber nur einem Zug* besteht. Die Grundidee des Beweises ist: Für den Zug gibt es im Fahrplan immer einen *führenden Fahrplaneintrag*, der die Lage der Zugfahrt auf der Zeitachse bestimmt. Bei der verteilten Simulation kann die zugehörige Belegungszeit andere Belegungszeiten verschieben, selbst aber nicht verschoben werden.

Ich spreche hier über alle Probleme Π mit $|\mathsf{P}| > 1$ und $\mathsf{T} = \{t\}$ und die davon induzierten Σ.

Bemerkung 13.1
In diesem Fall spielen die Blockausschluss-Bedingungen (Definition 9.1 (6), S. 60) keine Rolle. Ein Zug kann ja mit keinem anderen in Konflikt treten. Da jeder Zug jeden Abschnitt höchstens ein Mal benutzt, kommt sich also schon deshalb kein Zug irgendwo selbst in die Quere. Ich werde die Blockausschluss-Bedingungen deshalb in diesem Kapitel nicht berücksichtigen.

Bemerkung 13.2
In Abbildung 13.1 [S. 80] werden die unterschiedlichen Zusammenhänge der zu bestimmenden Größen dargestellt: für vier verschiedene Situationen in jeweils einem Teil mit drei Abschnitten die Abhängigkeiten der Größen start_t und dur_t von den Bedingungen für mindur_t und minend_t. Jede Teilgrafik ist als Graph zu verstehen, in dem der Weg von links nach rechts und die Zeit von oben nach unten aufgetragen ist. Die etwas dickeren senkrechten Linien bezeichnen die Teil-Grenzen, die darin enthaltenen drei Rechtecke beschreiben für die drei Abschnitte den Belegungs-Beginn start_t und das Belegungs-Ende $\mathsf{start}_t + \mathsf{dur}_t$. Die Dreiecke symbolisieren die Fahrplanbedingung minend_t (das Belegungsende darf nicht vor minend_t liegen): wenn ein Dreieck ausgefüllt ist, ist die Fahrplanzeit identisch mit dem entsprechenden Belegungsende, in diesem Fall hat die Bedingung quasi *aktiv* zu dieser Wertebelegung beigetragen. Wenn das Dreieck nicht ausgefüllt ist, ist das tatsächliche Belegungsende ungleich (und damit größer als) der Fahrplan-mäßigen Zeit und implizit durch andere Bedingungen gegeben.

Die Pfeile geben die Mindestbelegungszeiten mindur_t an (die Linien für Start und Ende dürfen nicht näher zusammenrücken, als durch den Pfeil gegeben ist), die Doppelpfeile (mit X-Spitze) geben die Überlappung an, die ja konstant definiert ist. Die Bedingungen außerhalb der Begrenzungslinien des jeweiligen Teils symbolisieren die Bedingungen des Nachbar-Teils, die ja über den Algorithmus Σ_p Einfluss auf die Größen des lokalen Teils nehmen.

Ich betrachte die verschiedenen Fälle (entsprechend Abbildung 13.1, S. 80), wie sich Σ_p und die Bedingungen auf die Werte der zu bestimmenden Größen auswirken. Wie oben schreibe ich mit $\mathsf{T} = \{t\}$: $\Sigma_p(a) = a'_p$, $a = (\mathsf{start}_{t,q}, \mathsf{dur}_{t,q})_{q \in \mathsf{P}}$, $a'_p = (\mathsf{start}'_{t,p}, \mathsf{dur}'_{t,p})$, $\mathsf{start}_t(s) := \mathsf{start}_{t,\mathsf{part}(s)}$, $\mathsf{dur}_t(s) := \mathsf{dur}_{t,\mathsf{part}(s)}$.

(1) Hier haben wir als Spezialfall, dass alle Fahrplan-Bedingungen konsistent sind mit den Fahrzeit-Bedingungen und deshalb jedes Belegungs-Ende identisch ist mit der Fahrplan-Bedingung (alle sind *aktiv*). Wenn in dieser Situation die Ausfahrzeit $e = \mathsf{start}'_t(s) + \mathsf{dur}'_t(s)$ mit $s := \mathsf{last}_t(p)$ auf ihr Minimum gesetzt werden soll, ist dieses gleich $\mathsf{minend}_t(s)$. Die dur'_t aller anderen Abschnitte werden anschließend durch Σ_p ebenfalls auf ihr jeweiliges Minimum gesetzt, jeweils konsistent mit dem jeweiligen Wert von minend_t.

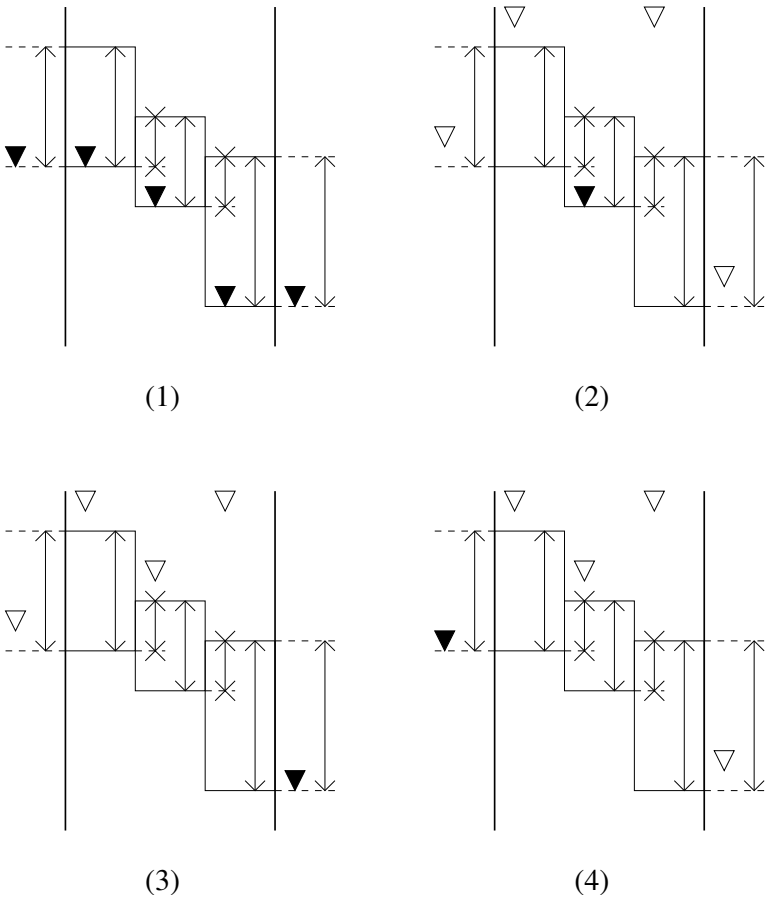


Abbildung 13.1: Abhängigkeiten der durch die Simulation zu bestimmenden Größen.

Dies ist wie gesagt ein Spezialfall. Häufiger sind folgende Fälle.

(2) In diesem Fall ist nur eine Fahrplan-Bedingung konsistent mit den Fahrzeit-Bedingungen: die eines innen liegenden Abschnitts, typischerweise in einem Bahnhof. Wenn hier durch Σ_p die Ausfahrzeit e auf ihr Minimum gesetzt wird, ist diese größer als die eigentlich durch den Fahrplan vorgesehene Ausfahrzeit. Wir sehen hier, dass diese an einer Abschnittsgrenze in den beteiligten Abschnitten durchaus unterschiedlich sein kann; der Zug muss so fahren, dass er keine von beiden unterschreitet. Durch die dann per Σ_p gegebene minimale Durchfahrzeit durch alle Abschnitte und damit den ganzen Teil ergibt für die Einfahrt ebenfalls ein Belegungsende das später liegt als vorgesehen.

(3) Dieser Fall ist ähnlich. Hier wird die früheste Ausfahrzeit allein durch den Fahrplaneintrag des Nachbarabschnitts bestimmt. Alle anderen Zeiten sind durch die minimalen Durchfahrzeiten eindeutig gegeben, keine der anderen Fahrplan-Bedingungen ist *aktiv*.

(4) Ähnlich in diesem Fall. Hier aber ist die früheste Ausfahrzeit nicht explizit durch einen Fahrplan-Eintrag an der Ausfahrt gegeben, sondern implizit durch die früheste Einfahrzeit und die minimalen Durchfahrzeiten.

Definition 13.3 (Lokal führender Fahrplaneintrag und Abschnitt $\underline{s}_{t,p}$)
An obiger Betrachtung sehen wir, dass, wenn ein Zug einen Teil zur frühestmöglichen Zeit durchfährt (Abbildung 13.1 [S. 80], (1) und (2)), es mindestens einen Fahrplaneintrag gibt, der dafür verantwortlich ist, dass der Zug nicht noch früher fahren kann. In obigen Spezialfällen sind diese *aktiv*.

Ich nenne diesen Eintrag im Folgenden den *lokal führenden Fahrplaneintrag*, den zugehörenden Abschnitt nenne ich entsprechend den *lokal führenden Abschnitt*. Offenbar ist er durch die gegebenen Bedingung klar bestimmt, also unabhängig von konkreten Belegungen. Es kann aber mehrere (gleichberechtigte) geben: dann sei der lokal führende ein beliebiger davon.

Ich schreibe für den *lokal führenden Abschnitt* des Zugs t im Teil p: $\underline{s}_{t,p}$.

Korollar 13.4 (Unabhängigkeit von dur_t)
An Bemerkung 13.2 [S. 79] sehen wir auch, dass in jedem möglichen Fall die Belegung der dur_t, der Durchfahrt-Dauern also, unabhängig ist von den start_t. Für den vorliegenden Fall (ein Zug) sind die Dauern nur abhängig von den gegebenen mindur_t.

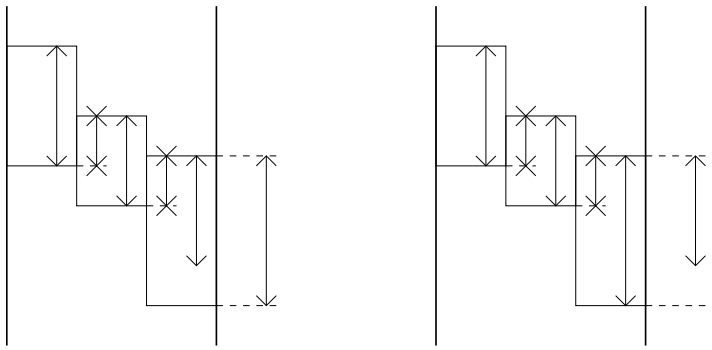
Bemerkung 13.5
Allerdings sind die dur_t nicht nur von den *lokalen* mindur_t abhängig (s.a. Abbildung 13.2, S. 82):

Gegeben Teil p, Zug t, Einfahr-Abschnitt $s_< := \mathsf{first}_t(p)$, Ausfahr-Abschnitt $s_> := \mathsf{last}_t(p)$, $s'_< : (s_<, s'_<) \in \mathsf{crossing}_{t,p}$, $s'_> : (s_>, s'_>) \in \mathsf{crossing}_{t,p}$.

Dann muss z.B. gelten: $\mathsf{dur}'_t(s_<) \geq \mathsf{mindur}_t(s_<) \wedge \mathsf{dur}'_t(s_<) \geq \mathsf{dur}_t(s'_<) \wedge \mathsf{dur}_t(s'_<) \geq \mathsf{mindur}_t(s'_<)$, mithin $\mathsf{dur}'_t(s_<) \geq \mathsf{mindur}_t(s'_<)$.

Lemma 13.6 (Konvergenz der dur_t)
Die dur_t sind nach endlich vielen Schritten global konsistent.

Beweis: Gegeben wie oben Teil p, Zug t, Abschnitte $s_>, s'_>$ ($s_<, s'_<$ funktionieren entsprechend). Siehe Abbildung 13.2 [S. 82]:

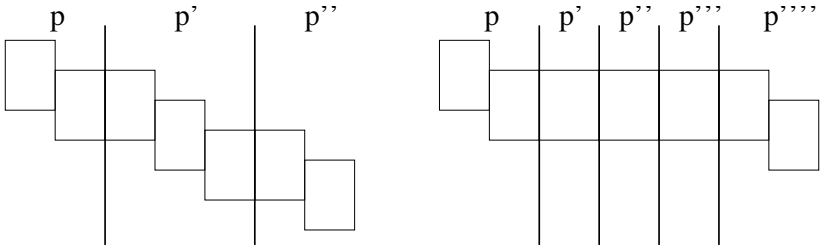


(1) (2)

Abbildung 13.2: Abhängigkeiten der durch die Simulation zu bestimmenden Größen: unterschiedliche Mindest-Dauern an der Ausfahrt. Die tatsächliche Dauer dur_t muss größer oder gleich beiden sein.

(1) Hier ist nach Σ_p $\mathsf{dur}'_t(s_>) = \mathsf{dur}_t(s'_>)$ und damit schon konsistent mit dem Nachbarn.

(2) Hier ist $\mathsf{dur}'_t(s_>) = \mathsf{mindur}_t(s_>) > \mathsf{dur}_t(s'_>)$ inkonsistent mit dem Nachbarn. Wir müssen wieder zwei Fälle unterscheiden: siehe Abbildung 13.3 [S. 82].



(a) (b)

Abbildung 13.3: Belegungsdauern über Teilgrenzen hinweg in unterschiedlichen Konstellationen.

(a) Hier fährt der Zug von p nach p' nach p''. p' besteht aus mehr als einem Abschnitt, deshalb muss hier nur das lokale Minimum ($\mathsf{mindur}_t(s'_>)$) berücksichtigt werden. Das geschieht bereits bei der nächsten Berechnung von p' und ab dann sind diese beiden Teile bezüglich dur_t konsistent.

(b) Der Zug fährt hier durch mehrere Teile, die aus je nur einem Abschnitt bestehen (bzgl. t). Das hat zur Folge, dass für alle beteiligten dur_t die Mindest-Belegungszeit mindur_t aller anderen berücksichtigt werden

muss: jedes dur_t muss gleich sein dem Maximum aller entsprechenden mindur_t.
Diese Konsistenz wird aber offensichtlich nach endlich vielen Schritten hergestellt: Ausgehend von z.B. dem gegebenen Abschnitt wird quasi in Fahrtrichtung des Zugs das Maximum der Mindest-Belegungszeiten propagiert bis zu dem Teil, der für den (einzig relevanten) Abschnitt das Maximum aller beteiligten fordert. Und von diesem ausgehend, werden die dur_t *rückwärts* zum aktuellen und *vorwärts* bis zum letzten dieser Reihe *propagiert*.
Immer natürlich vorausgesetzt, dass es nur endlich viele Teile und Abschnitte gibt (Definition 8.1, S. 54) und dass jeder noch nicht konsistente Teil nach endlich vielen Schritten neu berechnet wird (Sched, Definition 11.1 (4), S. 73).

□

Definition 13.7 (Konsistente Zwischenbelegung inter)
Wir wissen nun, dass die dur_t unabhängig von den start_t konvergieren, also diesbezüglich nach endlich vielen Schritten ein konsistenter Zustand hergestellt ist. Sei $\underline{\mathsf{inter}} \in \mathsf{Assign}$ ein solcher Zustand.

Σ_p legt dann auch in allen folgenden Schritten für alle $p \in \mathsf{P}$ und alle $s \in \mathsf{S}_p$ alle dur_t minimal fest. Daraus folgt, dass in allen folgenden Schritten die Durchfahrzeiten in den einzelnen Abschnitten konstant sind und damit für jeden Teil die Durchfahrzeit konstant ist.

$\underline{\mathsf{inter}}$ ist nicht eindeutig bestimmt! Weil es hier nicht um Eindeutigkeit des Ergebnisses geht (siehe hierzu Abschnitt 16.2, S. 92), sondern um Terminierung des Algorithmus, spielt das hier keine Rolle. Wichtig ist nur, dass ein solcher Zustand irgendwann (nach endlich vielen Schritten) existiert und welche Eigenschaften er hat.

Abbildung 13.4 (b) [S. 84] zeigt eine solche, quasi *teilkonsistente*, Zwischenbelegung zusammen mit den entsprechenden führenden Fahrplaneinträgen. Sie liegt hier in fast jedem Teil deutlich später als die Fahrplaneinträge des Zuges vorgeben. Das muss nicht sein. Aber kein Teil von $\underline{\mathsf{inter}}$ kann früher als ein Fahrplaneintrag liegen, weil $\underline{\mathsf{inter}} \in \mathsf{Assign}$!

Definition 13.8 (dur-Konvergente dur)
Wie gesagt gilt für $\underline{\mathsf{inter}}$ und alle nachfolgenden durch Σ berechneten Belegungen, dass die Durchfahrzeiten in den einzelnen Abschnitten und damit für jeden Teil die Durchfahrzeit eindeutig bestimmt ist.

Sei $d(p)$ für jeden Teil $p \in \mathsf{P}_t$ diese Durchfahrzeit. Außerdem sei $a(p) := \mathsf{pno}_t^{-1}(\mathsf{pno}_t(p) - 1)$ der Vorgänger-Teil zu p bezüglich Zug t und wie oben $s_<(p) := \mathsf{first}_t(p)$ der Einfahrabschnitt des Zuges t in den Teil p.

Damit definiere ich die dur-Konvergente $\underline{\mathsf{dur}}$:

(1) $\underline{\mathsf{dur}} : \mathsf{P}_t \to \mathbb{N}$

(2) $$\underline{\mathsf{dur}}(p) := \begin{cases} 0 & \mathsf{pno}_t(p) = 1 \\ \underline{\mathsf{dur}}(a(p)) + d(a(p)) - \mathsf{dur}_{t,p}(s_<(p)) & \text{sonst} \end{cases}$$

Betrachte Abbildung 13.4 (a) [S. 84]: Im ersten Teil ist $\underline{\mathsf{dur}}(p) = 0$. In allen folgenden Teilen p ist $\underline{\mathsf{dur}}(p)$ so definiert, dass die Startzeit im ersten Abschnitt

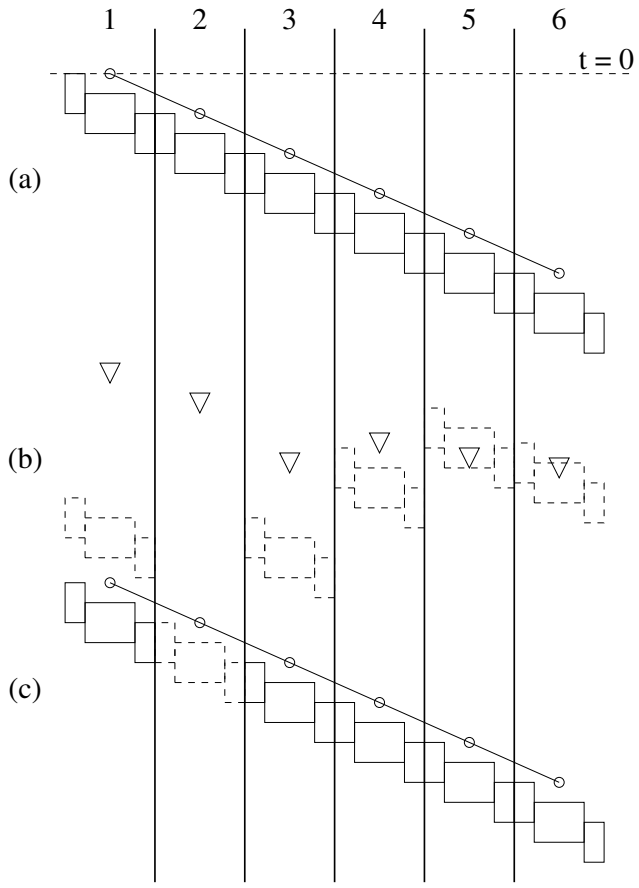


Abbildung 13.4: Die Schritte von einer bzgl. dur konsistenten Belegung inter zu einer, die auch bzgl. start konsistent ist: Hier beispielhaft sechs Teile $\{p_1, ..., p_6\}$ mit $\mathsf{pno}_t(p_1) = 1, ..., \mathsf{pno}_t(p_6) = 6$.
(a) dur-Konvergente: die Gerade ist der Graph von dur.
(b) Bzgl. dur (aber noch nicht bzgl. start) konsistente Zwischenbelegung inter (gestrichelte Kästchen) und in jedem Teil der *lokal führende Fahrplaneintrag* $\mathsf{minend}_{t,p}(\underline{s}_{t,p})$ (Dreiecke).
(c) Endgültige Konvergente start: im Teil $w = p_2$ ist sie identisch mit inter.

von p ($s_<(p)$) gleich der Startzeit im letzten Abschnitt des vorhergehenden Teils $a(p)$ ist. Damit ist $\underline{\mathsf{dur}}$ konsistent bezüglich der gegebenen Durchfahrzeiten und den Startzeiten! In der Abbildung haben übrigens alle Teile p die gleiche Durchfahrzeit $d(p)$, weshalb der Graph von $\underline{\mathsf{dur}}$ eine Gerade ist.

Bemerkung 13.9
Anders als $\underline{\mathsf{inter}}$ ist $\underline{\mathsf{dur}}$ nicht als Belegung definiert, sondern vereinfachend als Abbildung $\mathsf{P}_t \to \mathbb{N}$. $\underline{\mathsf{dur}}$ gibt damit nicht im Detail die Belegung wieder, sondern nur die relative Position der lokalen Teil-Belegungen. Diese sind ja jeweils konsistent und verändern sich intern durch Anwendungen von Σ_p bzw. Σ nicht mehr, wie wir seit Definition 13.7 [S. 83] wissen. $\underline{\mathsf{dur}}$ – und weiter unten $\underline{\mathsf{start}}$ – ist also der Übersichtlichkeit halber vereinfacht, könnte aber natürlich trivial zu einer Belegungs-Struktur erweitert werden:

Die von $\underline{\mathsf{dur}}$ induzierte Belegung $(\mathsf{start}_{t,p}^{\underline{\mathsf{dur}}}, \mathsf{dur}_{t,p}^{\underline{\mathsf{dur}}})_{p\in\mathsf{P}}$ ist mit $\underline{\mathsf{inter}} = (\mathsf{start}_{t,p}^{\underline{\mathsf{inter}}}, \mathsf{dur}_{t,p}^{\underline{\mathsf{inter}}})_{p\in\mathsf{P}}$:

(1) $\mathsf{start}_{t,p}^{\underline{\mathsf{dur}}}(s) := \mathsf{start}_{t,p}^{\underline{\mathsf{inter}}}(s) - \mathsf{start}_{t,p}^{\underline{\mathsf{inter}}}(\mathsf{first}_t(p)) + \underline{\mathsf{dur}}(p)$

(2) $\mathsf{dur}_{t,p}^{\underline{\mathsf{dur}}}(s) := \mathsf{dur}_{t,p}^{\underline{\mathsf{inter}}}(s)$

Die dur-Konvergente $\underline{\mathsf{dur}}$ ist schon die für den Zug endgültige *relative* Position der Belegungszeiten zueinander, bestimmt durch die gegebenen Dauern. Das Ergebnis von Σ wird dann eine Verschiebung dieser Konvergente sein. Die absolute Position wird bestimmt durch die Fahrplaneinträge und die bei $\underline{\mathsf{inter}}$ bereits gegebene Position der Belegungszeiten.

Definition 13.10 ($\underline{\mathsf{inter}}$-$\underline{\mathsf{dur}}$-Abstand $\underline{\mathsf{interdur}}$)
Betrachte Abbildung 13.4 (b) [S. 84]: Die gestrichelten Kästchen beschreiben die oben definierte Zwischenbelegung $\underline{\mathsf{inter}}$.

Ich definiere den jeweiligen Abstand $\underline{\mathsf{interdur}}$ zwischen $\underline{\mathsf{inter}}$ und der dur-Konvergente $\underline{\mathsf{dur}}$. Sei wieder $\underline{\mathsf{inter}} = (\mathsf{start}_{t,p}^{\underline{\mathsf{inter}}}, \mathsf{dur}_{t,p}^{\underline{\mathsf{inter}}})_{p\in\mathsf{P}}$.

(1) $\underline{\mathsf{interdur}} : \mathsf{P}_t \to \mathbb{N}$

(2) $\underline{\mathsf{interdur}}(p) := \mathsf{start}_{t,p}^{\underline{\mathsf{inter}}}(\underline{s}_{t,p}) - \underline{\mathsf{dur}}(p)$

$\underline{s}_{t,p}$ ist der *lokale führende Fahrplaneintrag* aus Definition 13.3 [S. 81].

Definition 13.11 (Verschobene und endgültige Konvergente $\underline{\mathsf{start}}$)
Sei $w \in \mathsf{P}$ ein Teil, sodass $\nexists q \in \mathsf{P} : \underline{\mathsf{interdur}}(q) > \underline{\mathsf{interdur}}(w)$, also so, dass $\underline{\mathsf{interdur}}$ maximal ist. Dann ist $\underline{\mathsf{start}}$ definiert durch:

(1) $\underline{\mathsf{start}} : \mathsf{P}_t \to \mathbb{N}$

(2) $\underline{\mathsf{start}}(p) := \underline{\mathsf{interdur}}(p) + \underline{\mathsf{dur}}(w)$

Zur Veranschaulichung siehe nochmal Abbildung 13.4 (c) [S. 84]: Die dur-Konvergente $\underline{\mathsf{dur}}$ ist so weit in die Zukunft verschoben, dass sie im Teil $w : \mathsf{pno}_t(w) = 2$ mit $\underline{\mathsf{inter}}$ identisch ist. w ist der Teil, der die absolute Position von $\underline{\mathsf{dur}}$ bestimmt. So ist $\underline{\mathsf{start}}$ die globale Konvergente für die Fahrt des Zugs durch alle Abschnitte.

Hier ist wieder $\underline{\mathsf{start}}$ vereinfacht als Abbildung $\mathsf{P}_t \to \mathbb{N}$ definiert und nicht als Belegung, siehe auch Bemerkung 13.9 [S. 85].

Lemma 13.12 (Belegungen überschreiten niemals start)
Die Konvergente start wird auch durch wiederholte Anwendungen von Σ_p (per Σ) in keinem Teil $p \in \mathsf{P}$ jemals überschritten.

Dadurch überschreitet natürlich auch keine *vollständige* Belegung jemals start.

Beweis: Per Definition liegt die Belegung in keinem Teil von inter jenseits start. Σ_p verschiebt die Belegung in Teil p aber niemals über die in einem Nachbarn hinaus: immer gerade so weit, dass die Startzeiten des ersten und des letzten Abschnitts mit den entsprechenden in den Nachbarabschnitten übereinstimmen. Aus obigen Definitionen folgt dann, dass eine solche Verschiebung niemals eine Belegung über start hinaus in die Zukunft verschiebt.

Damit wird in keinem Teil p die Konvergente start jemals überschritten. □

Lemma 13.13 (start wird in allen Teilen erreicht)
Gegeben inter und die daraus folgende Konvergente start. Die Teil-Belegungen aller $p \in \mathsf{P}_t$ konvergieren in start.

Damit konvergieren natürlich auch alle *vollständige* Belegungen in start.

Beweis: Seit Lemma 12.10 [S. 77] und Lemma 12.11 [S. 78] wissen wir, dass Σ erst dann keine Veränderung mehr an einer Belegung a vornimmt, wenn $\mathsf{Later}(a) = \emptyset$ und damit a konsistent ist. Das aber ist erst dann der Fall, wenn in allen Teilen p die Belegung die Konvergente start erreicht hat. Im Teil w aus Definition 13.11 [S. 85] nämlich ist start schon in inter erreicht. Die Nachbarn von w sind erst dann mit w konsistent, wenn sie in start liegen. Deren Nachbarn ebenfalls usw. □

Lemma 13.14 (Konvergenz der start_t)
Nach endlich vielen Schritten sind ausgehend von inter die start_t konsistent.

Beweis: Folgt direkt aus Lemma 13.12 [S. 86] und Lemma 13.13 [S. 86]. □

Lemma 13.15 (Konvergenz für $|\mathsf{P}| > 1 \wedge |\mathsf{T}| = 1$)
Die Σ konvergieren für $|\mathsf{P}| > 1 \wedge |\mathsf{T}| = 1$.

Beweis: Hier ist zu zeigen, dass für einen Zug die Anfangszeiten start_t und Belegungsdauern dur_t nach endlich vielen Schritten durch alle Teile propagiert werden.

Laut Korollar 13.4 [S. 81] entwickeln sich die dur_t unabhängig von den start_t. Unter anderem deshalb konvergieren die dur_t nach endlich vielen Schritten: Lemma 13.6 [S. 81]. Sobald eine solche bezüglich dur_t konsistente Belegung inter existiert, konvergieren nach Lemma 13.14 [S. 86] auch die start_t nach wiederum endlich vielen Schritten. Deshalb sind immer nach endlich vielen Schritten die start_t und dur_t konsistent, die Konvergente start ist erreicht. □

14 Konvergenz für mehrere Züge

Hier wird gezeigt, dass DRS auch dann endlich konvergiert, wenn das Problem mehr als einen Zug und mehr als einen Teil enthält. Hier spielt die globale Zug-Reihenfolge eine wichtige Rolle. Sie verhindert, dass Züge sich gegenseitig verdrängen können. Dann kann man zeigen, dass der erste der n Züge nach endlich vielen globalen Iterationsschritten nicht mehr verschoben wird, und per Induktion, dass dann Stück für Stück alle anderen Züge irgendwann nicht mehr verschoben werden.

Bemerkung 14.1
Wir wissen nun also, dass (jedes) Σ konvergiert, wenn wir beliebig viele Züge haben aber nur einen Teil (Lemma 12.14, S. 78) und wenn wir mehr als einen Teil haben aber nur einen Zug (Lemma 13.15, S. 86). Ich zeige jetzt, dass Σ auch dann konvergiert, wenn wir mehr als einen Teil *und* mehr als einen Zug haben. Erstaunlicherweise ist dieser Fall erheblich einfacher als der vorige.

Lemma 14.2 (Σ konvergiert für $|\mathsf{P}| > 1 \wedge |\mathsf{T}| > 1$)
Auch wenn $|\mathsf{T}| > 1$ benötigt (jedes) Σ immer nur endlich viele Schritte, um eine konsistente Belegung zu berechnen.

Beweis: In diesem Fall kann jeder Zug mit jedem anderen in Konflikt geraten: über die Blockausschlussbedingung, Definition 9.1 (6) [S. 60]. Anders als im vorherigen Kapitel müssen wir diese Bedingung hier also berücksichtigen.

Gegeben $n = |\mathsf{T}|$. Ich beweise per Induktion nach $i = 1, ..., n$: die Belegung jedes Zugs $t = \mathsf{tno}^{-1}(i)$ ist nach endlich vielen Schritten global konsistent.

$i = 1$ Sei $t = \mathsf{tno}^{-1}(1)$. Laut Definition 10.1 (9) [S. 67] wird dieser Zug in jedem Teil immer zuerst behandelt. Auch wenn er überall gemäß den zu berücksichtigenden Bedingungen belegt werden muss, ist dadurch bei t kein anderer Zug zu berücksichtigen. Mithin gelten für dieses t genau die Voraussetzungen für Lemma 13.15 [S. 86]. Deswegen konvergiert Σ für $t = \mathsf{tno}^{-1}(1)$ nach endlich vielen Schritten.

$i > 1$ Sei als Induktionsvoraussetzung gegeben: alle $j < i$ konvergieren endlich.

Also liegen nach endlich vielen Schritten alle $t_j = \mathsf{tno}^{-1}(j), j < i$ fest. Sei a eine solche Belegung. Dann gilt für Zug $t = \mathsf{tno}^{-1}(i)$: die schon eingelegten Züge t_j sind quasi *Hindernisse* für t. Ich unterscheide zwei Fälle:

(1) t liegt nach endlich vielen Schritten konsistent zeitlich *zwischen den existierenden* Zügen:

Dann sind wir fertig (t ist konsistent nach endlich vielen Schritten).

In diesem Fall ist übrigens die Fahrtdauer dur nicht wirklich unabhängig von der Startzeit start. Je nach Lage des Zugs in der Zeit (also start) können unterschiedliche andere Züge $t_j = \mathsf{tno}^{-1}(j), j < i$ den Zug t verdrängen und dadurch lokal unterschiedliche Fahrtdauern erzwingen. Nachdem wir uns in diesem Fall aber nicht auf Lemma 13.15 [S. 86] stützen, ist die Unabhängigkeits-Bedingung auch nicht notwendig.

(2) t liegt nach endlich vielen Schritten *nicht* zwischen den existierenden Zügen:

Dann wird die Belegung in mindestens einem Teil hinter alle anderen Züge hinausgeschoben: Wegen Lemma 12.11 [S. 78] (Fortschritt von Σ_p) und weil ja sonst schon vor den anderen die endliche Konvergenz erreicht wäre. Dieser Teil aber zieht dann alle anderen mit und genau wie in Lemma 13.13 [S. 86] und dann Lemma 13.15 [S. 86] konvergiert der Prozess wieder endlich. t liegt dann zeitlich ($\mathsf{start}_{t,p}$) später als alle anderen Züge.

□

Bemerkung 14.3 (Notwendigkeit der Zug-Reihenfolge)
Die Bedingung, dass Σ_p in jedem Teil die Züge in einer global fest vorgegebenen Reihenfolge bearbeitet, ist notwendig für die Terminierung. Insbesondere kann ohne diese Bedingung der Induktionsbeweis nicht geführt werden. Abbildung 14.1 [S. 89] zeigt, was passieren kann, wenn diese Bedingung nicht gegeben ist:

(a) Hier sind drei aufeinander folgende Schritte für zwei angrenzende Teile und zwei sich entgegenkommende Züge dargestellt. im 1. Schritt wird im linken Teil der von links nach rechts fahrende Zug (Treppe von links oben nach rechts unten) zuerst behandelt (i) und dann (ii) der von rechts nach links fahrende (Treppe von rechts oben nach links unten). Im 2. Schritt dann werden im rechten Abschnitt die Anfangszeiten und Fahrtdauern im rechten Teil angeglichen. Das geschieht (gemäß Σ_p) wieder in derselben Reihenfolge: zuerst der Zug von links nach rechts (i), dann der andere (ii). Schon nach der dritten Iteration sind die Belegungen für beide Züge global konsistent.

(b) Anders hier: Hier wird immer im linken Teil zuerst (i) der Zug behandelt, der von links nach rechts fährt, dann (ii) der von rechts nach links und im rechten Teil werden beide Züge immer in der dazu umgekehrten Reihenfolge behandelt. Also ist jeweils lokal die Reihenfolge konstant, aber global nicht konsistent. Und dann können sich die beiden Züge unendlich oft gegenseitig verdrängen: Im ersten Schritt wird der von links nach rechts fahrende zuerst behandelt und wird dadurch zeitlich vor die anderen gelegt. Der von rechts nach links fahrende wird vom ersten verdrängt. In der zweiten Iteration geschieht im anderen Teil dasselbe nur eben spiegelverkehrt: jetzt verdrängt der von rechts nach links fahrende Zug den anderen. Im dritten Schritt geht es genauso weiter, jetzt wieder auf der rechten Seite. Und so verdrängen sich die beiden Züge für immer.

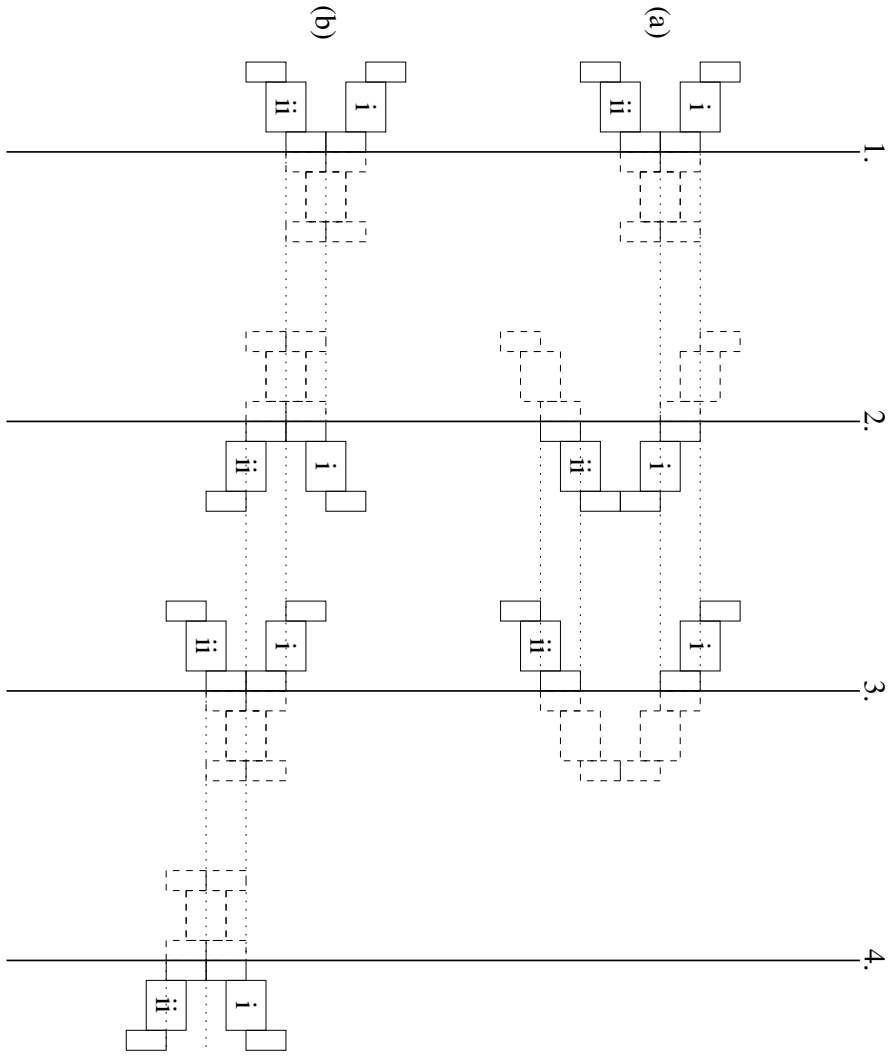


Abbildung 14.1: Was passiert, wenn die Züge nicht in jedem Teil in der gleichen Reihenfolge belegt werden: in (b) verdrängen sich zwei Züge gegenseitig unendlich oft.

15 Korrektheit und Terminierung

Aus den vorhergehenden Abschnitten folgt, dass DRS in allen möglichen Fällen endlich terminiert. Daraus lässt sich folgern, dass das globale Ergebnis korrekt sein muss. Das gilt sogar für unendliche Zeitskalen. Im konkreten Ablauf aber sind Zeitskalen immer endlich. Dann aber kann man folgern, dass DRS bei genügend groß gewählter Zeitskala immer eine Lösung findet.

Satz 15.1 (Terminierung von Σ)
Wir wissen nun, dass Σ in folgenden Fällen nach endlich vielen Schritten eine global konsistente Belegung findet:

$|\mathsf{P}| = 1$ wegen Lemma 12.14 [S. 78]

$|\mathsf{P}| > 1 \wedge |\mathsf{T}| = 1$ wegen Lemma 13.15 [S. 86]

$|\mathsf{P}| > 1 \wedge |\mathsf{T}| > 1$ wegen Lemma 14.2 [S. 87]

Also in allen Fällen. Wegen Lemma 12.6 [S. 77] konvergiert dann Σ immer (die gefundene global konsistente Belegung wird nicht mehr verändert) jeweils mit dieser Lösung.

Satz 15.2 (Korrektheit)
Σ liefert also immer eine global konsistente Lösung.

Wegen Korollar 10.7 [S. 68] ist außerdem jede Lösung in allen Teilen lokal korrekt. Damit ist jede konvergente Lösung von Σ lokal korrekt und global konsistent und damit global korrekt.

Bemerkung 15.3 (Zeitskala)
In unseren formalen Betrachtungen konnten wir die Zeitskala als unendlich annehmen: Alle Abbildungen der Simulationsalgorithmen, die die Simulationszeit betreffen, gehen in die Menge der natürlichen Zahlen. Damit konnten wir für die verteilte Simulation zeigen, dass sie immer nach endlich vielen *Schritten* ein korrektes Ergebnis liefert.

Jede konkrete lokale Simulation aber ist immer endlich. Insbesondere wenn – wie im SIMONE-Projekt – der lokale Simulator einen FD-Constraint-Solver (siehe Abschnitt 3.4.4, S. 26) benutzt, müssen wir die Zeitskala als endlich annehmen. Sonst kann man typischerweise nicht mehr davon ausgehen, dass der Solver das CSP in endlicher Zeit löst: Das widersprüchliche Problem `A #< B, B #< A` beispielsweise könnte über unendlichen Wertebereichen mit klassischen nichtsymbolischen Propagations-Verfahren in endlicher Zeit *nicht* gelöst werden, die Propagation würde sich ähnlich wie in Bemerkung 14.3 [S. 88] endlos *aufschaukeln*.

Σ konvergiert also sogar für eine unendliche Zeitskala nach endlich vielen Schritten, lokal aber sind in der Regel endliche Wertebereiche gegeben. Theoretisch kann es also passieren, dass lokal keine Lösung gefunden wird. Schließlich kann der gegebene Fahrplan ja derart eng gesteckt sein und so viele Züge enthalten, dass für die gegebene Zeitskala gar keine korrekte Lösung existiert.

Wenn auf der Zeitskala eine obere Schranke existiert, dann kann diese von keinem Zug in keinem Abschnitt überschritten werden. Σ versucht aber immer, jeden Zug so weit hinauszuschieben, bis seine Fahrt global konsistent ist. Und wenn dadurch ein Zug in einem Abschnitt diese obere Schranke erreicht, muss

das Programm abbrechen, weil ja dann per Σ eben keine Lösung gefunden werden kann. Dieser Fall ist aber nicht inhärent in Σ enthalten sondern muss von der Implementierung als Spezialfall behandelt werden.

Der angegebene Termierungsbeweis bedeutet aber für die Praxis, dass wenn die Zeitskala nur *genügend groß ist*, Σ auch immer eine korrekte Lösung findet! Insbesondere werden Deadlocks vermieden, was eine der Anforderungen an den Algorithmus war. Und tatsächlich kann man eine solche genügend große Zeitskala in der Praxis in aller Regel auch finden, wie unsere Tests (siehe Kapitel 25, S. 160) zeigen.

16 Zusammenfassung und Diskussion

In diesem Teil wurde der verteilte Algorithmus DRS beschrieben und anhand der formalen Konstruktionen Π, Solution, Σ_p und Σ gezeigt, dass jede verteilte Simulation Σ für jedes Simulationsproblem Π nach endlich vielen Schritten eine korrekte Simulation berechnet. Dies gilt für den verteilten Algorithmus sogar für unendliche Zeitskalen. Weil lokal typischerweise endliche Wertebereiche gegeben sind, berechnet Σ dann eine korrekte Lösung, wenn die Zeitskala nur genügend groß ist, was in der Praxis in der Regel der Fall ist. Dabei kann der lokale Simulationsalgorithmus relativ beliebig funktionieren, er muss nur wenige spezielle Anforderungen erfüllen. In diesem Kapitel sollen noch einige weitere Aspekte des verteilten Algorithmus diskutiert werden.

16.1 Zug-Wiedereintritt

Wie ich weiter vorne (in Bemerkung 8.2, S. 55) schon angedeutet hatte, erlauben obige Definitionen nicht, dass ein Zug einen Simulations-Teil mehr als einmal betritt. Ich habe diese Einschränkung hier vorgenommen, damit vor allem die formale Darstellung nicht noch komplexer wird. Der verteilte Algorithmus und die lokale Simulation kann prinzipiell auch mit Konstellationen umgehen, in denen Züge Simulationsteile mehrmals betreten: Seien zwei Teile p und p' gegeben und ein Zug t, der von p nach p' und dann wieder zurück nach p fährt. Aus der Sicht von p' gibt es kein Problem, weil t diesen Teil ja genau einmal betritt und einmal verlässt. In p' kann t so behandelt werden, als wäre der frühere Abschnitt (vor der Ausfahrt nach p') ein anderer Zug als der im späteren Abschnitt. Man kann also die beiden Fahrten von t in p als Fahrten unterschiedlicher Züge betrachten. Offensichtlich kann man damit den allgemeineren Fall behandeln, ohne dass die Eigenschaften des spezielleren Falls (ohne die Wiedereintrittsmöglichkeit) bezüglich Korrektheit und Terminierung berührt werden.

16.2 Nicht- / Determiniertheit

Wie ich in Korollar 10.6 [S. 68] festgestellt habe, ist die lokale Simulation Σ_p deterministisch. Die verteilte Ausführung Σ aber ist wegen der nicht-deterministischen Ausführungsreihenfolge Sched tatsächlich nicht-determiniert. Zu einem gegebenen Simulationsproblem Π können also unterschiedliche Abläufe Σ stattfinden, die auch zu unterschiedlichen Endergebnissen führen können. Das ist einerseits eine schöne Eigenschaft, weil sie dem Algorithmus Freiheiten lässt.

Andererseits aber möchte der Anwender gerne (so hat er im Projekt SIMONE geäußert), dass Simulationen *wiederholbar* sein müssen, dass also ein und dieselbe Konfiguration in unterschiedlichen Simulationsläufen dasselbe Simulationsergebnis erzeugt. Eine solche Forderung ist in verteilten Systemen oft schwer zu realisieren.

In DRS kann diese Anforderung aber relativ einfach erfüllt werden: Die Ausführung der Iterationsschritte wird synchronisiert, so dass in jedem globalen Schritt alle lokalen Simulationen ausgeführt werden, schließlich alle gleichzeitig kommunizieren, und erst dann alle gleichzeitig mit der nächsten Simulationsrunde beginnen. Offensichtlich garantiert das Determiniertheit: Beginnend beim ersten Schritt $i = 0$ haben alle lokalen Simulatoren in jedem möglichen

Lauf die gleiche Anfangsinformation, bestehend aus den Verschiebungen, die die Nachbarn geschickt haben. Und weil die lokalen Simulationen deterministisch sind, berechnen sie aus diesen Informationen in jedem möglichen Lauf dasselbe Simulationsergebnis, was wiederum immer zur jeweils gleichen Menge an Änderungen führt, die im nächsten Schritt $i + 1$ von den immer gleichen Nachbarn berücksichtigt werden müssen.

Das synchronisierte Verfahren ist also von keiner konkreten Ausführungsreihenfolge mehr abhängig, es ist damit sogar deterministisch.

Unsere Implementierung DRS realisiert sowohl den nicht-determinierten Fall als auch den synchronisierten, determinierten: siehe Abschnitt 20.4 [S. 139].

16.3 Problem-Zerlegung

Man könnte versucht sein, das gegebene Simulationsproblem nicht-disjunkt zu zerlegen, so dass also die Problem-Zerlegung nicht wie oben definiert eine Partition ist. Die Teile würden sich dann überlappen. Das wichtigste Argument dafür wäre, dass man durch die Überlappung lokal in den Simulationsteilen mehr Information zur Verfügung hat, die in der lokalen Simulation zum Beispiel durch Propagation zu lokal *besseren* Simulationsergebnissen führen würde.

Diese Idee lässt sich zwar mit dem gegebenen Verfahren behandeln, die Schnittstelle zwischen benachbarten Teilen würde komplizierter, aber es spricht formal nichts gegen eine Verallgemeinerung obiger Methoden auf solche Problemzerlegungen.

Allerdings ist dieser Ansatz praktisch nicht viel versprechend:

- Wie gesagt würde die Schnittstelle zwischen den Simulationsteilen deutlich komplexer (anstatt einzelner Zeiten an den Grenzen zwischen den Teilen müssten alle Belegungszeiten in allen Gleisabschnitten der überlappenden Teile konsistent gemacht werden),
- dadurch würde der Kommunikationsaufwand erheblich steigen,
- die überlappenden Teile würden mehrfach berechnet,
- mit der Information, die man lokal zusätzlich zur Verfügung hat (in einem Teil, der jetzt zusätzliche Informationen aus überlappenden Abschnitten eines Nachbarteils besitzt) kann man wenig anfangen, weil diese ja zwischen den überlappenden Teilen inkonsistent sein kann.

16.4 Züge in die Vergangenheit verschieben

Eine andere häufig geäußerte Idee ist, die Züge beim verteilten Algorithmus nicht immer in die Zukunft zu verschieben, sondern einzelne Züge auch mal in die Vergangenheit. Wenn ein Zug also global inkonsistent ist, sollte er dann an einer Grenze vom globalen Algorithmus Σ nicht automatisch auf den spätest möglichen Zeitpunkt verschoben werden, sondern auf einen früheren. Damit könnte man versuchen, inkonsistente Zeiten des Zuges an der Grenze einander anzunähern.

Dieses Vorgehen ist aber auch problematisch: Σ_p legt ja jeden Zug in jedem lokalen Teil zum frühest möglichen Zeitpunkt ein. So habe ich das oben definiert und so wird das auch bei allen Eisenbahn-Simulationen gemacht, schließlich ist

der frühestmögliche Zeitpunkt ja immer der, der am nächsten am Fahrplan liegt. Und weil nun eben jeder Zug lokal schon zum jeweils frühest möglichen Zeitpunkt fährt, ist es in aller Regel nicht möglich, einen Zug global, also an einer Grenze zwischen zwei Teilen, früher als in einem der beiden Teile vorgesehen, fahren zu lassen.

Der *frühestmögliche* Zeitpunkt ist bei der Fahrplansimulation also der, der dem Fahrplan am nächsten liegt. Da wirkt der Fahrplan ja als hartes Constraint, da *kein Zug vor* der planmäßigen Zeit abfahren darf. Das DRS-Verfahren lässt sich außer zur Simulation prinzipiell aber auch zur Konstruktion von Fahrplänen einsetzen. Und in diesem Fall ist der Fahrplan nicht mehr ein hartes Constraint. In diesem Fall könnte man also Züge global auch in die Vergangenheit verschieben.

16.5 Lokale Optimierung

Das Prinzip, Züge nur in die Vergangenheit zu verschieben, kann zu schlechten Simulationsergebnissen führen. *Schlechte Ergebnisse* sind hier solche, bei den viele Züge große Verspätungen aufweisen. Das wesentliche Bestreben der Bahn im Betrieb ist es ja, Verspätungen zu minimieren.

So kann es während einer Simulation passieren, dass ein Zug t einen Zug t' durch seine höhere Priorität mehrmals verdrängt, bis die Simulation schließlich für t eine konsistente Fahrt gefunden hat. Jetzt aber kann t schon so weit in die Zukunft verschoben worden sein, dass t' jetzt wieder zeitlich vor t Platz hätte. Das momentane Verfahren verhindert nun aber, dass t' auch wirklich früher fahren darf.

Es stellt sich damit also die Frage nach lokalen Optimierungsmöglichkeiten. Lokal könnte man beispielsweise bei der Simulation eines Zuges mit hoher Priorität Rücksicht auf andere Züge mit niedrigerer Priorität nehmen. Das aber würde implizit die für die Terminierung wichtige globale Zugreihenfolge aushebeln.

Man könnte aber versuchen, das ganze System quasi *kontrolliert abzukühlen*: man könnte eine *Zugnummerngrenze* z während der Simulation immer weiter erhöhen und lokal immer solche Züge, deren Nummer unterhalb der Grenze z liegen nicht mehr verschieben (die müssen dann natürlich schon alle konsistent sein), die Züge jenseits von z aber auch in die Vergangenheit verschieben. Im Ergebnis hätte man sicher mehr Rechenaufwand, möglicherweise aber zumindest für einige Züge ein besseres Simulationsergebnis.

Generell gilt aber bei diesem wie bei allen realen Optimierungsproblemen, dass man auf vollständige Suche verzichten muss, weil wegen der NP-Vollständigkeit der Probleme nie in vernünftiger Zeit alle Lösungen getestet werden können.

16.6 Zugreihenfolge

Die strikte Einhaltung der globalen Zugreihenfolge birgt Probleme, die sich auf die Qualität des Simulationsergebnisses auswirken können. Man kann diese strikte Einhaltung bzw. generell die globale Zugreihenfolge aber auch fallen lassen. Der Algorithmus terminiert dann nicht mehr automatisch. Man könnte Terminierung aber auch durch spezielle ad-hoc-Verfahren erreichen: So könnte man versuchen, das gegenseitige Aufschaukeln aus Bemerkung 14.3 [S. 88] gezielt zu

erkennen und dann lokal durch eine lokale Reihenfolge zwischen den problematischen Zügen das Aufschaukeln zu verhindern.

Auch könnte man einfach Iterationsgrenzen einführen und die Iterationen nach Überschreiten der Grenzen abbrechen. Dieses Verfahren würde aber nicht mehr in jedem Fall eine Lösung finden, es wäre nicht mehr in obigem Sinne Deadlock-frei.

16.7 Constraints und DRS

Wie ich in Bemerkung 10.3 [S. 67] schon angedeutet habe, kann man die lokale Simulation Σ_p besonders einfach unter Verwendung von Constraint-Propagation darstellen. Prinzipiell kann man lokal aber auch andere Simulations-Verfahren verwenden. Allerdings muss die lokale Simulation immer sicherstellen, dass die von außen (den Nachbarn) hinzugefügten Zusatzbedingungen (wie: Zug t muss den Teil p später als 10:00 verlassen) eingehalten werden. Und so etwas kann bei anderen Simulations-Ansätzen, die eben nicht vorausschauend arbeiten (Abschnitt 4.1.4, S. 40), schwierig umzusetzen sein. Insofern gehören DRS und Constraint-basierte Simulation sehr eng zusammen.

Teil III

Realisierung

DRS ist nicht nur ein formales Konzept sondern bereits umfassend realisiert. Der folgende Teil beschreibt diese Umsetzung.

Zunächst werden auf Basis der einführenden Bemerkungen (Kapitel 2, S. 6) in Kapitel 17 [S. 98] Anforderungen an die Implementierung von DRS definiert. Dann werden entsprechend des Standards UML (*Unified Modeling Language*, [BRJ99] [RJB99]) System-Design (Kapitel 18, S. 101) und interne Abläufe (Kapitel 19, S. 116) erläutert. Kapitel 20 [S. 130] widmet sich ausführlich der Kontrolle des verteilten Algorithmus (inkl. Beweis der Korrektheit der Implementierung). Wie Simulationsprobleme zerlegt und auf Rechenknoten verteilt werden, erläutert Kapitel 21 [S. 141]. Im Projekt SIMONE wurde ein Simulator entwickelt, der in das DRS als externes Modul eingebunden werden kann: siehe Kapitel 22 [S. 146]. Wir haben aus verschiedenen Gründen einen Entwicklungsprozess in das System integriert, Kapitel 23 [S. 149] zeigt wie und warum. Kapitel 24 [S. 155] schließlich fasst die Realisierung zusammen und diskutiert deren Eigenschaften anhand der eingangs definierten Anforderungen.

17 Anforderungen

Bereits in der Einführung in Kapitel 2 [S. 6] wurden einige Motivationen für verteilte Systeme beschrieben und Probleme, die bei der Umsetzung zu lösen sind. Im Folgenden werden anhand der Kriterien Informationsverteilung, Stabilität, Erweiterbarkeit und Heterogenität die wesentlichen Design-Entscheidungen motiviert. Schließlich wird UML-konform gezeigt, welche Anwendungsfälle das System unterstützen muss.

Grundsätzliche Motivationen zum Bau des Systems DRS sind: Genaues Verständnis von Eisenbahn-Systemen (Fahrplan, Infrastruktur, Rollmaterial) durch Simulation (Kapitel 4, S. 38) und schnelle Berechnung von großen Simulationen. Letztere wird zum einen durch die Verwendung von Constraint-Programmierung (Kapitel 3, S. 21) realisiert und zum anderen durch die Verteilung und Parallelisierung der Berechnung.

In Kapitel 2 [S. 6] haben wir gesehen, dass es viele Motivationen für Verteilte Systeme gibt, die allerdings im einzelnen realisiert werden müssen, und Probleme, die gelöst werden müssen. In der Summe sind das über zwanzig verschiedene Aspekte. Weil das so viele sind, orientiere ich mich hier in der Motivation des Designs von DRS nicht direkt an diesen, sondern an einigen wenigen übergeordneten. Auf die einzelnen Aspekte der Verteilten Systeme wird später im Kapitel 24 [S. 155] im Detail eingegangen.

Ich habe versucht, mit dem Design von DRS vor allem folgende Ziele zu erreichen: optimale Informationsverteilung, Stabilität aller Komponenten, Erweiterbarkeit des Gesamtsystems und Heterogenität. Außerdem haben sich die intendierten Anwendungsfälle auf das Design ausgewirkt.

Grundsätzlich muss DRS hoch-performant sein. Zum einen war das die Ausgangsforderung und -Motivation für die *verteilte Berechnung* und den DRS-Algorithmus. Insofern schlägt sich diese Anforderung sehr grundlegend im Design nieder. Außerdem haben wir bei der Implementierung aller Komponenten auf optimale Laufzeiteigenschaften geachtet.

17.1 Informationsverteilung

Wie bereits in Abschnitt 2.2 [S. 9] festgestellt, müssen in verteilten Systemen viele verschiedene Informationen unterschiedlicher Gültigkeiten verteilt werden. In DRS wird das durch einen zentralen Informationsdienst *DIR* (Abschnitt 18.2, S. 103) realisiert, der für statische Konfigurationen als auch für den dynamischen globalen Systemzustand zuständig ist. Er zum Beispiel weiß, welche Rechenknoten gerade im System vorhanden sind und welche Anwender das System benutzen.

In DRS gibt es viele verschiedene Komponenten, die jeweils für die verteilte Umgebung und konkrete Berechnungen konfiguriert werden müssen. Die Konfiguration aller Komponenten des Systems wird durch *Setting*-Objekte (Abschnitt 18.6, S. 112) realisiert, die ihre Werte aus Einstellungsdateien, von der Kommandozeile beim Programm-Aufruf, aber auch aus Grafischen Benutzeroberflächen erhalten.

Für die Kommunikation zwischen den Komponenten von DRS benutzen wir den Mechanismus *Java-RMI*, siehe auch Abschnitt 2.3.3 [S. 16] und Kapitel 19 [S. 116].

17.2 Stabilität

Selbst im *Prototypen* eines verteilten Systems ist Stabilität besonders wichtig: Während bei einem monolithischen System, das nur lokal läuft, Bedienungs- oder System-Fehler lokal behoben werden können – im Extremfall durch Beenden und Neustarten –, darf ein verteiltes System nicht durch jeden Fehler abstürzen oder hängen bleiben. Es ist dann nämlich viel aufwändiger wieder in einen konsistenten Gesamtzustand zu bringen als das lokale System.

DRS muss also Bedienungsfehler tolerieren. Das wird erreicht, indem gestartete Simulationen schnell und sicher vom Benutzer abgebrochen werden können. Siehe hierzu Abschnitt 19.8 [S. 126].

DRS aber ist robust selbst gegenüber Programmierfehlern im eigenen System: Die Rechenkomponenten (Abschnitt 18.3, S. 103) sind eingebettet in ein existierendes stabiles System, das selbst dann korrekt weiterarbeitet und den korrekten Gesamtzustand des Systems sichert, wenn einzelne Komponenten von DRS nicht korrekt arbeiten. Fast alle Komponenten können sogar während der Laufzeit des Gesamtsystems ausgetauscht werden, siehe nächster Abschnitt.

17.3 Erweiterbarkeit

DRS ist ein Prototyp, der sich in permanenter Entwicklung befindet. Jede Weiter-Entwicklung muss im verteilten System getestet werden. Dafür haben wir eine *Online-Update*-Möglichkeit geschaffen, mit der der Entwickler von seinem Arbeitsplatz aus per standardisierten Prozessen die meisten Komponenten des Laufzeit-Systems erneuern und durch verbesserte ersetzen kann. Wie das technisch funktioniert, ist in Abschnitt 23.5 [S. 152] genau beschrieben.

Gleichzeitig muss ein laufendes Gesamtsystem um neue Rechenknoten oder Anwender erweitert werden können, ohne das System neu starten zu müssen. Genauso müssen im System registrierte Rechner und Anwender entfernt werden können. Siehe hierzu Abschnitt 18.2 [S. 103] und Abschnitt 18.3 [S. 103].

17.4 Heterogenität

Ein Ziel von DRS ist es, vorhandene Rechenkapazität im Stile eines Grid-Systems zu nutzen (Abschnitt 2.3.2, S. 14). Weil die vorhandene Kapazität in unserem Fall und im Allgemeinen heterogen ist, muss DRS System-unabhängig sein. Es muss beispielsweise sowohl unter MS Windows als auch unter UNIX laufen. Zum einen erreichen wir das durch die Verwendung der Java-Plattform [Jav], die selbst auf vielen unterschiedlichen Systemen läuft und diesbezüglich quasi eine abstrakte, System-unabhängige virtuelle Maschine zur Verfügung stellt. Außerdem bietet Java eine Vielzahl moderner Dienste standardmäßig an, wie Kommunikation zwischen verteilten Systemen, Laufzeitumgebung inklusive Garbage-Collector und dynamischem Class-Loader, Datenbank-Schnittstellen, parallele Verarbeitung und Thread-Unterstützung, Grafische Benutzerschnittstellen, Web-Dienste usw. Weil viele Java-Werkzeuge kostenlos verfügbar sind und trotzdem qualitativ hochwertig, hat Java in der letzten Zeit gerade im Forschungs-Kontext einige Bedeutung erlangt. Alles das macht Java zur idealen Plattform für DRS.

Eine große technische Herausforderung ist folgende Anforderung die sich aus der Tatsache ergibt, dass DRS im Rahmen eines Forschungs- und Entwicklungs-

projekts entstanden ist: DRS muss einen System-externen lokalen Simulator einbinden können. Dieser ist als CLP-Anwendung in CHIP [CHI01] realisiert. Wir haben dafür in DRS eine Simulator-Abstraktion (Abschnitt 18.5, S. 108) geschaffen, die es erlaubt, verschiedene Simulatoren einzubinden und zu verwenden (Abschnitt 18.5, S. 108, und Kapitel 22, S. 146).

17.5 Anwendungsfälle

Es gibt drei potentielle Nutzergruppen für das System DRS: Anwender (der Simulation), Administrator und Entwickler. Letzterer ist vor allem deshalb ein gleichberechtigter Nutzer, weil das System DRS noch ein Prototyp ist und sich permanent entwickelt. DRS verfügt deshalb über einen integrierten Entwicklungsprozess (Kapitel 23, S. 149).

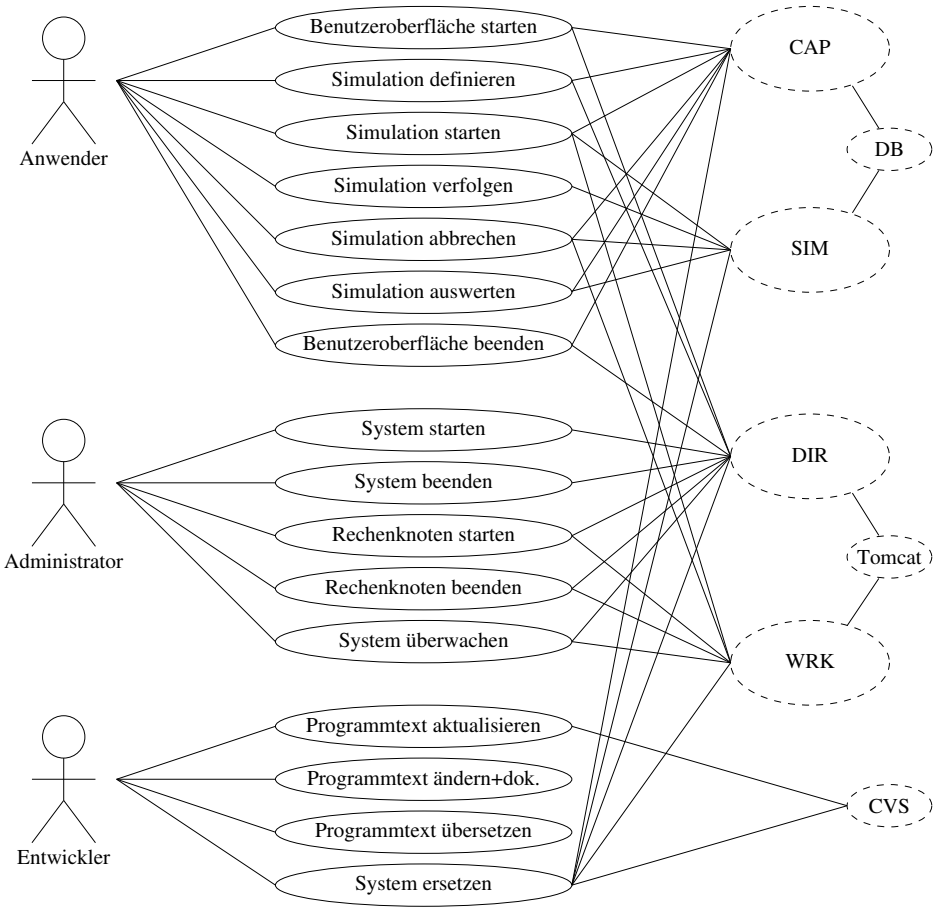


Abbildung 17.1: Nutzer, Anwendungsfälle und beteiligte Komponenten

Abbildung 17.1 [S. 100] zeigt die Anwendungsfälle als *Use Case Diagram* [BRJ99, 17]. Hier sind auch die Beziehungen zu den Anwendern auf der einen und den DRS-Komponenten auf der anderen Seite dargestellt.

18 Design

In diesem Kapitel wird das Design des DRS-Systems beschrieben: zunächst die globale Architektur und dann, nach einer kurzen Einführung in die hier benutzten UML-Konzepte, die wichtigsten Komponenten DIR, WRK, CAP, SIM, System-Konfiguration, Datenbank, die Laufzeitumgebung Tomcat und das Versions-Management per CVS.

18.1 Globale Sicht

18.1.1 Architektur

Die Architektur des Systems DRS möchte ich zunächst anhand der beteiligten Komponenten und deren Verteilung auf einzelne Rechner darstellen. UML sieht dafür *Deployment Diagramme* vor [BRJ99, 30]: siehe Abbildung 18.1 [S. 101].

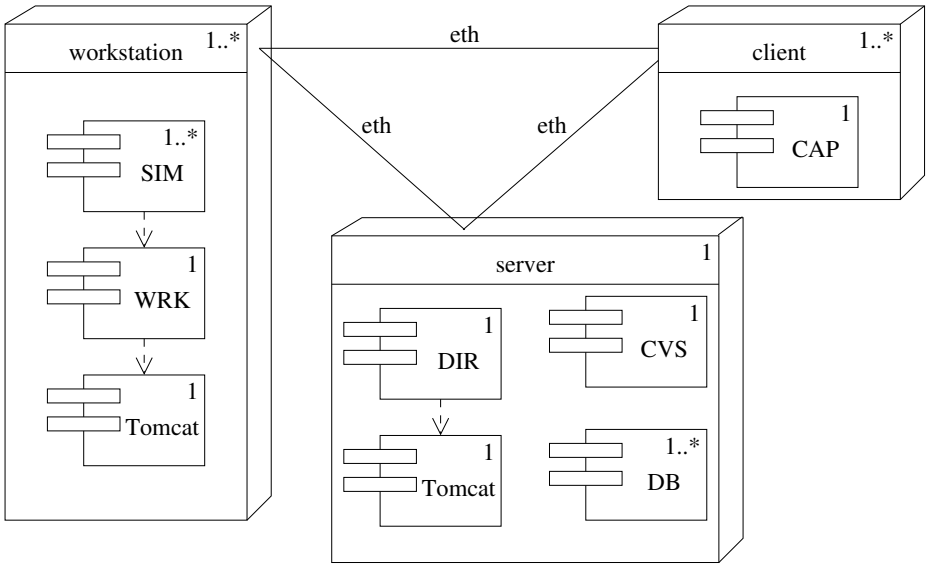


Abbildung 18.1: *Deployment-Diagramm*: Verteilung der Komponenten auf Rechner.

Für die Informationszentrale ist ein Server-Rechner vorgesehen, auf dem die Komponenten DIR (siehe Abschnitt 18.2, S. 103), CVS (Abschnitt 18.9, S. 115) und die Datenbank DB (Abschnitt 18.7, S. 113) laufen. Der *Directory-Service* DIR ist die Zentrale eines laufenden DRS-Systems. Am DIR melden sich alle anderen Komponenten an und melden ihm wichtige Zustandsänderungen. Dieser Dienst läuft in einem Tomcat-Webserver (Abschnitt 18.8, S. 114), der u.a. eine stabile Laufzeitumgebung herstellt. Das *Concurrent Versions System* CVS dient der Versionierung des Programm-Sourcecodes und wird für den integrierten Entwicklungsprozess (Kapitel 23, S. 149) benötigt. In der Datenbank DB

sind Simulations-Spezifikationen gespeichert und dorthin werden die Simulationsergebnisse geschrieben. Diese zentralen Dienste müssen nicht wirklich auf einem gemeinsamen Rechner liegen, alle müssen aber auf einem hoch-verfügbaren Server laufen. In einer laufenden DRS-Instanz gibt es genau einen DIR-Service, eingebettet in genau einen Tomcat-Server, genau ein CVS und typischerweise eine Datenbank.

Die Simulationsberechnungen werden auf Workstations ausgeführt, in der Regel viel mehr als einer. Auf jeder solchen Station läuft genau ein *Worker* WRK (Abschnitt 18.3, S. 103), wiederum eingebettet in einen Tomcat, und für eine konkrete Simulation typischerweise mehrere *Simulatoren* SIM (Abschnitt 18.5, S. 108). Workstations sollten schnelle Arbeitsplatzrechner sein. Diese brauchen nicht so hochgradig verfügbar zu sein wie der Server. Arbeitsplatzrechner können etwa auch ausgeschaltet sein. Ein DRS-System verträgt es, wenn WRK-Komponenten an- und abgemeldet werden.

Der Anwender benutzt das System über die *Control-Application* CAP (Abschnitt 18.4, S. 107). Natürlich können gleichzeitig mehrere CAP an einem gemeinsamen DRS-System teilnehmen, aber nur einer pro Client-Rechner. Die Client-Rechner können kleinere Computer sein und müssen generell nur für die Dauer einer von dort gesteuerten Simulation laufen.

Die Rechenknoten sind untereinander über Standard-Netzwerke verbunden, üblicherweise Ethernet. Theoretisch können sie damit örtlich sehr weit auseinander liegen und zum Beispiel über Internet verbunden sein. Eine solche Konfiguration wirkt sich aber auf das Laufzeitverhalten aus. Übrigens kann auch das gesamte System auf einem einzigen Rechner laufen!

18.1.2 Abstraktionen

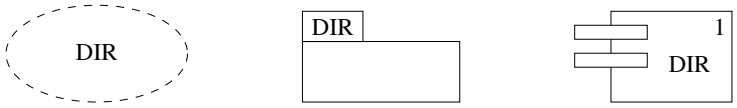


Abbildung 18.2: Unterschiedliche Abstraktionen in UML-Diagrammen.

UML sieht gerade für die Komponenten eines Systems unterschiedliche Sichtweisen vor, die sich in verschiedenen Symbolen niederschlagen. Abbildung 18.2 [S. 102] zeigt diese zusammen mit den UML-Diagramm-Typen. Die Unterschiede sind im Wesentlichen:

Collaboration Sehr abstrakte Zusammenfassung zusammengehörender Einheiten in einem *Use-Case-Diagram*.

Package Zusammengehörende Einheiten in der Implementierung.

Component Die Programme eines *Package* zur Laufzeit.

18.1.3 Klassen

Sehr viel detaillierter als im Deployment-Diagramm können in Klassendiagrammen (*Class-Diagram*, [BRJ99, 8]) die (Unter-) Einheiten der Komponenten und deren Funktionalität (Attribute, Methoden und zum Teil verschachtelt *Innere Klassen*) dargestellt werden: Abbildung 18.3 [S. 104] zeigt im Überblick die wichtigsten. Die folgenden Abschnitte beschreiben diese Klassen detailliert.

18.2 Directory-Service DIR

Der *Directory-Service*, realisiert in der Komponente DIR, ist die Informationszentrale jedes DRS-Systems. Es ist der einzige Punkt im System, der allen anderen Programmen initial bekannt sein muss. Über ihn werden außerdem wichtige Status- und Programm-Änderungen verbreitet. Abbildung 18.4 [S. 105] zeigt das Klassendiagramm der wichtigsten Klasse `DirectoryService`.

Worker können sich hier über die Methode `register(Worker)` anmelden, und werden in einer Liste gespeichert. Per `unregister(Worker)` melden sich Worker beim Beenden entsprechend ab. Graphische Benutzeroberflächen (oder CAP-Komponenten) erfahren hier, welche Worker es gibt (`snapshot()`) und können den regelmäßigen Empfang von Status-Änderungen abonnieren (`unsubscribe(CAP)` und `subscribe(CAP)`). Außerdem kann eine CAP hier Worker reservieren: `request(CAP,Worker)`. Wie diese Dinge im einzelnen funktionieren, ist weiter unten im Kapitel 19 [S. 116] beschrieben. Der DIR weiß damit, welche Worker im System vorhandenen sind und welchen Zustand sie haben (ob reserviert oder nicht).

18.2.1 Web-Oberfläche

Der DIR kann alle seine Informationen als HTML-Text darstellen. Nachdem der DIR in einem Tomcat Webserver läuft, kann dieses dynamische Dokument zum Beispiel im Intranet – technisch ein Teil des Internet, der aber nur einem begrenzten Nutzerkreis zugänglich ist – veröffentlicht werden, sodass der Administrator per einfachem Web-Browser Zugriff auf die Status-Informationen des DIR hat. Abbildung 18.5 (1) [S. 105] zeigt, wie das konkret aussieht. Man kann hier auch sehen, dass die Seite des DIR Verweise auf Status-Seiten aller WRK (Abschnitt 18.3.1, S. 106) hat. Eine WRK-Oberfläche zeigt Abbildung 18.5 (2) [S. 105].

18.3 Worker WRK

Der *Worker* WRK ermöglicht die Berechnung einer Simulation auf einer Workstation. Abbildung 18.6 [S. 106] veranschaulicht die Fähigkeiten des WRK. Ein `Worker` ist im Status BUSY, wenn er einer `CAP` zugeordnet ist, die ihn dann für Berechnungen *exklusiv* nutzen darf, und IDLE, wenn er keiner `CAP` zugeordnet ist. Nur, wenn er IDLE ist, kann er einer `CAP` zugeordnet werden. Die Datenstruktur `sims` ordnet jeder `Simulation` des Workers ein `Simu`-Objekt zu, das die Simulation durchführt. Die Details der Abläufe sind in Kapitel 19 [S. 116] genauer ausgeführt.

Der WRK ist die Laufzeitumgebung eines Simulators (Abschnitt 18.5, S. 108) auf einem Rechner. Ein Simulator kann nur auf einem Rechner gestartet werden, auf dem schon ein WRK läuft. Das ist zum Teil der Tatsache geschuldet, dass es

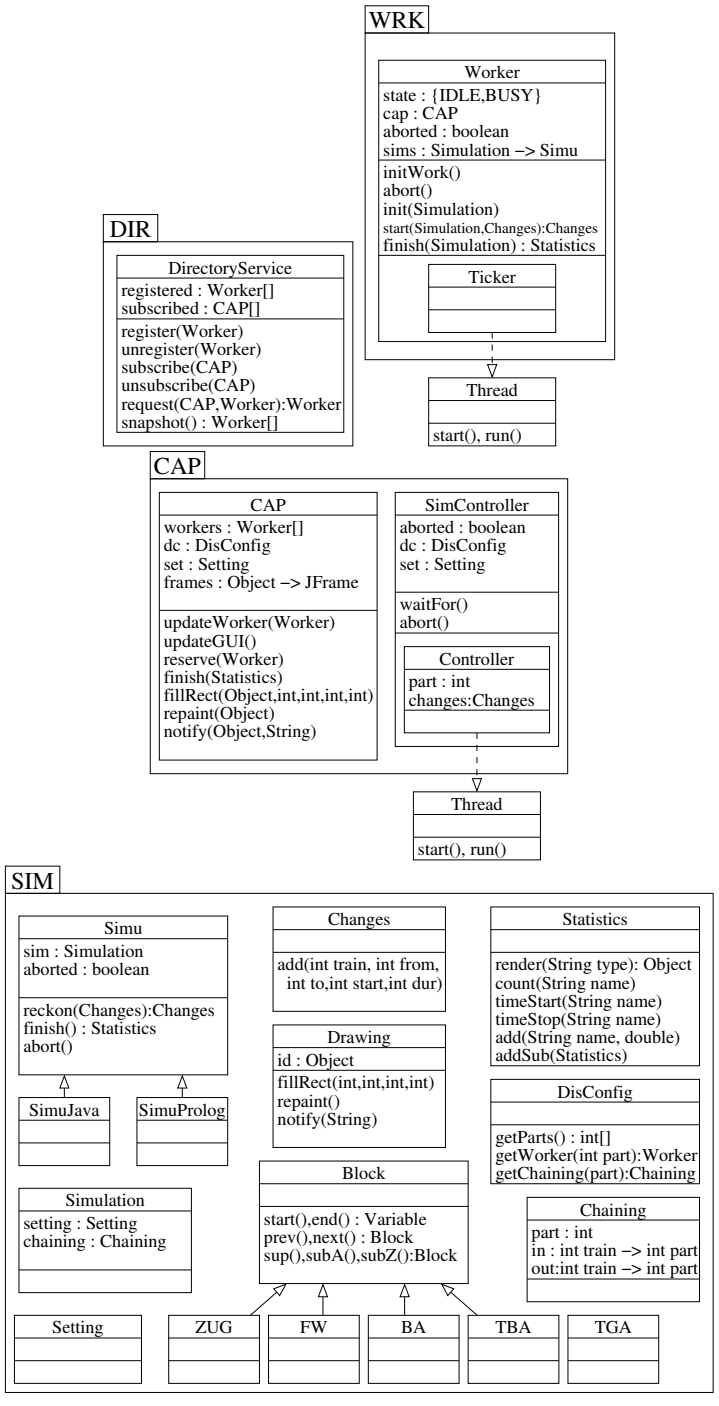


Abbildung 18.3: Die wichtigsten Klassen mit Attributen und Methoden.

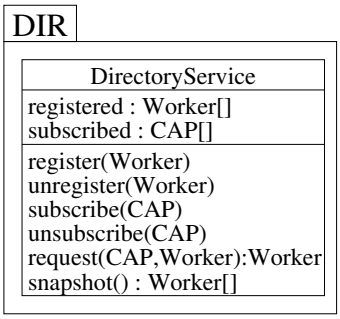
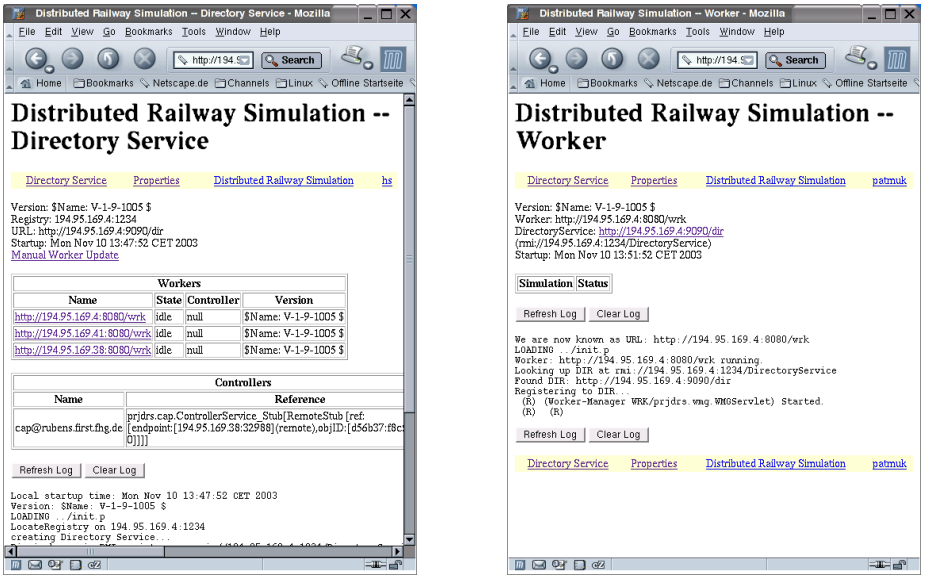


Abbildung 18.4: Klassendiagramm DIR.



(1) (2)

Abbildung 18.5: Status-Seiten im Web des DIR (1) und des WRK (2).

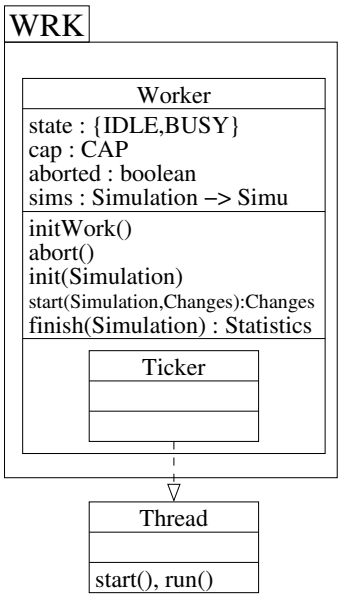


Abbildung 18.6: Klassen der Komponente WRK. `Ticker` ist ein `Thread` – *erbt von der Klasse* ***`Thread`*** – und `Thread` ist eine Standard-Java-Klasse, die quasi-parallele Verarbeitung mit einer CPU auf einem Rechner ermöglicht.

unter Microsoft Windows nicht wie unter UNIX eine eingebaute Standard-Möglichkeit gibt, automatisiert von einem Rechner auf einem anderen Prozesse zu starten. Deshalb verwenden wir eine proprietäre Komponente, eben den WRK.

Dieser muss in der Regel von Hand und lokal auf der Arbeitsstation gestartet werden. Das ist allerdings kaum ein Problem, zum einen, da man ihn nur einmal starten muss, und er lange Zeit laufen kann (für viele Simulationen und sogar über Software-Updates hinweg). Zum anderen ist das gewissermaßen eine Sicherheits-Barriere, mit der sichergestellt ist, dass nur auf ganz bestimmten Rechnern simuliert werden kann.

Wir machen aus der Not eine Tugend: nachdem der WRK auf jedem Rechner, auf dem eine konkrete Simulation durchgeführt werden soll, laufen muss, machen wir ihn zur vollständigen Dienst-Plattform für den Simulator. Er bietet alle Dienste an, die ein Simulator benötigt und die nicht ohnehin durch die Standard-Umgebung Java gegeben sind.

18.3.1 Web-Oberfläche

Wie der DIR kann auch der WRK seinen Zustand per HTML und Tomcat in einer Web-Oberfläche darstellen. Abbildung 18.5 (2) [S. 105] zeigt, wie das konkret aussieht. Die Web-Oberfläche dient übrigens auch der Anzeige von Debug-Ausgaben aus allen Programmteilen, die im WRK laufen, also vor allem des Simulators. Hierauf können dann Administrator oder Entwickler sehr einfach zugreifen.

18.4 Control-Application CAP

Die *Control-Application* CAP stellt zum einen die graphische Benutzeroberfläche für den Anwender von DRS dar, siehe Abschnitt 18.4.1 [S. 108]. Zum anderen kontrolliert die CAP über die Klasse `SimController` den verteilten Meta-Algorithmus und koordiniert damit die Arbeit der über die WRK verteilten lokalen Simulatoren. Abbildung 18.7 [S. 107] zeigt die Klassen des Pakets CAP.

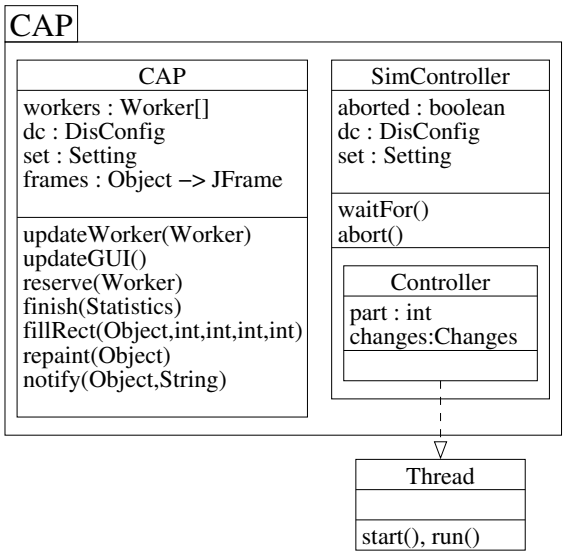


Abbildung 18.7: Die Klassen des Pakets CAP. `Controller` ist wie `Ticker` (Abbildung 18.6, S. 106) ein Thread.

Die Klasse `CAP` – nicht zu verwechseln mit dem gleichnamigen Paket – kennt alle `Worker` und weiß über die den Workern zugeordete `CAP`-Referenz, welche davon im Augenblick für sie selbst reserviert sind. Außerdem kennt sie die verteilte Konfiguration `DisConfig`, die definiert, wie ein gegebenes Simulationsproblem zerlegt ist, wo es wie berechnet wird, usw. (Abschnitt 18.5.5 [S. 110]). Das `Setting` (Abschnitt 18.6, S. 112) erlaubt die Definition beliebiger Einstellungen aus Dateien, über die Kommandozeile, oder aus der GUI heraus, und definiert damit die Simulations-Konfiguration. Der `CAP` werden Status-Änderungen von Workern gemeldet (`updateWorker(Worker)`), worauf die GUI aktualisiert werden muss. Die interne Methode `reserve(Worker)`, die aus der GUI heraus getriggert wird, veranlasst die Reservierung eines Workers beim DIR. Über die Methode `finish(Statistics)` wird der CAP mitgeteilt, dass eine Simulation beendet ist. Außerdem hat die CAP Methoden zur Status-Anzeige für entfernte Simulatoren.

Der `SimController` bekommt die Informationen zur `DisConfig` und zum `Setting` von der `CAP`. Er weiß, wenn eine Simulation dabei ist, abgebrochen zu werden, und hat Methoden, um auf die Beendigung einer Simulation zu warten und um sie auf Befehl des Anwenders vorzeitig zu beenden. Die Klasse `Controller` ist eine interne oder eingebettete Klasse vom Typ `Thread`. Weil sie

in `SimController` eingebettet ist, hat sie Zugriff auf alle Methoden und Attribute, insbesondere auf das Attribut `aborted`. Damit wirkt sich das Setzen dieses Attributs im `SimController` unmittelbar auf die `Controller` aus. `Controller` kennt den `part` des Teils, für den er zuständig ist, und die aktuellen `changes`, siehe unten.

Wie der `SimController` zusammen mit den `Controller`-Objekten die Steuerung des verteilten Algorithmus übernimmt, ist detailliert in Kapitel 20 [S. 130] beschrieben.

18.4.1 GUI

Abbildung 18.8 [S. 109] zeigt die GUI (*Graphical User Interface*), die graphische Benutzeroberfläche für den Anwender von DRS. Die GUI ist die einzige Schnittstelle des *Anwenders* zum System. Hier kann er Simulationen definieren, starten, überwachen, auswerten. Wie diese Dinge im einzelnen vor sich gehen, ist im Kapitel 19 [S. 116] beschrieben.

18.5 Simulator SIM

Das Paket SIM ist das umfangreichste Paket von DRS: siehe Abbildung 18.9 [S. 110]. Es umfasst den eigentlichen lokalen *Simulator* und viele notwendige Klassen drumherum. Die folgenden Unter-Abschnitte beschreiben die meisten davon. Für `SimuProlog` (Kapitel 22 [S. 146]) und `Setting` (Abschnitt 18.6 [S. 112]) habe ich aber eigene Abschnitte vorgesehen, weil diese Klassen komplexer sind und eingehender betrachtet werden müssen.

18.5.1 Simulation

Ein Objekt `Simulation` definiert eine lokale Simulation vollständig, und das vor allem anhand der Einstellungen `Setting` und der Verknüpfung `Chaining` der lokalen Simulation mit den Nachbarn.

18.5.2 Chaining

`Chaining` definiert die Verkettung einer lokalen Simulation mit den Nachbarn. Es kennt den Teil `part`, den die lokale Simulation berechnet, und für alle einfahrenden Züge `in` und ausfahrenden Züge `out` die Abbildung zu jeweils dem Teil, aus dem der Zug kommt bzw. in den der Zug fährt.

18.5.3 Changes

`Changes` speichert die Änderungen, die eine Simulation an den bisherigen Ergebnissen gemacht hat. Einem solchen Objekt kann der `SimController` bzw. dessen `Controller` neue Änderungen bzgl. eines Zuges, des Ausfahrts- und des Einfahrts-Abschnitts, und der neu geltenden Zeiten hinzufügen.

18.5.4 Statistics

Das `Statistics`-Objekt, das am Ende einer Simulation erzeugt wird, speichert die Statistiken der Simulation. Die Anwendung kann damit Zeiten stoppen

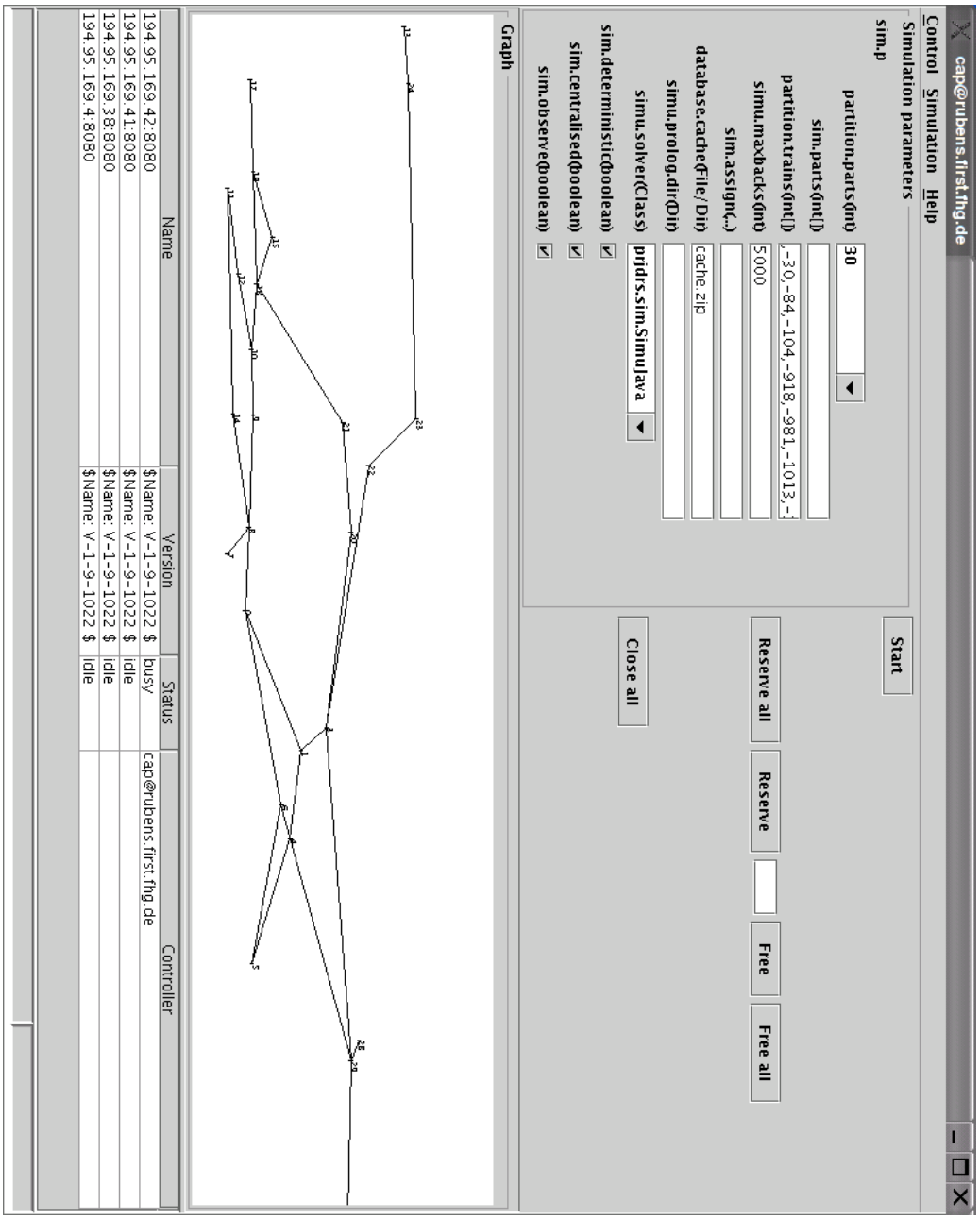


Abbildung 18.8: GUI mit Eingabefeldern für die Definition einer Simulation, verschiedenen Start-Buttons, dem aktuellen Graphen des Betriebsstellen-Netzes und der Liste der im System verfügbaren Worker.

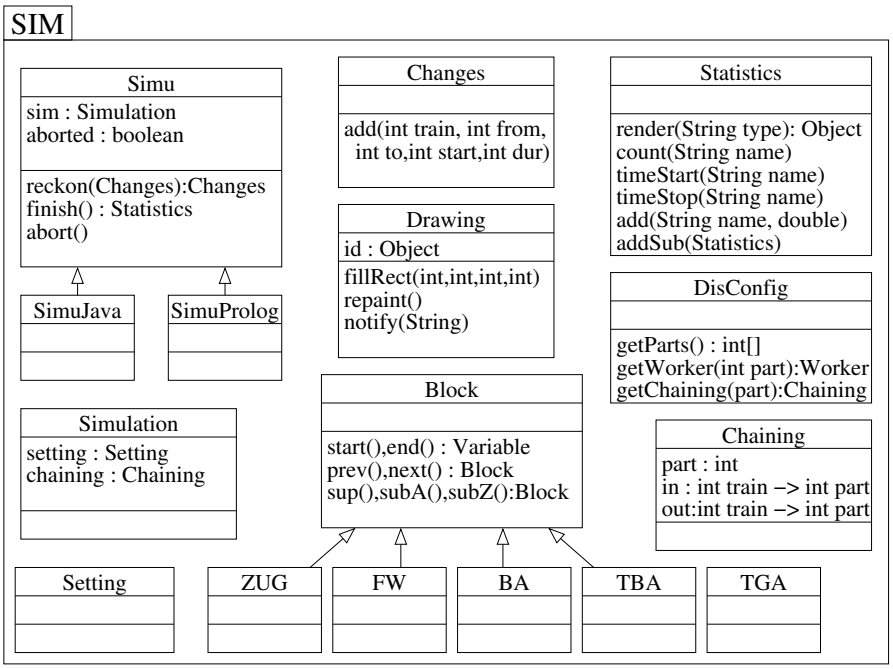


Abbildung 18.9: Klassen des Pakets SIM.

(`timeStart()` bzw. `timeStop()`), Durchläufe zählen (`count()`), Werte hinzufügen, die aufsummiert werden sollen (`add()`). Außerdem können zu einer Statistik mehrere *Unterstatistiken* hinzugefügt werden, die dann in der übergeordneten zusammengefasst werden (`addSub()`). Der Inhalt einer Statistik kann auf verschiedene Arten dargestellt werden (`render()`), zum Beispiel als ASCII-Text oder formatiert als HTML.

18.5.5 DisConfig

`DisConfig` definiert, in welche Teile ein gegebenes Simulationsproblem zerlegt ist (`getParts()`), wo die einzelnen Teile berechnet werden (`getWorker(part)`), und wie die Teile verkettet sind (`getChaining(part)`). `DisConfig` zerlegt das Problem anhand der oben skizzierten Optimierungsfunktion (Abschnitt 2.3.4, S. 17) statisch oder adaptiv. Diese Verfahren sind sehr umfangreich und deshalb in einem eigenen Kapitel genau beschrieben: Kapitel 21 [S. 141].

18.5.6 Drawing

`Drawing` ermöglicht lokalen Simulationen, ihren Zustand in der GUI der CAP darzustellen, indem sie Zeichenoperationen in ein lokales Fenster als Remote-Methoden (zum Aufruf über Rechnergrenzen hinweg) zur Verfügung stellt. Siehe hierzu auch den konkreten Ablauf in Abschnitt 19.7, S. 124.

18.5.7 Block

`Block` realisiert zusammen mit den Unterklassen `ZUG`, `FW`, `BA` und `TBA` die interne Datenstruktur des lokalen Simulators. Die Daten dafür kommen aus der Datenbank, die die folgende Struktur hat: Abbildung 18.10 [S. 111].

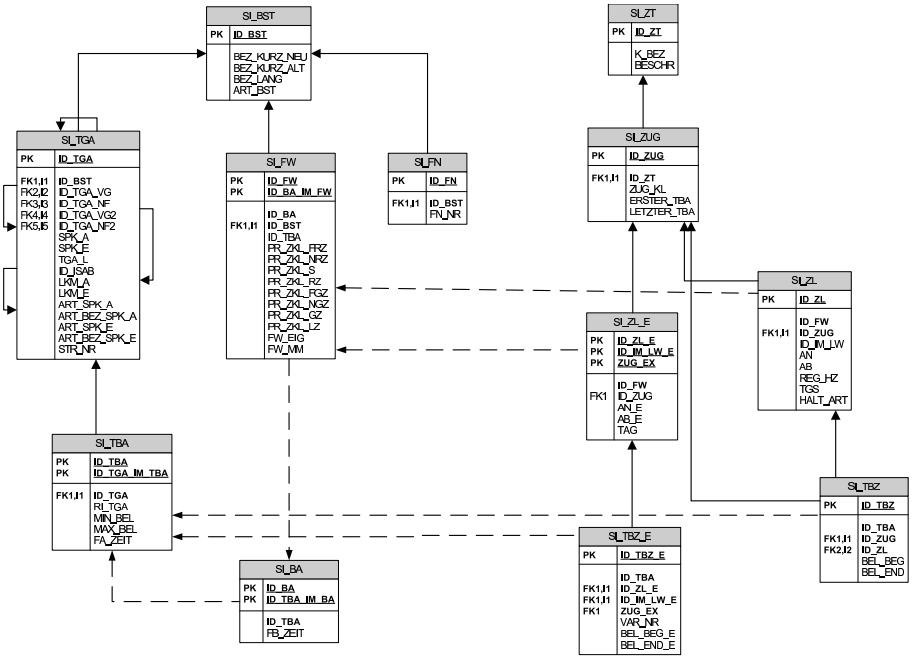


Abbildung 18.10: Struktur der SIMONE-Datenbank, aus der die interne Struktur in den `Block`-Objekten aufgebaut wird.

Jeder `Block` hat eine Constraint-`Variable` für den Start-Zeitpunkt und für den Ende-Zeitpunkt der Belegung des Blocks. Die Variablen für die Zeitpunkte werden vom lokalen Simulationsalgorithmus belegt, sie sind entsprechend der ausführlichen Beschreibung im Teil II [S. 51] verknüpft. Nicht jeder Block hat für diese Zeiten eigene Variablen, zum Beispiel gibt es Blöcke mit identischen Start-Zeiten, welche sich eine Constraint-Variable teilen. Die Methoden `start()` und `end()` reflektieren das.

Die Blöcke sind untereinander verkettet durch die Verweise (hier als Methoden dargestellt) `prev()`, `next()`, `sup()`, `subA()`, `subZ()`. Abbildung 18.11 [S. 112] zeigt, wie die Verweise zwischen den Unterklassen verwendet werden, um diese miteinander zu verketten. Jeder Block `ZUG` setzt sich aus mehreren Blöcken `FW` zusammen, diese aus mehreren `BA` und diese wiederum aus mehreren `TBA`. Außerdem sind die `TBA` mit den `TGA` verknüpft (n-m-Beziehung).

Die Klassen-Namen leiten sich übrigens aus Begriffen der Eisenbahn ab: Fahrweg, Belegungsabschnitt, Teilbelegungsabschnitt, Teilgleisabschnitt. Letztere sind die Gleisabschnitte, die exklusiv belegt werden müssen.

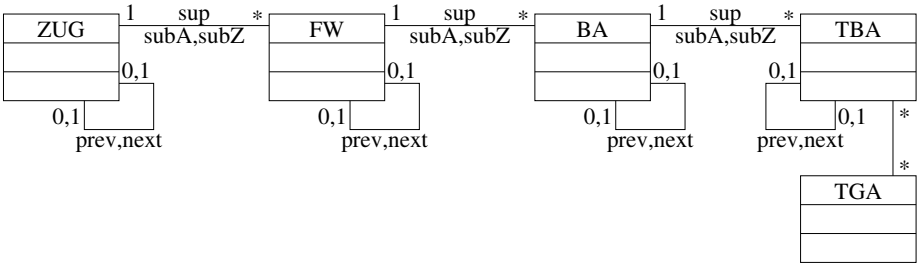


Abbildung 18.11: Verhältnis der Blöcke `ZUG`, `FW`, `BA` und `TBA` zueinander und zu den Gleisabschnitten `TGA`.

18.5.8 Simu

`Simu` ist die abstrakte Klasse für den lokalen Simulator. Sie stellt Methoden bereit für die Berechnung einer Simulation unter Berücksichtigung eventuell vorhandener Änderungen (`reckon(Changes)`), zum Beenden einer Simulation (`finish()`), und zum außerordentlichen Abbruch einer Simulation. `Simu` kennt die Definition ihrer Simulation über ein `Simulation`-Objekt.

Im augenblicklichen Prototypen existieren zwei Implementierungen für `Simu`, zwischen denen der Anwender wählen kann: `SimuJava` und `SimuProlog`. Letzterer erlaubt die Einbindung eines externen Simulators, der im laufenden Projekt SIMONE in CHIP-Prolog realisiert ist. Die Integration des externen Programms ist, wie gesagt, eine der technischen Herausforderungen dieser Arbeit. Wie das genau geht, steht in Kapitel 22 [S. 146].

`SimuJava` realisiert einen etwas rudimentären Simulator, der vollständig in Java realisiert ist und unseren neuen Constraint-Solver `firstcs` [HMSW03] benutzt. `SimuJava` implementiert exakt den lokalen Algorithmus, wie er im Theorie-Teil beschrieben ist (Kapitel 10, S. 66). `SimuJava` arbeitet dabei – anders als `SimuProlog`, das ein externes Programm ist – auf der Datenstruktur `Block`.

18.6 Konfiguration

Das DRS-System wird konfiguriert durch Einstellungsdateien, beim Aufruf eines Programms aus der Kommandozeile, oder aus der Graphischen Benutzeroberfläche der CAP. In Abbildung 18.8 [S. 109] kann man sehen, wie in der CAP die Einstellungen eingegeben werden können. Eine Datei sieht zum Beispiel wie folgt aus: siehe Abbildung 18.12 [S. 113].

Eine solche Datei besteht grundsätzlich aus Zeilen der Form `key=value`, wobei Schlüssel und Wert beliebige Zeichenketten sind. Einstellungsdateien können andere Dateien einbeziehen (`INCLUDE`). Im Wert einer Einstellung können Werte anderer Einstellungen eingebettet werden (z.B. `{sim.name}`), es gibt Makros für einige Laufzeit-Werte (z.B. `{DATE.TIME}`), die Ersetzungen können sogar verschachtelt werden (`{conf.{nodes}}`).

Beim Start eines Java-Programms können ebenfalls solche Einstellungen gesetzt werden:

```
hs@rubens:~/PRJDRS> java SimuJava @s.p -nodes=4
```

```
INCLUDE=../CAP/simulation.p, slices.p
simu.maxbacks=5000
simu.picture=false
nodes=4
conf.2=1,2
conf.4=1,2,3,4
conf={conf.{nodes}}
sim.background=#000000
sim.name=TEST
sim.subject=DRS RESULT {sim.name} {DATE.TIME} {nodes}
sim.mailto=hans.schlenker@first.fhg.de
```

Abbildung 18.12: Einstellungsdatei zur Konfiguration.

`@s.p` veranlasst, dass die Einstellungsdatei `s.p` geladen wird, `-nodes=4` setzt die Einstellung `nodes` auf den Wert 4. Hiermit wird die Einstellung aus der Datei `s.p` überschrieben und damit auch implizit die Einstellung `conf` auf den Wert von `conf.4` gesetzt.

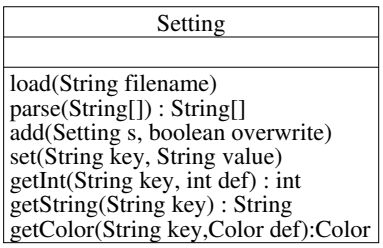


Abbildung 18.13: Die Methoden der Klasse `Setting`.

Alle diese Funktionen werden in der Klasse `Setting` bereitgestellt: siehe Abbildung 18.13 [S. 113]. Hiermit können Dateien geladen werden, die Kommandozeilen-Parameter an das Java-Programm verarbeitet werden, zu existierenden Einstellungen können neue hinzugefügt werden. Außerdem kann man aus dem Programm heraus einzelne Einstellungen vornehmen. Jede Einstellung wird mit Methoden gelesen, die diese in einen angeforderten Typ umwandelt, wie zum Beispiel `int` oder `String`, oder komplexere Objekte wie `Color`.

18.7 Datenbank DB

Das Datenbank-Modul DB ist technisch ein Standard-Datenbank-Management-System (DBMS). Die Datenbank enthält vor allem die Problemspezifikationen. Die Struktur der Datenbank ist in Abbildung 18.10 [S. 111] dargestellt. Wir benutzen für den Zugriff auf die Datenbank den Java-Standard *JDBC* [JDB]. Das ist eine generalisierte SQL-basierte Schnittstelle zu vielen Datenbank-Systemen.

Da sich die Problemspezifikation in unserem Fall sehr selten ändert und der Zugriff auf die zentrale Datenbank oft lange dauert, zum Beispiel weil mehrere

Nutzer gleichzeitig darauf zugreifen, oder sogar im Falle einer Präsentation ohne Kontakt zum Heimat-Server unmöglich ist, haben wir einen Caching-Mechanismus realisiert. Der Cache ist als Modul (JDBC-Treiber) zwischen der Anwendung und der Datenbank realisiert, über das jede Datenbank-Abfrage aus der Anwendung läuft. Die Anfrage wird eindeutig durch den Anfrage-String identifiziert (zusammen mit einer Meta-Information: dem Datenbank-Schema). Der Cache speichert die Ergebnisse von Datenbank-Anfragen, wenn sie nicht schon gecached vorliegen, in Dateien auf der lokalen Platte ab, und greift bei zukünftigen Abfragen darauf zurück. Jede Anfrage wird in einer eigenen Datei abgelegt, deren Name aus einem Hashwert des Anfrage-Strings gebildet wird. In der Datei sind die Daten nach einem Header, der Typ und Inhalt der Datei angibt, zeilenweise und spaltenweise als Text abgelegt. Der Cache kann auch alle Dateien aus dem Archiv einer einzigen Zip-Datei nehmen.

Diese Methode funktioniert sehr gut: Die Zugriff-Zeiten sind verglichen mit den Datenbank-Anfragen nicht selten um den Faktor Zehn schneller, die Benutzung ist u.a. durch die Möglichkeit der Zip-Datei sehr einfach.

Der Cache unterstützt im Moment kein *Ageing*, überprüft also nicht, ob die gecacheten Daten aktuell sind. Für unsere Zwecke ist das trotzdem im Moment ausreichend, weil sich die Grundkonfiguration (Infrastruktur und Fahrplan) selten ändert und der Anwender lokal Änderungen definieren kann (z.B. zeitliches Verschieben eines Zuges), ohne an der Datenbank etwas ändern zu müssen.

18.8 Laufzeitumgebung Tomcat

Vor allem für die Worker benötigen wir eine einheitliche Laufzeitumgebung, die von lokalen Gegebenheiten auf den Workstations, etwa dem Betriebssystem, abstrahiert und einheitlich dieselben Dienste zur Verfügung stellt. Diese muss stabil sein, sollte Remote-Management und andere Applikations-Server-Dienste implementieren.

Damit geht DRS schon sehr in Richtung *Enterprise Application* (siehe auch Abschnitt 2.3.6, S. 17). Dafür gibt es bei Java ein eigenes *Software Development Kit* (SDK) und darin noch viele weitere Pakete und Standards. Wir wollen das System aber zum einen so einfach wie möglich halten. Zum anderen wollen wir soweit möglich auf die Verwendung kommerzieller Pakete verzichten. Die Sparte *Enterprise Applications* ist aber genau der Java-Bereich, der für die kommerzielle Verwendung von Java gedacht ist und hier weidlich ausgenutzt wird. Hier also verlässt man den freien offenen Teil der Java-Welt.

Wir verwenden hier den Web-Server Tomcat [Apab], ein Apache-Open-Source Projekt. Das ist ein vollständig in Java implementierter Web-Server, der schon einige Jahre existiert und von vielen Anwendern auf der ganzen Welt genutzt wird. Tomcat ist also ausführlich getestet und entsprechend stabil.

Tomcat ist ein *Servlet Container* [SUNa], *Servlets* sind Java-Programme, die eine bestimmte Schnittstelle implementieren und damit eine Web-Applikation realisieren. Tomcat kann beliebige solche Servlets laden, starten, beenden, updaten usw. All das kann der Administrator von Hand per Fern-Verwaltung erledigen, oder aus der Ferne von anderen Programmen übernommen werden. Damit kann man per Tomcat fast beliebige Java-Programme verwalten.

Außerdem können Servlets HTML-Benutzer-Oberflächen haben, die Tomcat als Web-Server im Internet zur Verfügung stellt. All das macht Tomcat zur idealen Plattform für die DRS-Worker und den DIR.

18.9 Versions-Management CVS

Das *Concurrent Versions System* CVS [CVS] ist ein externes Programmsystem zur Versions-Kontrolle des Programmcodes.

In einem CVS gibt es ein zentrales *Repository* (englisch für *Speicher* oder *Depot*) in dem alle Versionen des Source-Codes eines Programms gespeichert werden. Außerdem hat jeder Entwickler eine eigene Kopie dieses Repositorys. Er – bzw. sie – verändert also während der Entwicklung nur seine eigene Kopie. Wenn er in dem Teil für den er zuständig ist, einen brauchbaren Zustand erreicht hat – der sich zum Beispiel kompilieren lässt oder sogar erfolgreich getestet wurde – kann der Entwickler das CVS veranlassen, seine Kopie in das Repository einzupflegen. Außerdem kann er per CVS die Neuerungen im Repository in seine Kopie übernehmen. Versionen im Repository können einheitliche Namen oder vom Nutzer vergebene Versionsnummern erhalten, wodurch konsistente Zustände gekennzeichnet werden können.

Wir benutzen ein CVS für den integrierten Entwicklungsprozess, wie er in Kapitel 23 [S. 149] beschrieben ist.

19 Abläufe

Im vorangehenden Kapitel *Design* wurden die statischen Abhängigkeiten innerhalb und zwischen den System-Komponenten beschrieben. Im folgenden Kapitel werden die dynamischen Eigenschaften erläutert: welche Ereignisse welche Abläufe auslösen und wie sich diese in den Komponenten auswirken.

Nachdem wir im letzten Kapitel die Komponenten von DRS gesehen haben, folgt nun die Beschreibung der Abläufe im System. Auch hier verwende ich wieder UML-Diagramme, in der Regel *Sequence-Diagrams* [BRJ99, 18]. Als Beispiel siehe Abbildung 19.1 [S. 116]. Hier sind nebeneinander die Objekte `c1` (vom Typ `CAP`), `d` (vom Typ `DirectoryService`) und `c2` aufgetragen. Darunter verlaufen senkrecht die *Lebenslinien* der Objekte. Die Objekte rufen gegenseitig Methoden auf (durchgezogene Pfeile), die zum Teil einen Wert zurückgeben (gestrichelter Pfeil). Die Dauer der Ausführung der Methode ist durch das schmale senkrechte Rechteck gegeben – ich mache hier keinen Unterschied zwischen synchronen und asynchronen Aufrufen. In den geschweiften Klammern stehen Zustände des Objekts (hier `d`). In den Diagrammen gebe ich vor allem die entfernten Aufrufe an, die über Rechnergrenzen hinweg ausgeführt werden. Gepunktete Linien zwischen den Objekten kennzeichnen Grenzen zwischen Rechnern.

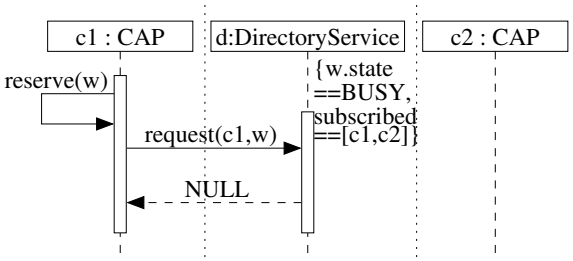


Abbildung 19.1: Beispiel eines Sequenz-Diagramms.

Viele der Abläufe haben ihren Auslöser in der GUI des Anwenders. Einige andere werden durch Programmstarts ausgelöst. Alle DRS-Programme werden vom Administrator per *Ant-Task* gestartet. *Ant* [Apaa] ist wie Tomcat ein Apache-Open-Source Projekt und vor allem für automatische Übersetzungs-Prozeduren gedacht. Man kann damit Abhängigkeiten zwischen Programmteilen definieren und festlegen, wie diese übersetzt werden sollen. Diese Mechanismen kann man aber auch sehr gut zum Konfigurieren und Starten eines Programms verwenden. Ant hat übrigens auch den Vorteil, dass es unter Windows wie unter UNIX funktioniert und jeweils einfach von einer Shell aus gestartet werden kann. Siehe auch Kapitel 23 [S. 149].

In den folgenden Abschnitten beschreibe ich jeweils zuerst, wie der jeweilige Ablauf ausgelöst wird, dann den Ablauf selbst, und schließlich etwaige Auswirkungen des Ablaufs. Zunächst aber folgt zum besseren Verständnis des Folgenden eine Einführung in den Kommunikationsmechanismus *RMI*.

19.1 Remote-Method-Invocation

Wir benutzen für die Kommunikation im verteilten System die Java-Technologie *Remote-Method-Invocation* [RMI]. Kurz gesagt, kann man damit von einem Java-Programm aus in einem beliebigen anderen Methoden aufrufen, fast so, als ob man diese Methoden im selben Programm aufrufen würde. Abbildung 19.2 [S. 117] zeigt, wie Klassen und Aufrufe in einem einfachen System zusammenhängen, das RMI benutzt.

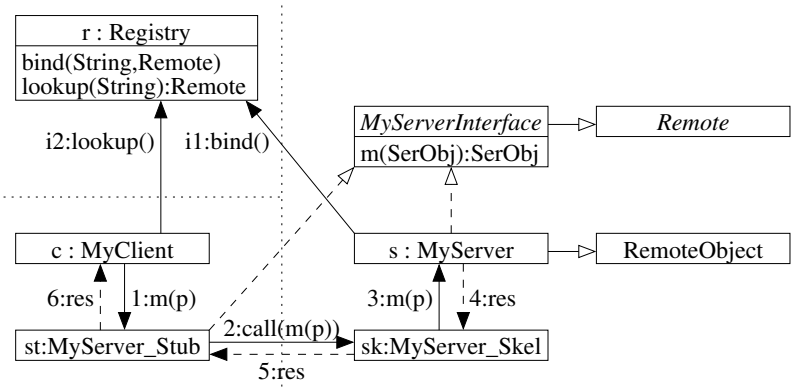


Abbildung 19.2: Klassen und Abläufe in einem einfachen System, das RMI benutzt. Die gepunkteten Linien bezeichnen Rechnergrenzen. Die unausgefüllten Pfeile beschreiben Vererbungs-Beziehungen zwischen Klassen, die ausgefüllten Pfeile Aufrufe von Methoden. Kästen, die nur mit einem Klassennamen bezeichnet sind, stehen für die Klasse (oder kursive Namen für Interfaces), die die mit `o:K` bezeichnet sind, ein konkretes Objekt `o` der Klasse `K`. Die Aufrufe sind durchnummeriert, `i1` und `i2` sind Initialisierungen, `1-6` beschreiben einen(!) Remote-Aufruf von `MyClient` nach `MyServer`.

Die Anwendung bestehe hier aus zwei Klassen, `MyClient` und `MyServer`. Der Client möchte auf dem Server eine Methode `m()` aufrufen. Dafür muss die Klasse `MyServer` ein Interface `MyServerInterface` implementieren, das die aufzurufende Methode deklariert: `m(SerObj):SerObj`. Die Methode `m()` bekommt also als Parameter ein *serialisierbares* Objekt und liefert ein solches als Ergebnis zurück. *Serialisierbar* ist ein Java-Objekt, wenn sein kompletter Zustand in irgendeiner Art geschrieben und gelesen werden kann. Serialisierbar sind die meisten Objekte, einige, wie Threads oder Streams, aber sind aufgrund flüchtiger Zustands-Information nicht serialisierbar. Serialisierbare Objekte können über verschiedene Kommunikationskanäle verschickt werden, zum Beispiel eben über RMI.

Java verlangt, dass ein Remote-Interface wie `MyServerInterface` seinerseits das Interface `Remote` erbt, und Server-Programme, auf denen Methoden aus der Ferne aufgerufen werden, die Klasse `RemoteObject` beerben. Wenn das der Fall ist, kann das Java-Programm `rmic` zu diesen Klassen eine *Skeleton*-Klasse und eine *Stub*-Klasse automatisch generieren. Die Klasse `MyServer_Stub` kümmert sich auf der Client-Seite um die Kommunikation, `MyServer_Skel` auf der Server-Seite.

Zusätzlich ist eine `Registry` erforderlich, bei der sich Server unter Angabe eines beliebigen Namens anmelden können (`bind()`) und bei der sich Clients Referenzen auf angemeldete Server holen können (`lookup`), um mit ihnen zu kommunizieren. Die Registry kann auf einem beliebigen Rechner laufen, oft läuft sie beim Anwendungsprogramm auf dem Server. Bei der Initialisierung muss sich der Server per `r.bind("myname",this)` in die Registry eintragen (Schritt `i1`), der Client holt sich per `server=r.lookup("myname")` (Schritt `i2`) die Referenz auf den Server.

Wenn nun der Client die Methode `m()` auf dem Server aufrufen will, sieht das im Programm aus wie jeder andere Methodenaufruf auch: `result=server.m(par)`. Es passiert dann folgendes: die Methode `m()` wird lokal in der Klasse `MyServer_Stub` aufgerufen (Schritt `1`; tatsächlich referenziert `server` im Client das Objekt `st`). Dort wird der Parameter `p` serialisiert, also umgewandelt, sodass er kommuniziert werden kann. `st` übermittelt dann dem Objekt `sk` beim Server, dass dieser dort die Methode `m()` mit dem Parameter `p` aufrufen soll (Schritt `2`). `sk` erzeugt nun zunächst aus der serialisierten Beschreibung von `p` ein echtes Objekt, das identisch ist zu `p` auf der Client-Seite, und ruft im Server `s` die Methode `m()` auf (Schritt `3`). Nachdem `s` die Methode vollständig abgearbeitet hat, gibt `s` das Ergebnis `res` an `sk` zurück (Schritt `4`). Jetzt serialisiert `sk` das Ergebnis und schickt es an `st` zurück (Schritt `5`), das es in ein zu `res` identisches Objekt umwandelt und an den ursprünglichen Auslöser `c` zurück gibt (Schritt `6`).

RMI ermöglicht also den Aufruf entfernter Methoden, wobei dieser synchron stattfindet: das Programm `c` *wartet*, bis das Ergebnis `res` des Aufrufs zurückkommt. Im Server muss während dessen nichts warten, der RMI-Mechanismus sorgt dafür, dass die Ausführung von `m()` in einem eigenen Thread abläuft, der parallel zum Rest des Programms `Server` ausgeführt wird.

Ich habe übrigens in Kapitel 18 [S. 101] den technischen Unterschied zwischen einem Server-Programm wie `DirectoryService` oder `Worker` und dem Remote-Interface nicht explizit dargestellt, um die Darstellung einfacher zu halten. Außerdem ermöglicht RMI nur den entfernten Aufruf von Methoden, das Client-Programm hat keinen direkten Zugriff auf *Attribute* des Server-Programms. Tatsächlich gibt es deshalb ein serialisierbares Objekt `WorkerInfo`, das im DIR erzeugt und zwischen den verteilten Komponenten kommuniziert wird. Dieses hält für den Worker unter anderem dessen `status`. Auch von dieser Feinheit habe ich oben und im folgenden aus Übersichtlichkeitsgründen abstrahiert.

19.2 System starten

Das System zu starten heißt zunächst nur, auf dem Server-Rechner den DIR zu starten, und erfolgt durch folgendes Kommando:

```
hs@server:~/PRJDRS/DIR> ant start
```

Man beachte hier, dass Ant in dem Verzeichnis gestartet wird, in dem sich das DIR-Programme befindet (`PRJDRS/DIR`).

Konkret wird dadurch ein Tomcat-Server gestartet (Abschnitt 18.8, S. 114) und in diesem die Klasse `DirectoryService` als Web-Service. DIR ist dann zunächst die einzige Komponente, es finden keine entfernten Methoden-Aufrufe statt.

Zusätzlich zum DIR muss der Administrator noch (möglichst viele) Worker – auf denen später die Simulationen berechnet werden – starten: siehe Abschnitt 19.4 [S. 119].

19.3 CAP starten

Der Anwender startet die Benutzeroberfläche, indem er auf seinem Client-Rechner die CAP startet:

```
hs@client:~/PRJDRS/CAP> ant start
```

Der folgende Ablauf ist in Abbildung 19.3 [S. 119] dargestellt.

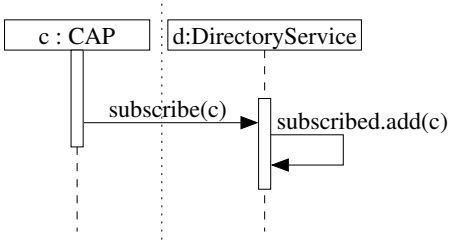


Abbildung 19.3: Sequenz-Diagramm: Start der CAP.

19.4 WRK starten

Der Administrator startet auf einer Workstation einen WRK im entsprechenden Verzeichnis wieder durch den Ant-Aufruf:

```
hs@workstation:~/PRJDRS/WRK> ant start
```

Abbildung 19.4 [S. 120] skizziert den weiteren Ablauf: Der `Worker w` erzeugt zunächst einen Thread `Ticker t`, der einmal pro Minute den `w` beim `DirectoryService d` registriert. Daher kann sogar der DIR mal ausfallen und neu gestartet werden, ohne dass alle laufenden Worker im System für immer verloren sind und man sie von Hand aufsammeln muss. Der initiale Status des WRK im DIR ist `IDLE`. Etwaig abonnierte CAPs werden darüber informiert, dass ein neuer Worker existiert (natürlich nur, wenn er wirklich neu ist). Die CAPs müssen dann ihre GUIs aktualisieren.

19.5 Worker reservieren

Der Anwender kann in der GUI einen Worker für sich reservieren, auf dem er später rechnen kann: siehe Abbildung 19.5 [S. 120].

Jetzt gibt es zwei Möglichkeiten: Entweder ist der Worker beim DIR als `BUSY` bekannt, dann ist er schon zugeordnet und kann der CAP nicht gegeben werden. Das zeigt Abbildung 19.6 [S. 120]. Der `DirectoryService d` gibt in diesem Fall der `CAP c1` den Wert `NULL` zurück, um anzuzeigen, dass die Reservierung nicht erfolgreich war. `c1` bzw. der Anwender muss dann entsprechend reagieren und etwa versuchen, einen anderen Worker zu reservieren.

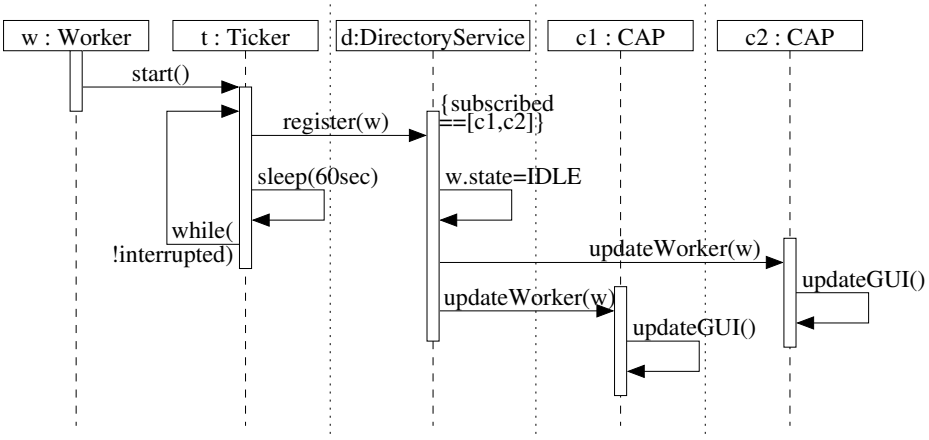


Abbildung 19.4: Sequenz-Diagramm: Start eines Workers.

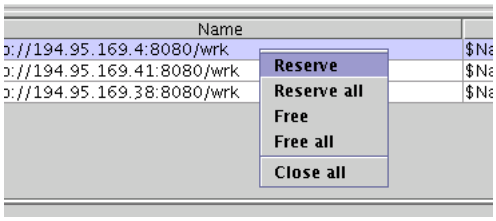


Abbildung 19.5: Ausschnitt aus der GUI: Reservieren eines bestimmten Workers.

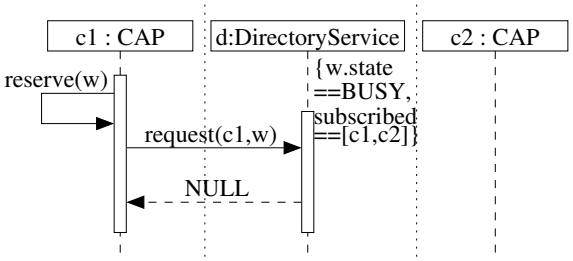


Abbildung 19.6: Sequenz-Diagramm Worker-Reservierung: Hier ist der angeforderte Worker bereits belegt.

Wenn der angeforderte Worker allerdings `IDLE` ist (Abbildung 19.7, S. 121), weist `d` dem Worker die `CAP c1` zu, setzt dessen Status auf `BUSY` und gibt `c1` den Worker `w` zurück. Weil sich der Status von `w` geändert hat, müssen die abonnierten CAPs informiert werden.

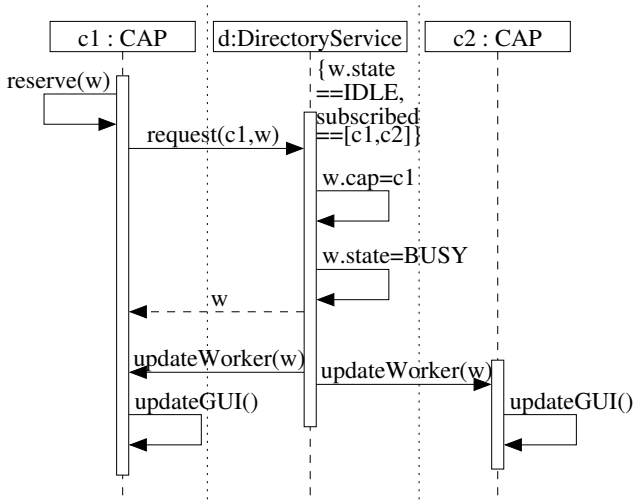


Abbildung 19.7: Sequenz-Diagramm Worker-Reservierung: Hier ist der angeforderte Worker verfügbar und wird für `c1` reserviert.

Man kann übrigens beim DIR auch *einen beliebigen* `Worker` – also nicht nur einen bestimmten `w` – anfordern. Das Freigeben von Workern geht ähnlich, nur eben umgekehrt.

19.6 Simulation

Ausgelöst und gestartet wird die Simulation in der CAP durch Drücken des START-Knopfs: Abbildung 19.8 [S. 121].

Abbildung 19.8: Der Knopf in der GUI, der die verteilte Simulation startet.

Laut Davis und Smith [DS83] verläuft *Verteilte Problemlösung in 4 Phasen: Problemzerlegung, Problemverteilung, Abarbeitung der Teilprobleme und Lösungssynthese* (siehe auch [Fri00, 2.1.4]). Genau das passiert, wenn der DRS-Anwender den Start-Knopf drückt: siehe Abbildung 19.9 [S. 123], hier für

eine Simulation mit zwei Teilen `p1,p2` und einem Worker `w1`. Die waagrechten Zickzack-Linien in der Abbildung trennen drei der vier Phasen, die Problemzerlegung wird hier nicht ausführlich dargestellt, sie findet ja lokal in `CAP` statt. Die senkrechte gepunktete Linie zeigt wieder die Grenze zwischen unterschiedlichen Rechnern.

Das Problem ist also initial zerlegt, die Zerlegung ist in der Konfiguration `DisConfig dc` gespeichert. Dann wird für diese Simulation ein `SimController sc` erzeugt, der sie zentral steuert. Der `SimController` erzeugt für jeden Teil einen `Controller` und anschließend einen `Thread`, der auf das Ende der Simulation wartet und schließlich die `CAP` darüber informiert.

Die Controller sind eigene Threads, die bei der CAP laufen, und haben die Lebensdauer ihrer `run()`-Methode. Jeder Controller initiiert beim Worker `w1` *seine* Simulation, wodurch `w1` `Simu`-Objekte anlegt, die die Berechnung durchführen. Die `Simu`-Objekte holen ihr Teilproblem aus der Datenbank DB. Das ist hier nicht dargestellt.

Man beachte hier, dass es egal ist, wie das `Simu`-Objekt konkret implementiert ist, ob es also tatsächlich Java-Simulator `SimuJava` ist, oder `SimuProlog`. Das macht das verteilte System DRS unabhängig vom konkreten lokalen Simulator.

Die entfernten Aufrufe werden technisch durch RMI realisiert, was die gleichzeitige Ausführung mehrerer Aufrufe erlaubt (siehe Abschnitt 19.1, S. 117). In dem Diagramm habe ich das veranschaulicht, indem im Worker rechts und links neben seiner Lebenslinie Ablaufrechtecke stehen. Der Worker macht eine Art Multiplexing für alle Simulationen, für die er zuständig ist. Dadurch hat der Worker alle *seine* Simulationen unter Kontrolle und kann sie zum Beispiel abbrechen (Abschnitt 19.8, S. 126).

Nachdem die `Controller` die Simulationen initiiert haben, werden diese gestartet, zunächst mit einem *leeren* `Changes`-Objekt. Solange die Berechnungen nicht-leere `Changes` zurückgeben, muss die Berechnungsschleife wiederholt werden. Wir wissen aus Teil II [S. 51], dass dieser Prozess prinzipiell irgendwann terminiert. Wie das konkret aussieht, wie also die Changes vermittelt werden und wie das Abbruch-Kriterium `necessary` aussieht, ist in Kapitel 20 [S. 130] ausgeführt.

Am Ende einer Simulation erkennen die `Controller` im `SimController` gemeinsam die globale Konsistenz und informieren den `Worker` vom Ende der Simulation. Tatsächlich geben die `finish()`-Methoden jeweils ein `Statistics`-Objekt zurück. Ich habe diese Objekte in der Abbildung nicht eingezeichnet, damit sie nicht noch komplexer wird. Die `Statistics`-Objekte werden schließlich zusammengefasst und dem Benutzer angezeigt: siehe Abbildung 19.10 [S. 124]. Im vorliegenden Prototypen dient dies der Auswertung der gesamten Simulation.

Die `Controller`-Threads sind beendet, sobald ihre jeweilige Simulation beendet ist. Das führt wiederum zur Beendigung der Methode `waitFor()`. Das allerdings nicht durch Methoden-Aufruf, sondern durch anderweitige implizite Benachrichtigung mittels Verwendung gemeinsamer Variablen. Deshalb habe ich in dem Diagramm hier keinen Methoden-Pfeil benutzt, sondern den UML-Standard-Pfeil für *asynchrone Benachrichtigung*.

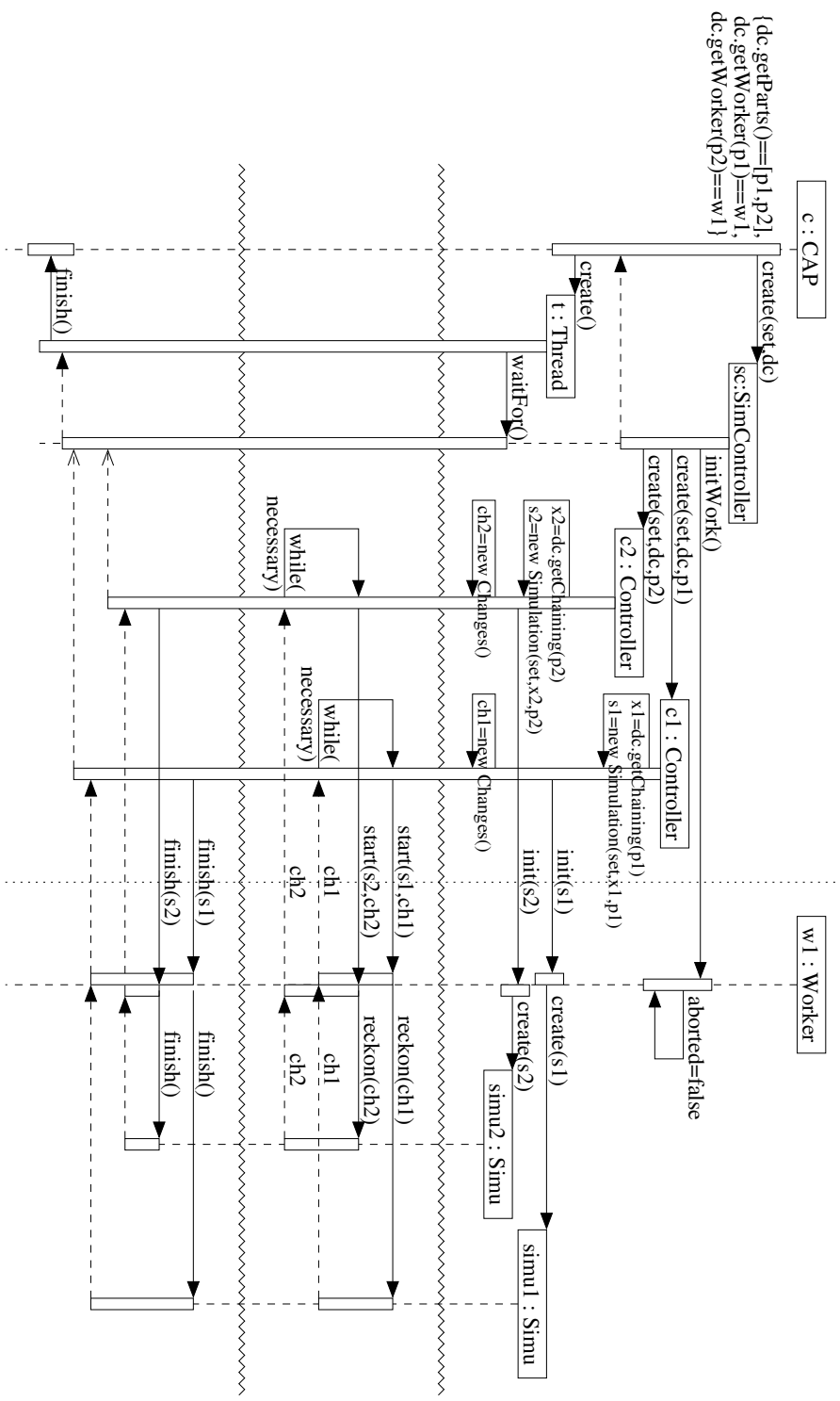


Abbildung 19.9: Sequenz-Diagramm: Verteile Simulation.

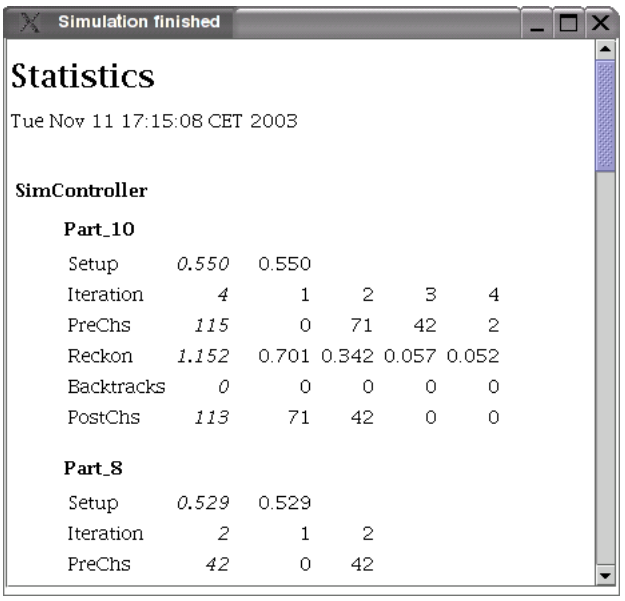


Abbildung 19.10: Anzeige der Simulations-Statistiken in der GUI.

19.7 Simulation überwachen

Zum Überwachen der Simulation kann der Simulator im Objekt `Simu` seinen Zustand in der CAP graphisch darstellen. Das sieht dann wie folgt aus: Abbildung 19.11 [S. 125].

Hier gibt es für jeden Simulations-Teil und damit jede lokale Simulation ein eigenes Fenster (bei der GUI), in das die lokalen Simulatoren `Simu`, die ja entfernt auf einem anderen Rechner ausgeführt werden, ihren Zustand *einzeichnen* können. In jedem Fenster sieht man die Zeitlinie aller Züge, die in dem Teil fahren. Jeweils links ist die Zugnummer an dem Zeitpunkt eingezeichnet, zu dem der Zug den Teil betritt, und am rechten Rand an dem Zeitpunkt, zu dem der Zug den Teil verlässt. Die Zeitpunkte sind durch eine Linie verbunden. Wenn eine solche Linie nicht vom linken zum rechten Rand verläuft, sondern von der Mitte oder bis zur Mitte, dann startet der Zug dort oder endet dort. Der Ablauf dieses Mechanismus ist in Abbildung 19.12 [S. 125] zu sehen.

Jedes `Simu`-Objekt erzeugt ein `Drawing`. Das `Drawing` überträgt die Zeichen-Methoden an die CAP, die aufgrund der eindeutigen `id` mit der Abbildung `frames` (Abschnitt 18.4, S. 107) das Fenster ermittelt, in das gezeichnet werden soll. Per `notify()` können alle Simulatoren wichtige Text-Meldungen an die `CAP` schicken, die dort in einem gemeinsamen Fenster angezeigt werden.

Dieser Ansatz realisiert für den lokalen Simulator quasi eine *Remote-GUI*. Das ist technisch interessant und anspruchsvoll und für unser System besonders gut geeignet: Dadurch gehört die graphische Darstellung des Simulators zum Simulator selbst und nicht zum Beispiel zur Kontroll-Anwendung CAP. So kann man auch für jede konkrete Simulator-Anwendung eine eigene graphische Darstellung definieren, ohne an der CAP etwas ändern zu müssen. Dieses Design ist übrigens sehr performant, unsere Laufzeittests (siehe Kapitel 25, S. 160) bele-

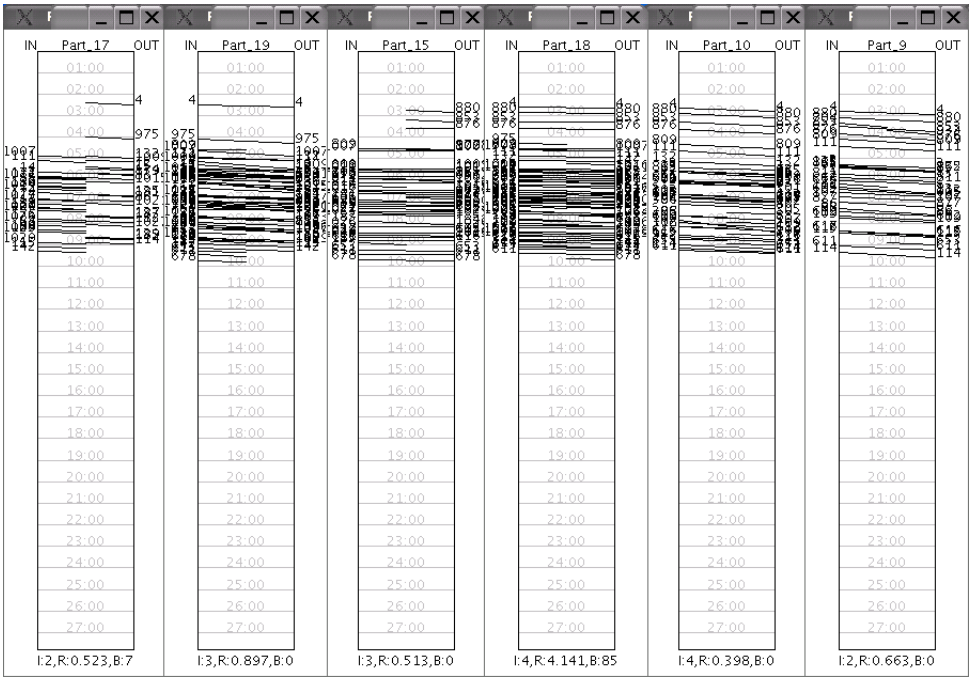


Abbildung 19.11: Darstellung des Berechnungszustands der verteilten lokalen Simulatoren in der GUI.

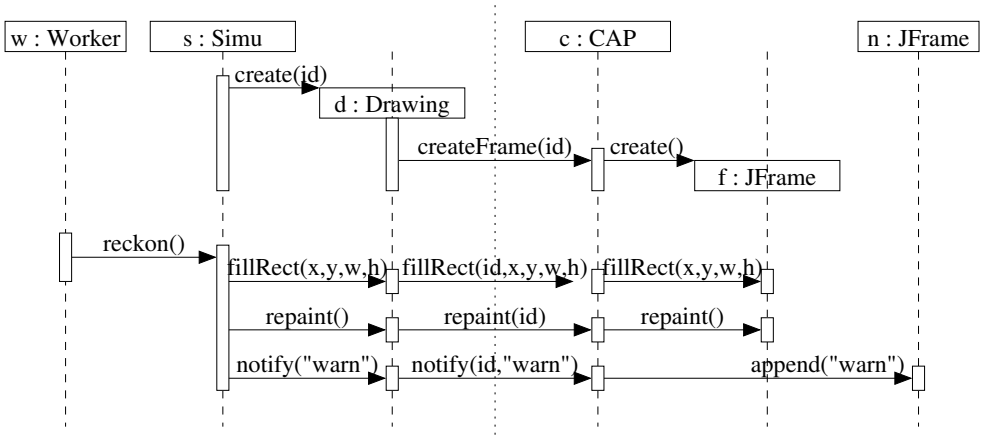


Abbildung 19.12: Sequenz-Diagramm: Darstellung des Berechnungszustands der verteilten lokalen Simulatoren.

gen das. Es müssen jeweils nur relativ wenige Zeichenbefehle übertragen werden und nicht etwa komplexe Datenstrukturen.

Zusätzlich zu der Möglichkeit, eine Zusammenfassung des eigenen Zustands bei der CAP anzuzeigen, hat jedes `Simu`-Objekt noch eine lokale Zeichenmöglichkeit, mit der detailliertere Abbilder vor allem des lokalen Suchprozesses gemacht werden. Abbildung 19.13 [S. 126] zeigt, wie das aussieht. Wir sehen hier zwei Fenster zweier `Simu`-Objekte, die beide auf demselben Worker laufen und die zeitlichen Verhältnisse einiger ihrer `Block`-Objekte darstellen.

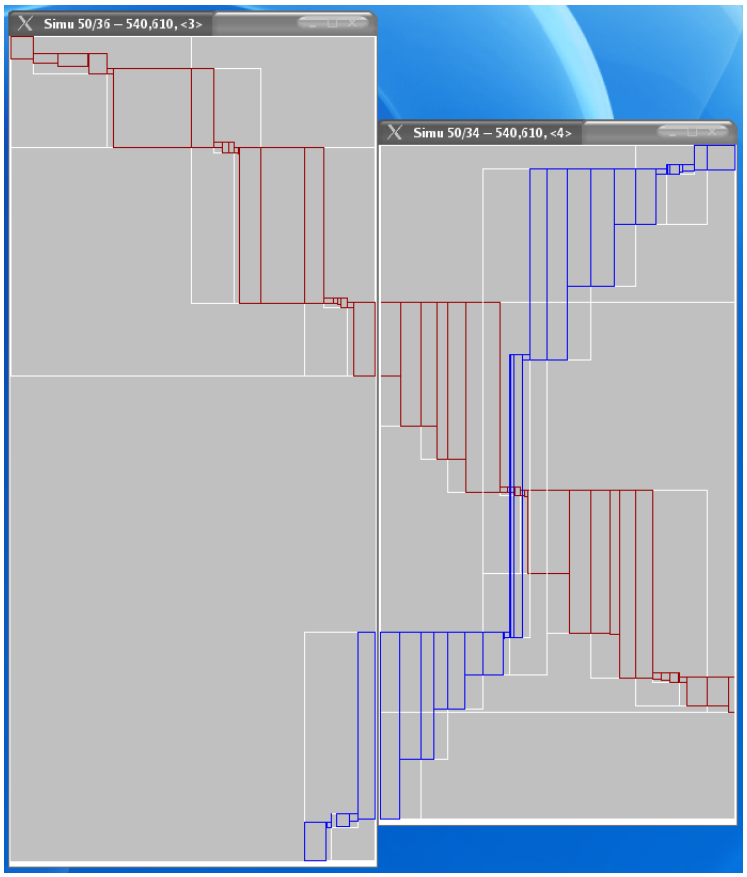


Abbildung 19.13: Detaillierte Darstellung der Berechnungsergebnisse lokal bei den Simulatoren.

Hier kann man übrigens sehen, dass die Belegungszeiten entsprechender Blöcke an den Grenzen auf die Sekunde identisch sind (waagerechte Linien sind auf gleicher Höhe), wie wir das ja für das Ergebnis der verteilten Simulation gefordert haben.

19.8 Simulation abbrechen

Eine laufende Simulation kann durch den Anwender durch Drücken des STOP-Knopfs in der GUI abgebrochen werden: siehe Abbildung 19.14 [S. 127].

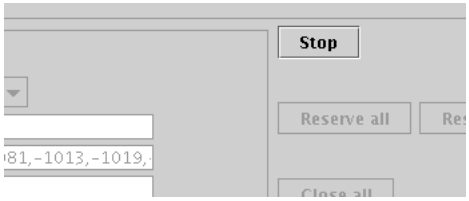


Abbildung 19.14: Abbruch-Button in der GUI.

Abbildung 19.15 [S. 128] zeigt den zeitlichen Ablauf der Vorgänge zum Abbruch einer Simulation mit zwei Parts und einem Worker entsprechend Abschnitt 19.6 [S. 121].

Wir sehen hier, dass der Abbruch auch durch das System selbst ausgelöst werden kann, zum Beispiel indem die `reckon()`-Methode einen Fehler wirft (`SimError`: wieder eine asynchrones Signal, in der Abbildung gekennzeichnet durch einen einfachen gestrichelten Pfeil). Dieser wird vom `Worker` an den `SimController` weitergereicht, der schließlich den Abbruch-Vorgang einleitet.

Daraufhin wird überall `aborted=true` gesetzt, damit zum Beispiel im Worker folgende Aufrufe von `init()` oder `start()` bzw. `create()` oder `reckon()` den Abbruch erkennen können. Alle diese Methoden beenden dann sofort.

Ein Abbruch wirkt sich von der CAP aus gesehen auf das ganze System aus: Alle reservierten Worker müssen all ihre Aufgaben abbrechen, sie sind ja immer exklusiv zugeordnet. Die Controller werden übergangen und die Worker direkt zum Abbruch aufgefordert.

`Controller` ist eine innere Klasse von `SimController`, und hat dadurch Zugriff auf dessen `aborted`-Variable. Die einfachen (Signal-) Pfeile symbolisieren diese Auswirkung. Indem `aborted` auf `true` gesetzt wird, werden die `Controller`-Threads beendet, wodurch wiederum im `SimController` die Methode `waitFor()` beendet, siehe auch Abbildung 19.9 [S. 123].

19.9 System beenden

Das *ganze System* sollte eigentlich nie beendet werden. Natürlich beendet der Anwender die Benutzeroberfläche regelmäßig, per Menü-Befehl aus der GUI. Ein Worker muss eigentlich nur beendet werden, wenn die Arbeitsstation abgeschaltet wird. Und der DIR sollte eigentlich immer laufen, wie es sich für Server-Dienste eben gehört.

Ein Worker kann durch folgendes Kommando – auf der jeweiligen Workstation, also nicht zentral! – heruntergefahren werden:

```
hs@workstation:~/PRJDRS/WRK> ant stop
```

Im Prinzip geschieht dann genau das Umgekehrte wie beim Worker-Start (Abschnitt 19.4, S. 119): Der Worker meldet sich beim DIR ab, dieser trägt ihn aus seinen Listen aus und informiert etwaige CAPs.

Der DIR wird genauso direkt auf dem Server-Rechner beendet:

```
hs@server:~/PRJDRS/DIR> ant stop
```

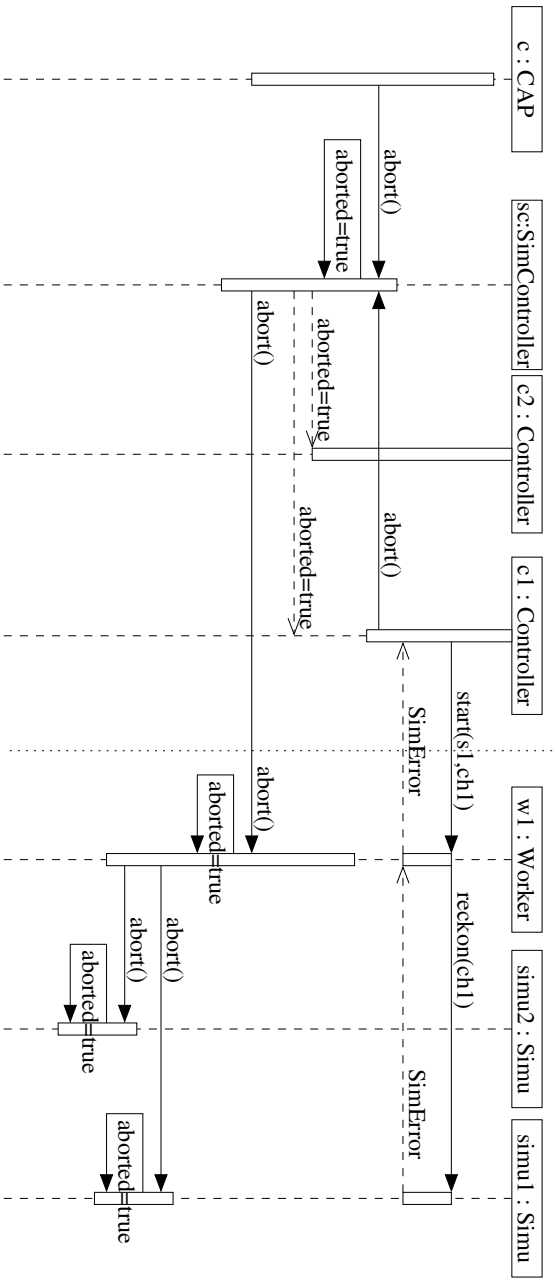


Abbildung 19.15: Sequenz-Diagramm: Simulationsabbruch.

Hier passiert aber mit dem Rest des System nichts, insbesondere wird sonst nichts heruntergefahren. Typischerweise sollten aber beim Austausch des DIR keine CAP laufen. Dann kann der DIR auf dem Server ausgetauscht werden, ohne dass der Rest des Systems neu gestartet werden muss. Die Worker laufen weiter und melden sich ohnehin regelmässig beim zentralen DIR an – siehe Abschnitt 19.4, S. 119 –, also auch beim erneuerten.

20 Kontrolle

Die Kontrolle des Algorithmus ist ein besonders kritischer Teil der Implementierung: Dieses Kapitel widmet sich der zentralen Kontrolle, zeigt formal, warum die Implementierung korrekt ist, und beschreibt die zusätzlich realisierte verteilte Kontrolle.

Nachdem wir nun das Design und die wesentlichen internen Abläufe von DRS gesehen haben, werde ich im Folgenden den Kern der Implementierung beschreiben: die Kontrolle des verteilten Algorithmus. Wir haben hier zunächst eine zentrale Kontrolle implementiert, die im `SimController` der CAP läuft (Abschnitt 20.1, S. 130). Für diesen Algorithmus konnten wir sogar formal die Korrektheit bzgl. Deadlockfreiheit und Terminierung beweisen (Abschnitt 20.2, S. 131). In verteilten Systemen ist es meist wünschenswert, gänzlich ohne zentrale Kontrolle auszukommen: damit kann ein kritischer Kommunikations-Engpass vermieden werden und die Stabilität des Systems kann durch Redundanz erhöht werden. Manche machen sogar die Abwesenheit einer Zentrale zum formalen Kriterium verteilter Systeme. Dafür haben wir die zentrale Kontrolle dezentralisiert (Abschnitt 20.3, S. 137). Alle diese Dinge hat Julia Ruhmane in [Ruh04] detailliert dargestellt. Am Ende dieses Kapitels gehe ich noch kurz auf die synchronisierte Kontrolle ein, den Ansatz, der den verteilten Algorithmus deterministisch macht (Abschnitt 20.4, S. 139).

20.1 Zentrale Kontrolle

In den vorhergehenden Kapiteln habe ich immer sehr genau UML-Notation und -Diagramme verwendet. Für abstrakte Darstellungen von Abläufen sieht UML *State-Charts* vor [BRJ99, 24]. Hier verwende ich die sehr ähnlichen *Petri-Netze*, die ich in Abschnitt 2.3.1 [S. 12] schon kurz vorgestellt habe, unter anderem wegen der umfangreich verfügbaren Beweismethoden.

Abbildung 20.1 [S. 131] zeigt das Petrinetz $\mathcal{N}_1$, das den Ablauf der zentralen Kontrolle in *einem* `Controller c` beschreibt (siehe auch Abschnitt 19.6, S. 121).

Der Ausgangszustand ist: `changesFilled,notWorking` (d.h., es gelten die beiden lokalen Zustände `changesFilled` und `notWorking`). Die einzig mögliche Aktion ist danach `startWorker`, weil nur für diese beide Eingangs-Zustände als Vorbedingung gelten. `startWorker` bedeutet, der `Controller c` veranlasst `Simu`, eine Berechnung durchzuführen. Das Ergebnis der Berechnung ist zum einen ein Ergebnis `result` und zum anderen der Zustand `changesEmpty`. Das Ergebnis `result` führt dazu, dass neue Changes an andere Controller geschickt werden (`addChanges`). Wenn nach einer Berechnung keine Änderungen an Nachbarn geschickt werden müssen, macht `addChanges` nichts, der lokale Zustandsübergang ist derselbe.

Der neue Zustand von `c` ist danach `changesEmpty,notWorking`. Jetzt können zwei Dinge geschehen: erstens kann `fillChanges` ausgeführt werden, weil andere Controller Änderungen an `c` schicken, oder die Aktion `finish` beendet den Algorithmus. Formal ist in diesem Petrinetz dieser Nicht-Determinismus dadurch repräsentiert, dass der Zustand `changesEmpty` zwei Aktionen auslösen kann, `finish` und `fillChanges`.

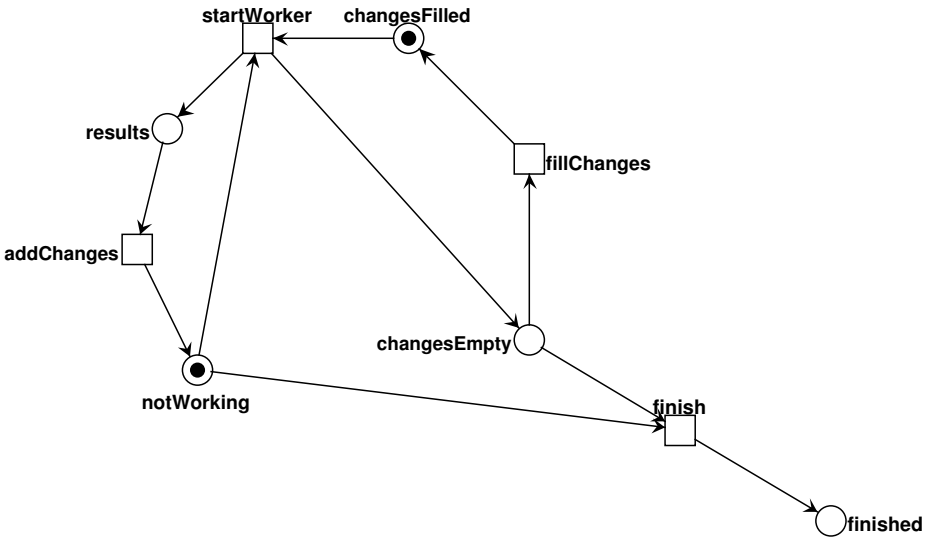


Abbildung 20.1: Petrinetz $\mathcal{N}_1$: Ablauf in einem `Controller c`. Abbildung aus [Ruh04].

20.2 Korrektheit der Implementierung

`changesFilled,notWorking`	$\xrightarrow{\texttt{startWorker}}$	`changesEmpty,results`
`changesEmpty,results`	$\xrightarrow{\texttt{addChanges}}$	`changesEmpty,notWorking`
`changesEmpty,results`	$\xrightarrow{\texttt{fillChanges}}$	`changesFilled,results`
`changesFilled,results`	$\xrightarrow{\texttt{addChanges}}$	`changesFilled,notWorking`
`changesEmpty,notWorking`	$\xrightarrow{\texttt{fillChanges}}$	`changesFilled,notWorking`
`changesEmpty,notWorking`	$\xrightarrow{\texttt{finish}}$	`finished`

Tabelle 20.1: Mögliche Zustandsübergänge im Petrinetz $\mathcal{N}_1$.

Ganz genau und formal sehen wir in Tabelle 20.1 [S. 131] alle möglichen Zustandsübergänge im Petrinetz $\mathcal{N}_1$. Hier steht jeweils links ein möglicher globaler Zustand, der sich aus jeweils geltenden lokalen Zuständen zusammensetzt (im Petrinetz dargestellt durch mit Marken besetzte Stellen), über dem Pfeil steht die ausgeführte Aktion (bzw. die Transition) und rechts des Pfeils der folgende globale Zustand.

Wir können unmittelbar ablesen, dass es – abgesehen vom intendierten Endzustand `finished` – keinen Zustand gibt, der keinen Nachfolgezustand hat, bzw. bei dem keine Aktion ausgeführt werden kann. Das System ist demnach *deadlockfrei* (oder *schwach lebendig*, siehe [Bau96, 6.1.2]). Außerdem sehen wir, dass das System nur dann beendet werden kann, wenn nicht gerechnet wird und keine

Änderungen mehr anstehen (`changesEmpty,notWorking`).

Dieses System stellt nur die Abläufe in *einem* `Controller` dar, obwohl zum Beispiel zwischen `addChanges` des Controllers `c` und `fillChanges` der anderen Controller ein wichtiger Zusammenhang besteht. Dafür müssen wir mehrere `Controller` in einem Netz darstellen. Mit den einfachen Stellen-Transitions-Netzen kann man das nur für eine gegebene Anzahl von Controllern machen, was dann auch leicht komplex wird. Abbildung 20.2 [S. 132] zeigt das entsprechende Netz für drei Controller. Auch diese Abbildung stammt aus [Ruh04].

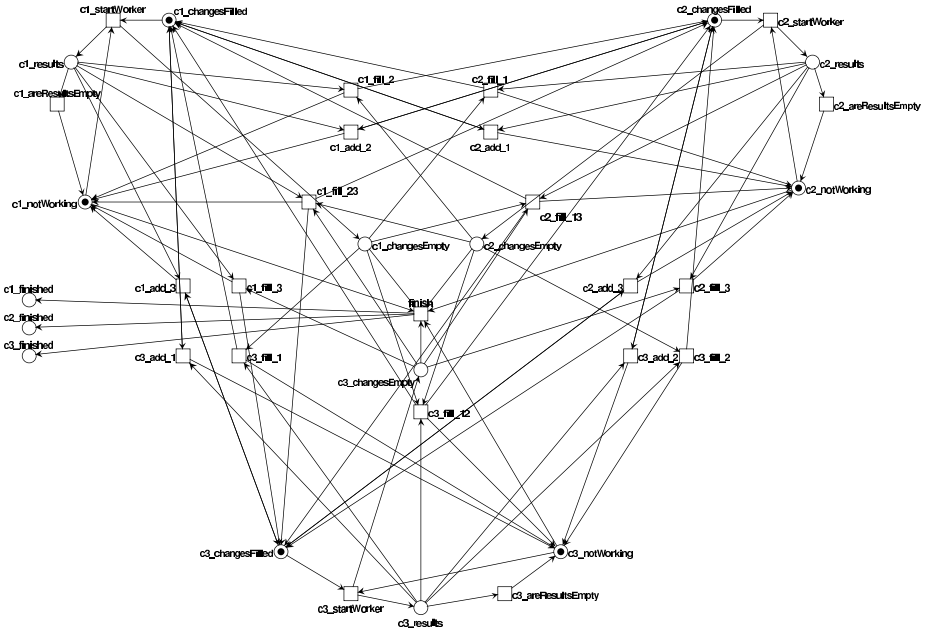


Abbildung 20.2: $\mathcal{N}_3$: Ablauf in einem System mit drei Controllern, `c1`, `c2` und `c3` [Ruh04].

20.2.1 Höhere Petri-Netze

Um Aussagen zur Implementierung des Kontroll-Algorithmus bezüglich *beliebig (endlich) vieler* Controller machen zu können, benötigen wir mächtigere Petrinetze als die einfachen *Stellen-Transitions-* oder *ST-Netze*. Generell werden mächtigere Systeme als *höhere Petrinetze* bezeichnet.

Wir verwenden die *Systeme mit individuellen Marken* oder auch *IM-Systeme* von Baumgarten [Bau96, 7]. Diese unterscheiden sich von den ST-Netzen in drei wesentlichen Punkten:

Stellen Die Stellen sind Multi-Mengen von Marken unterschiedlicher Typen.

Transitionen Die Transitionen verfügen über komplexe Bedingungen, die ausdrücken, wann eine Transition schalten oder *feuern* kann.

Kanten Kanten zwischen Stellen und Transitionen können ein Muster haben, das beschreibt, welche Marken über die Kante wandern können.

Definition 20.1 (IM-Netz)
Etwas formaler ([Bau96, 7] folgend) ist ein IM-Netz ein Tupel (S, T, F, D, P, W, M_0) mit

(1) S Menge der Stellen.

(2) T Menge der Transitionen, $T \cap S = \emptyset$.

(3) $F \subset (S \times T) \cup (T \times S)$ Flussrelation oder Graph des Netzes.

(4) $\forall s \in S : D(s)$ Stellentyp.

(5) $\forall t \in T : P(t)$ Transitionsprädikat.

(6) $\forall f \in F : W(f)$ Kantengewicht oder -Muster.

(7) $\forall s \in S : M_0(s) : D(s) \to \mathbb{N}_0$ initiale Belegung.

Ich möchte hier nicht noch formaler werden. Insbesondere die D, P und W sind formal recht komplizierte Konstruktionen, die ich hier nicht genauer ausführen möchte. Die geneigte Leserin möge bitte [Ruh04] oder [Bau96] bemühen.

Definition 20.2 ($\mathcal{N}_Z$)
Betrachten wir Abbildung 20.3 [S. 134], die Beschreibung des verteilten Algorithmus als IM-Netz $\mathcal{N}_Z$. Sei Z im Folgenden die Menge aller Controller in dem gegebenen DRS-System, über das wir hier sprechen und dessen Korrektheit wir zeigen wollen.

(1) Die Menge S der Stellen oder lokalen Zustände in $\mathcal{N}_Z$ ist `{changesFilled, results, notWorking, changesEmpty, finished}` und damit gleich der von $\mathcal{N}_1$. $\mathcal{N}_Z$ beschreibt die Zustandsübergänge in allen Controllern Z aus der Sicht eines $x \in Z$. Wenn eine Stelle eine Marke x enthält, bedeutet das, dass der Controller x sich im entsprechenden Zustand befindet. Jede Stelle enthält eine Teilmenge von Z als Marken, es können sich also immer keiner oder einige oder alle Controller in jedem der Zustände befinden.

   Hier die genauere Bedeutung einzelner Stellen (siehe auch [Ruh04]):

   `changesFilled` symbolisiert Changes, mit denen Controller ihre Worker starten. Falls `changesFilled` ein Controllersymbol enthält, bedeutet das, dass der entsprechender Controller seinen Part noch mal mit aktualisierten Changes berechnen muss.

   `results` symbolisiert Changes, die der Worker eventuell nach der Berechnung an seinen Controller liefert. Falls `results` ein Controllersymbol enthält, bedeutet das, dass der entsprechende Controller seinen Part berechnet, aber die Changes an andere Controller noch nicht verschickt hat.

   `changesEmpty` symbolisiert den Zustand des Controllers, in dem der Controller seinen Part berechnet und (noch) keine neuen Changes von anderen Controllern erhalten hat.

   `notWorking` symbolisiert den Zustand des Controllers, in dem der Controller seinen Part berechnet und seine Changes an andere Controller schon ausgeliefert hat.

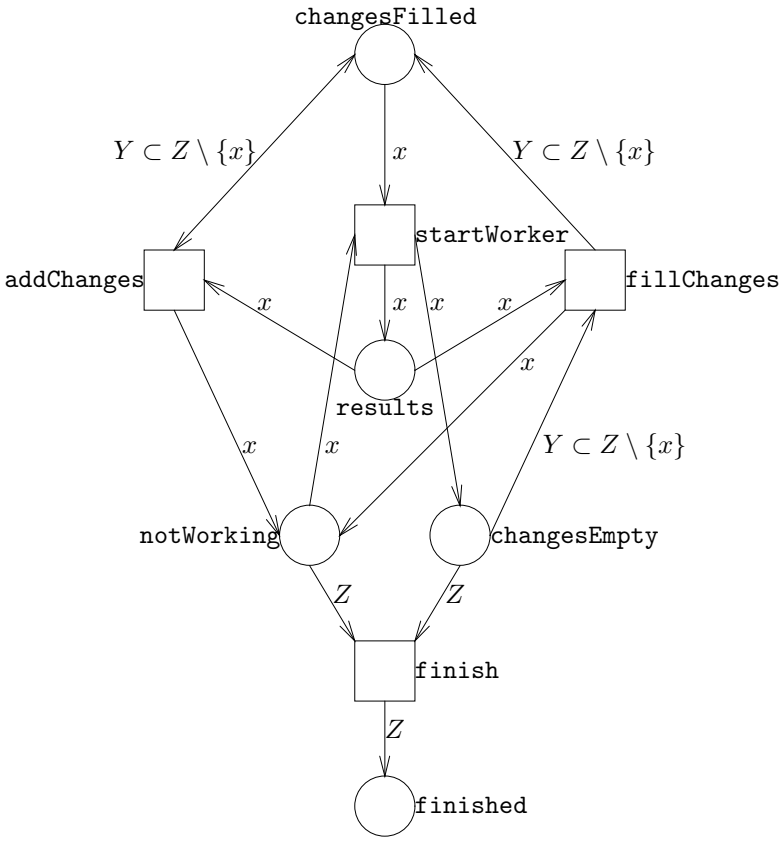


Abbildung 20.3: $\mathcal{N}_Z$: Höheres Petrinetz zur Beschreibung der Abläufe einer Menge Z von Controllern [Ruh04]. $x \in Z$ ist ein beliebiger Controller, Y ist die Menge der Controller an die x (evtl.) Changes schickt (deshalb: $Y \subset Z \setminus \{x\}$), Y darf auch leer sein.

`finished` wird nur erreicht, wenn die Simulation aller Parts von den Controllern erfolgreich abgeschlossen wurde.

(2) Genauso ist $T =$ `{addChanges, startWorker, fillChanges, finish}` in $\mathcal{N}_Z$ identisch der Menge der Transitionen in $\mathcal{N}_1$. Die Bedeutung der Transitionen, die Zustandsübergänge und meist konkrete Aktionen repräsentieren, im einzelnen:

`startWorker` symbolisiert die Berechnung des Parts durch einen Controller x (bzw. den damit gesteuerten Simulator). Dafür müssen die Vorbedingungen `notWorking` und `changesFilled` gelten (mit x besetzt sein). Die beiden x-Marken wandern dann zu den Zuständen `results` und `changesEmpty`.

`fillChanges` stellt das Versenden der errechneten Changes an andere Controller, deren `changesFilled` leer sind, dar. Die Marke x des versendenden Controllers geht dabei zu der Stelle `notWorking`. Die Eingangskante von der Stelle `changesEmpty` und die Ausgangskante zu der Stelle `changesFilled` sind mit $Y \subset Z \setminus \{x\}$ markiert. Das bedeutet, dass per `fillChanges` eine Teilmenge Y aller Controller Z ihren Zustand von `changesEmpty` nach `changesFilled` ändern. Hier im Petri-Netz erfolgt die Auswahl der Y nicht-deterministisch, da hier nicht die Berechnung im Simulator modelliert wird. In der Anwendung sind die Y genau die, an welche x Änderungen schickt (und die nicht ohnehin schon `changesFilled` haben).

Ein Beispiel: x will an y und z Änderungen schicken, y aber befindet sich schon im Zustand `changesFilled`. Dann geht x von `results` über `fillChanges` zu `notWorking`, die Menge $Y = \{z\}$ geht gleichzeitig von `changesEmpty` zu `changesFilled`.

`addChanges` funktioniert ähnlich `fillChanges`, ist aber für den Fall da, wenn alle Nachbarn von x, an die Änderungen geschickt werden müssen, sich bereits im Zustand `changesFilled` befinden. Der Doppelpfeil zwischen `changesFilled` und `addChanges` symbolisiert, dass es für die Anwendung von `addChanges` einige Y geben muss, die im Zustand `changesFilled` sind, diese aber ihren Zustand nicht ändern.

`finish` feuert nur, wenn alle Controller die Zustände `changesEmpty` und `notWorking` erreicht haben. Dabei werden alle Controllermarken diesen Stellen entnommen und der Stelle `finished` hinzugefügt. Das Ankommen aller Controller bei `finished` bedeutet erfolgreiche Termination der Berechnung.

(3) Die Flussrelation oder den Graphen F kann man unmittelbar an Abbildung 20.3 [S. 134] ablesen.

(4) Der Stellentyp aller Stellen ist gleich der Menge der Controller Z: $\forall s \in S : D(s) = Z$.

(5) Die Transitionsprädikate P sind durch obige informelle Beschreibung der Stellen und Transitionen gegeben.

(6) Die Kantengewichte W sind durch obige informelle Beschreibung der Stellen und Transitionen gegeben.

(7) Initiale Belegung M_0: Weil der Typ aller Stellen gleich ist und wir für die Stellen keine Multi-Mengen benötigen, können wir etwas einfacher als oben M_0 als Relation definieren: $M_0 = \{(\texttt{notWorking}, x), (\texttt{changesFilled}, x) \mid x \in Z\} \subset S \times Z$.

Zu Anfang befinden sich also alle Controller im Zustand: `changesFilled,notWorking`.

Aus diesen Definitionen ergeben sich folgende mögliche Zustandsübergänge in $\mathcal{N}_Z$: siehe Tabelle 20.2 [S. 136].

`changesFilled,notWorking`	$\xrightarrow{\texttt{startWorker}}$	`changesEmpty,results`
`changesEmpty,results`	$\xrightarrow{\texttt{addChanges}}$	`changesEmpty,notWorking`
`changesEmpty,results`	$\xrightarrow{\texttt{fillChanges}}$	`changesEmpty,notWorking`
`changesFilled,results`	$\xrightarrow{\texttt{addChanges}}$	`changesFilled,notWorking`
`changesFilled,results`	$\xrightarrow{\texttt{fillChanges}}$	`changesFilled,notWorking`
`changesEmpty,notWorking`	$\xrightarrow{\texttt{finish}}$	`finished`
`changesEmpty,notWorking`	$\xrightarrow{\texttt{(fillChanges)}}$	`changesFilled,notWorking`
`changesEmpty,results`	$\xrightarrow{\texttt{(fillChanges)}}$	`changesFilled,results`
`changesFilled,notWorking`	$\xrightarrow{\texttt{(addChanges)}}$	`changesFilled,notWorking`
`changesFilled,results`	$\xrightarrow{\texttt{(addChanges)}}$	`changesFilled,results`

Tabelle 20.2: Mögliche Zustandsübergänge in $\mathcal{N}_Z$ (Abbildung 20.3, S. 134). Die ersten sechs Übergänge beziehen sich auf x, die anderen (in Klammer) auf $Y \subset Z \setminus \{x\}$.

20.2.2 Deadlock-Freiheit

Deadlock-Freiheit heißt auch hier:

(1) dass alle globalen Zustände außer dem intendierten Endzustand (hier `finished`) mindestens einen möglichen globalen Nachfolgezustand haben, und

(2) dass der entsprechende Zustandsübergang auch ausgeführt werden kann.

Die Erfüllung der Bedingung (1) kann man leicht an der Tabelle 20.2 [S. 136] nachvollziehen. Anders als bei einfachen S-T-Netzen ist hier die Bedingung (2) nicht trivial. Wir müssen die einzelnen Fälle untersuchen.

Betrachten wir wieder die Übergänge im Controller x. Dieser kann sich laut Tabelle in einem der folgenden vier Zustände befinden:
`changesEmpty,notWorking`, `changesEmpty,results`, `changesFilled,notWorking`, oder `changesFilled,results`.

Im Zustand `changesFilled,notWorking` kann x immer `startWorker` ausführen, diese Aktion hat keine weitere Bedingung.

Im Zustand `changesEmpty,results` kann immer entweder `addChanges` oder `fillChanges` ausgeführt werden, wie wir oben bei der Beschreibung der Transitionen gesehen haben. Ebenso im Zustand `changesFilled,results`.

Im Zustand `changesEmpty,notWorking` kann x genau dann in den Endzustand `finished` wechseln, wenn alle anderen Controller im selben Zustand sind. Und dieser globale Zustand tritt irgendwann ein, weil der verteilte Algorithmus DRS immer nach endlich vielen Schritten terminiert (Kapitel 15, S. 90).

Alle anderen Übergänge sind für die Deadlock-Freiheit im Controller x unwichtig. Sie können zwar nicht-deterministisch den internen Zustand ändern, aber immer von einem der obigen vier in einen anderen dieser vier.

20.2.3 Fairness / Terminierung

Damit der verteilte Algorithmus DRS terminiert, muss die Ausführung der Simulationsteile bzw. Controller *fair* sein (Definition 11.1 (4), S. 73): kein Controller, der rechnen muss (hier: `changesFilled`), darf unendlich lange verzögert werden, er muss dann nach endlich vielen Schritten sicher wieder rechnen können (`startWorker`).

Es gibt für den Controller x zwei Zustände, in denen `changesFilled` gilt:

(1) `changesFilled,notWorking` und

(2) `changesFilled,results`

In (1) kann x unmittelbar `startWorker` ausführen.

In (2) kann x auch immer sofort seine Änderungen per `addChanges` oder `fillChanges` (asynchron) an die Nachbarn verschicken, der Nachfolgezustand ist jeweils `changesFilled,notWorking`.

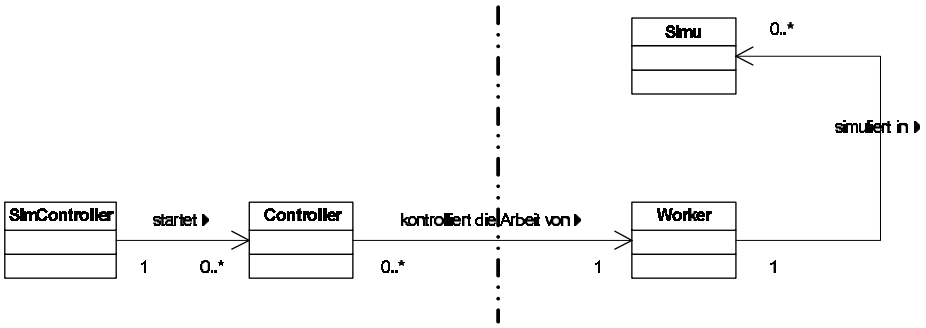
In jedem Fall also ist in x nach `changesFilled` die Aktion `startWorker` nach endlich vielen Schritten möglich.

20.3 Verteilte Kontrolle

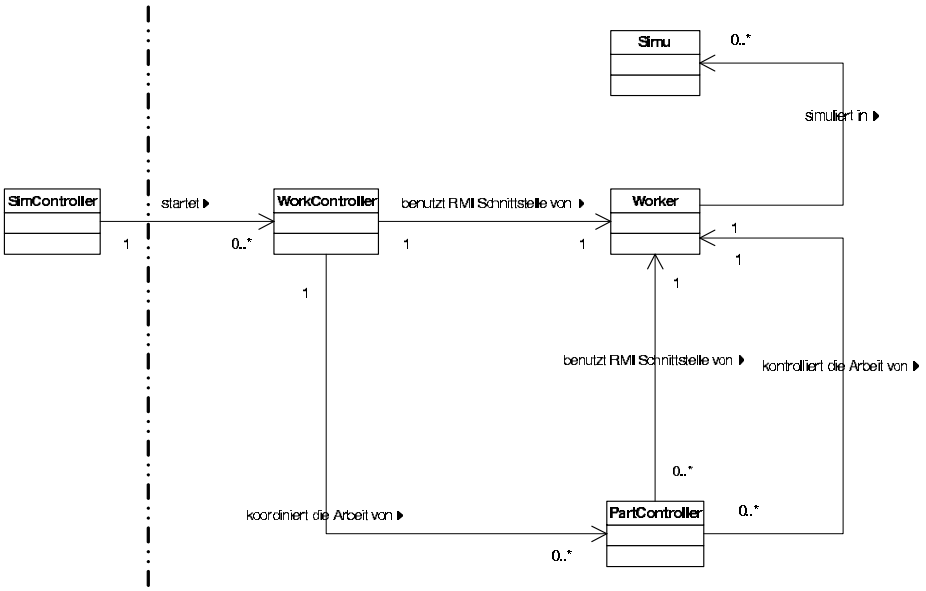
Bei der zentralen Kontrolle kommuniziert jeder Simulator ausschließlich mit der CAP: er sendet ihr die neuen Änderungen und empfängt von dort neue Änderungen. Das wird bei sehr großen Simulationen schnell zum Kommunikations--Engpass. Daher haben wir eine verteilte Kontrolle eingeführt. Abbildung 20.4 [S. 138] stellt zentrale und dezentrale Kontrolle gegenüber:

(a) Bei der zentralen Kontrolle hat jeder Simulationsteil einen Repräsentanten `Controller` in der CAP (links), der über den `Worker` die Simulation steuert. (Siehe auch Abschnitt 19.6, S. 121).

(b) Bei der verteilten Kontrolle ist der `Controller` aufgesplittet in `WorkController` und `PartController`. Dabei existiert pro `Worker` genau ein `WorkController` und für jeden Simulationsteil genau ein `PartController` und damit möglicherweise mehrere `PartController` pro `Worker`. Die `PartController` kommunizieren nicht mehr über die zentrale CAP miteinander sondern direkt mit andern `PartControllern` über den jeweiligen `WorkController`. Damit also findet die Kommunikation direkt zwischen den Workern statt.



(a)



(b)

Abbildung 20.4: Zentrale Kontrolle (a) und dezentrale Kontrolle (b). Abbildungen aus [Ruh04].

Wie ich schon angedeutet habe, kann bei der *zentralen* Kontrolle leicht festgestellt werden, ob Terminierung erreicht ist: dann, wenn sich alle `Controller` im Zustand `changesFilled,notWorking` befinden. Nicht aber bei der *dezentralen* Kontrolle: Um festzustellen, ob alle verteilten `PartController` sich in diesem Zustand befinden, könnten diese ihren Zustand an die zentrale CAP melden. Damit hätten wir aber wieder das Problem des Kommunikations-Engpasses. Deshalb haben wir einen moderneren Algorithmus zur Feststellung der Termination implementiert: DTD von Mahapatra und Dutt [MD01] (siehe Abschnitt 2.3.7, S. 18). Abbildung 20.5 [S. 140] zeigt beispielhaft den Terminierungs-Ablauf aus unserer Implementierung.

20.4 Synchronisierte Kontrolle

Die *nicht-synchronisierte* Kontrolle führt dazu, dass die Ausführung von DRS nicht-determiniert ist: Führt man für das selbe Simulationsproblem mehrere *globale* Simulationen durch, können die lokalen Simulationen jeweils in unterschiedlicher Reihenfolge ausgeführt werden. Das kann jeweils zu einem anderen (globalen) Simulationsergebnis führen. Die Partner im Forschungsprojekt SIMONE fordern aber *Reproduzierbarkeit* ihrer Simulationen. Daher haben wir einen synchronisierten Ablauf realisiert, wie er in Abschnitt 16.2 [S. 92] skizziert ist.

Dazu werden die lokalen Abläufe synchronisiert: vor dem Versenden der Nachrichten (Änderungen) und vor dem Starten der lokalen Simulationen, im Netz $\mathcal{N}_Z$ aus Abbildung 20.3 [S. 134] also quasi vor der Aktion `startWorker` und vor `addChanges` bzw. `fillChanges`. Diese Synchronisationspunkte darf jeder Controller erst dann überschreiten, wenn alle anderen Controller denselben Punkt erreicht haben. Damit wird erreicht, dass die lokalen Simulationen immer in derselben Reihenfolge berechnet werden, und dass nach jeder Simulation immer alle Änderungen an alle Nachbarn verschickt werden. Mithin macht dieses synchronisierte Verfahren den Algorithmus DRS *determiniert*.

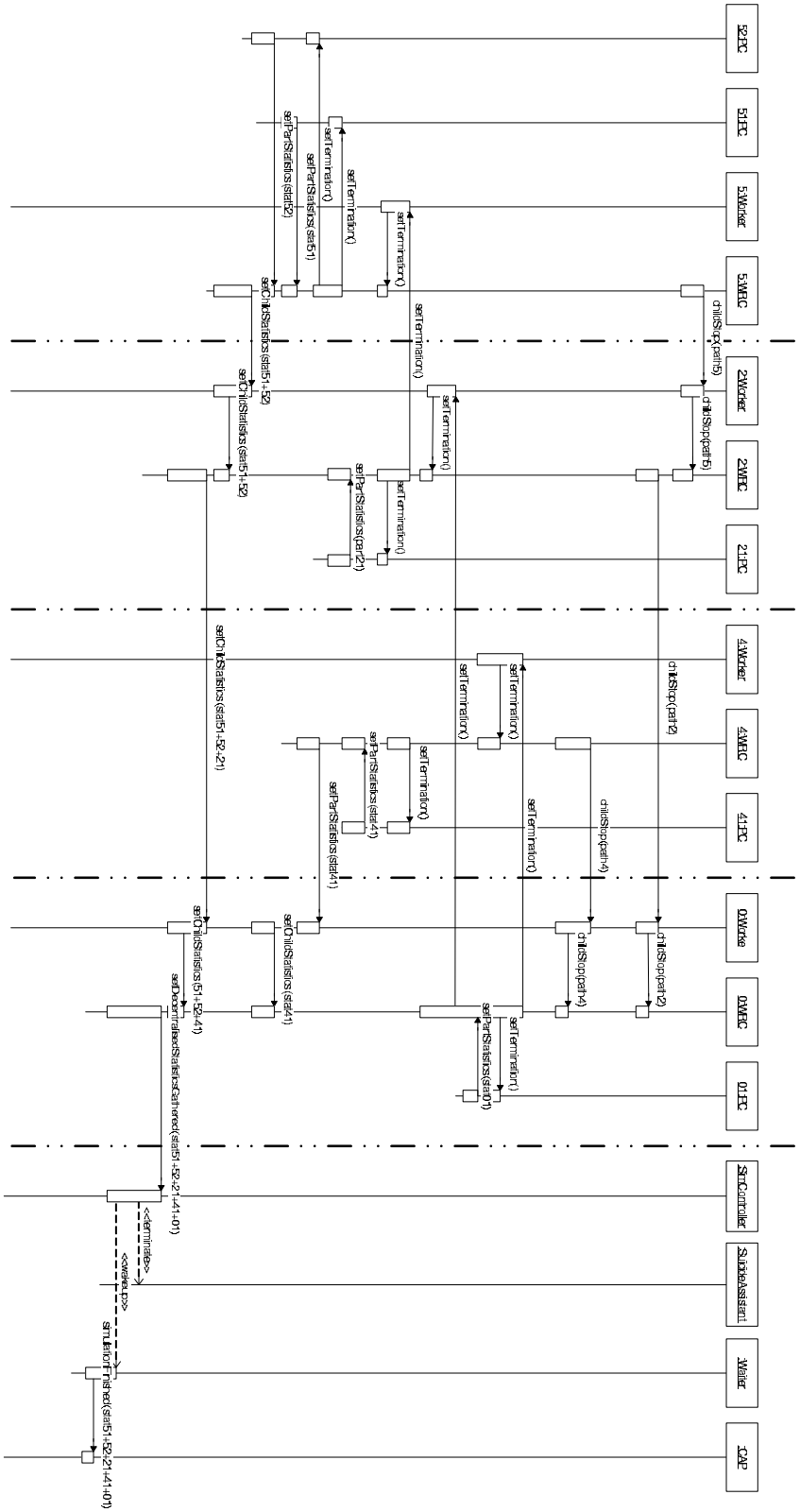


Abbildung 20.5: Terminierungs-Feststellung in unserer Implementierung des Algorithmus aus [MD01]. Abbildung aus [Ruh04].

21 Problem-Zerlegung

Jedes gegebene Simulationsproblem muss zur Bearbeitung durch das verteilte Verfahren DRS in Teilprobleme zerlegt werden, und diese müssen dann auf die vorhandenen Rechenknoten verteilt werden. In diesem Kapitel wird die Abstraktion `DisConfig` beschrieben, wie diese Zerlegung und Verteilung erledigt, und wie sie durch optimierende, adaptive Algorithmen erweitert werden kann.

Wie ich oben in Abschnitt 2.2 [S. 9] und Abschnitt 2.3.4 [S. 17] begründet habe, braucht man für ein konkretes Problem zur Berechnung auf einem verteilten System eine konkrete Konfiguration. Das zu berechnende Problem muss zerlegt und auf die verfügbaren Knoten verteilt werden, damit diese die lokalen Simulationen durchführen können. Wie das im Einzelnen geschieht, gibt eben die Konfiguration vor.

Aus der Darstellung des Algorithmus DRS und den zugrunde liegenden Daten in Teil II [S. 51] ergibt sich, dass ein Simulationsproblem entlang der Infrastruktur zerlegt werden muss. Jedes Element der Infrastruktur muss einem Simulationsteil zugeordnet werden. Zerlegung und Verteilung sind Optimierungsprobleme: Es hängt durchaus von der konkreten Konfiguration ab, wie schnell die globale Simulation berechnet wird. Dabei sind vor allem zwei Parameter zu optimieren: der Kommunikationsaufwand und die lokalen Rechenzeiten.

Der Kommunikationsaufwand zwischen Rechenknoten und damit zwischen Simulationsteilen hängt unmittelbar mit der Anzahl der Züge zusammen, die zwischen benachbarten Teilen hin und her fahren: Wenn viele Züge über Grenzen fahren, müssen viele Daten kommuniziert werden. Und der lokale Rechenaufwand ist unter anderem von der lokalen Problem-Komplexität abhängig: Wenn diese groß ist, muss lokal auch viel gerechnet werden. Der Rechenaufwand hängt aber zusätzlich auch von den grenzüberschreitenden Zügen ab: Wenn viele Züge über Grenzen fahren, müssen die Ein- und Ausfahrzeiten vieler Züge global konsistent gemacht werden. Das führt dazu, dass oft iteriert werden muss.

Auf das Netz bezogen, muss ein gegebenes Simulationsproblem also derart zerlegt werden, dass

- die lokale Problemkomplexität möglichst gleichverteilt ist, und
- die Anzahl der grenzüberschreitenden Züge minimal ist.

Glücklicherweise gibt es sehr gute Graph-Partitionierungsverfahren, mit denen wir unser Problem lösen können. Diese schneiden ein gegebenes gewichtetes Netz so auseinander, dass die Partitionierung bzgl. verschiedener Kriterien möglichst gut ist. Partitionierung bedeutet hier immer, dass eine Abbildung zwischen der Menge der Knoten und der Menge der gewünschten Teile erzeugt wird, also jeder Knoten genau einem Teil zugeordnet wird. Die Anzahl der gewünschten Teile muss dabei gegeben sein.

Wir bilden das gegebene Simulationsproblem auf ein gewichtetes Netz ab, zerschneiden es mit einem solchen Verfahren, und verwenden die Partitionierung als Problemzerlegung.

21.1 Zerlegungsverfahren Metis

Wir verwenden in DRS das Graph-Partitionierungsverfahren Metis [KK99] [SKK00b][SKK01]. Dieses ist eine besonders effiziente und frei erhältliche Im-

plementierung von *Multi-Level, Multi-Constraint, Multi-Objective* Partitionierungs-Algorithmen. Metis kann also Netze anhand mehrerer Kriterien zerlegen, wobei zusätzliche Nebenbedingungen berücksichtigt werden können. Der zentrale *Multi-Level*-Ansatz ist folgender:

1. Coarsening
2. Partitioning
3. Uncoarsening

In der ersten Phase, dem *Coarsening* (engl. *to coarsen* = vergröbern), wird der zu zerlegende Graph schrittweise vereinfacht, wobei die Anzahl der Knoten reduziert wird. Knoten des Ausgangsgraphen werden dabei jeweils zusammengefasst, wobei die Gewichte bzw. Stärken der Verbindungen zwischen den Knoten berücksichtigt werden.

Das Ergebnis des Coarsening ist also ein deutlich vereinfachter Graph, der in der *Partitioning*-Phase in Teile zerlegt wird (hier noch nicht notwendigerweise schon in die Anzahl der gewünschten Teile). Die Partitionierung achtet darauf, dass möglichst gleich schwere Teile entstehen, also die Knoten so auf die Teile verteilt werden, dass die Summe der Gewichte möglichst gleichmäßig verteilt ist.

In der letzten Phase, dem *Uncoarsening*, werden die Teile schrittweise verfeinert: Die zusammengefassten Knoten werden aufgelöst, wobei darauf geachtet wird, dass die Gleichgewichte (bzgl. Knoten und Kanten) nach Möglichkeit erhalten bleiben.

Diese Vorgehensweise hat sich bewährt. Es gibt viele konkrete Algorithmen für Coarsening, Partitioning und Uncoarsening. Metis implementiert wie gesagt einige davon. Wir benutzen die ursprüngliche Kombination: *Heavy-Edge-Matching* für das Uncoarsening, Bisektion für die Partitionierung, und *Bound-Kernighan-Lin* für das Uncoarsening. Die Autoren von Metis haben in ihren Publikationen (u.a. [KK99][SKK00b][SKK01]) eindrucksvoll dargelegt, dass diese Kombination für das konkrete Problem der Partitionierung gewichteter Graphen mit Gleichverteilung der Knotengewichte und Minimierung des Edge-Cut (der Gewichte der zerschnittenen Kanten) die beste ist.

Damit also erhalten wir bezüglich unserer Kriterien sehr gute Zerlegungen.

21.2 Konkrete Zerlegungen

Um konkrete Zerlegungen zu erhalten, müssen wir also das Simulationsproblem in ein Netz abbilden, das wir mit Metis partitionieren können. Glücklicherweise ist das Gleisnetz der deutschen Bahn bereits vor-partitioniert, wir müssen also nicht das Netz auf der Ebene der Gleisabschnitte zerlegen: Jedes Stück Gleis ist eindeutig einer *Betriebsstelle* (s.a. Kapitel 25, S. 160) zugeordnet. So hat eines der Beispiele mit denen wir im Projekt SIMONE arbeiten, 7111 Gleisabschnitte mit einer Gesamtlänge von ca. 1000km, die sich auf 104 Betriebsstellen verteilen. Das zu partitionierende Netz bilden wir damit wie folgt:

Knoten: Jede Betriebsstelle ist ein Knoten. Das Gewicht des Knotens ist die Anzahl der Gleisabschnitte in dem Knoten.

Kanten: Wenn zwischen zwei Knoten bzw. Betriebsstellen mindestens ein Zug fährt, werden sie durch eine Kante verbunden. Das Gewicht der Kante ist die Anzahl der Züge, die zwischen den beiden Betriebsstellen verkehren.

Abbildung 21.1 [S. 144] zeigt das konkrete Netz für dieses Beispiel.

Wenn wir dieses Netz nun mit dem Verfahren Metis wie beschrieben auseinander schneiden, erhalten wir eine Partitionierung, die für unsere Anforderungen *optimiert* ist. Unser Anspruch ist hier übrigens zunächst nicht, die wirklich *optimale* Zerlegung zu haben. Eine – nach obigen Kriterien – *gute* Zerlegung lässt zwar hoffen, dass die darauf aufbauende Simulation auch schnell läuft, die *optimale* Zerlegung ist aber keine Garantie für die kürzeste Laufzeit! Deshalb wollen wir keinen zu großen Aufwand in die Zerlegung stecken.

Leider müssen wir die Anzahl der gewünschten Teile ad hoc vorgeben. Auch die wirkt sich schließlich auf die Gesamt-Laufzeit aus, wir werden später an den empirischen Untersuchungen (Kapitel 25, S. 160) noch sehen, wie.

Abbildung 21.2 [S. 145] zeigt eine Zerlegung in 4 Teile, Abbildung 21.3 [S. 145] eine Zerlegung in 16 Teile. Wir benutzen später für die empirischen Untersuchungen unter anderem diese Zerlegungen.

Wir haben auch noch andere Zerlegungen ausgerechnet: für 30 und für 50 Teile. Dort wurden die einzelnen Teile aber so klein, dass Metis nicht mehr allen Teilen einen Knoten zuordnete: statt 30 Teilen gibt es nur 26 und statt 50 Teilen nur 37 mit mindestens einem Knoten. Metis ist offenbar darauf ausgelegt, dass die Anzahl der Knoten des Netzes sehr viel größer ist als die Anzahl der gewünschten Partitionen.

21.3 Verteilung

Für das gegebene Simulationsproblem haben wir also unterschiedliche Zerlegungen berechnet, die sich in der vorgegebenen Anzahl der Simulationsteile unterscheiden. Die Zerlegungen sind zusammen mit den anderen Simulationsdaten in der Datenbank gespeichert. Das Objekt `DisConfig` (Abschnitt 18.5.5, S. 110) liest diese Daten und bereitet sie für den `SimController` (Abschnitt 18.4, S. 107) auf. Der verwendet diese Konfiguration zur Verteilung der Simulationsteile und Steuerung der Simulation (Abschnitt 19.6, S. 121).

Das beschriebene Verfahren ist quasi eine *statische* Konfigurierung. `DisConfig` ist aber so angelegt, dass es auch weniger statisch eine *adaptive Zerlegung* vornehmen kann, abhängig von der gegebenen Rechner-Infrastruktur. Wir können damit die Zerlegung und Verteilung besser der konkret gegebenen Umgebung anpassen. So kann eine schnelle Workstation mit großer Speicherkapazität ein entsprechend größeres Teilproblem bekommen als ein kleinerer Rechner. Dieses Verfahren ist wie gesagt bereits angelegt, wird aber noch detailliert ausgearbeitet von Patrick Mukherjee [Muk04].

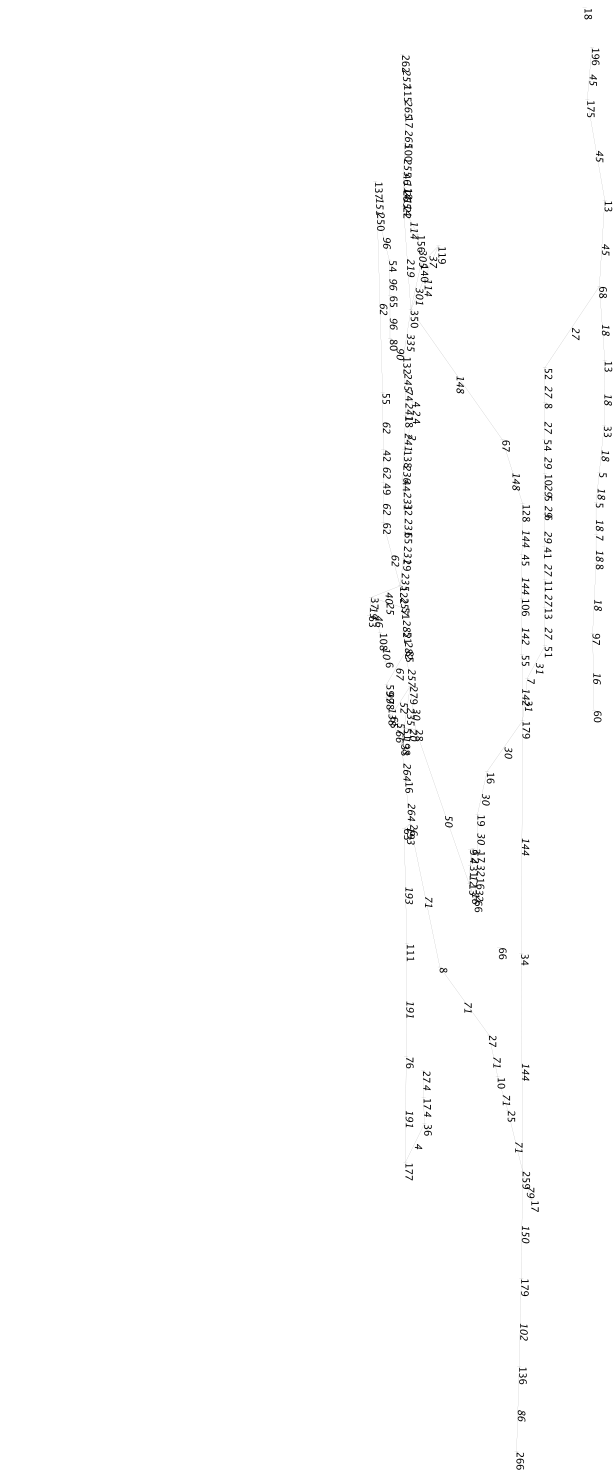


Abbildung 21.1: Das Betriebsstellen-Netz unseres Beispiels. An Knoten und Kanten (kursiv) stehen die Gewichte.

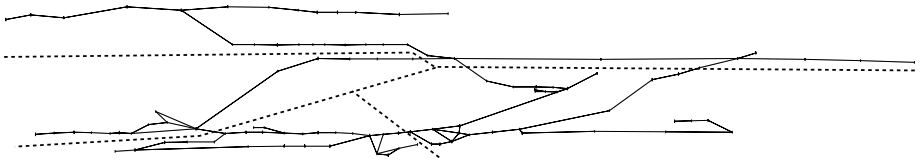


Abbildung 21.2: Zerlegung (gepunktete Linien) des Beispielnetzes in 4 Teile.

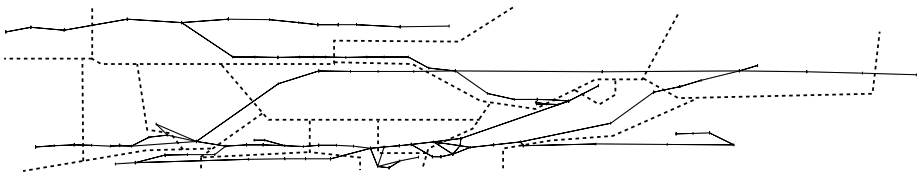


Abbildung 21.3: Zerlegung (gepunktete Linien) des Beispielnetzes in 16 Teile.

22 Externer Simulator

Im Forschungsprojekt SIMONE wird ein sehr komplexer Simulator als CHIP-Prolog-Programm entwickelt. Dieser soll im System DRS die lokalen Simulationen berechnen. Das folgende Kapitel beschreibt unsere generelle Schnittstelle zu Prolog-Programmen und wie damit konkret der externe Simulator eingebunden wird.

Eine der Anforderungen im Projekt SIMONE ist, dass das DRS-System anstatt des Java-Prototypen auch ein externes Programm als lokalen Simulator verwenden kann. Dieses externe Programm ist in CHIP-Prolog [CHI01] programmiert. CHIP besitzt keine Java-Schnittstelle und nur eine sehr rudimentäre C-Schnittstelle. Außerdem unterstützt CHIP keine (interne) Parallelverarbeitung.

Im Design von DRS (Kapitel 18, S. 101) habe ich dafür eine abstrakte Klasse `Simu` vorgesehen, die konkret von `SimuJava` (dem Prototyp) und `SimuProlog` implementiert wird. Letzteres ist die Schnittstelle zum externen Programm und implementiert vor allem die Methoden `reckon(Changes):Changes`, `finish()` und `abort()`.

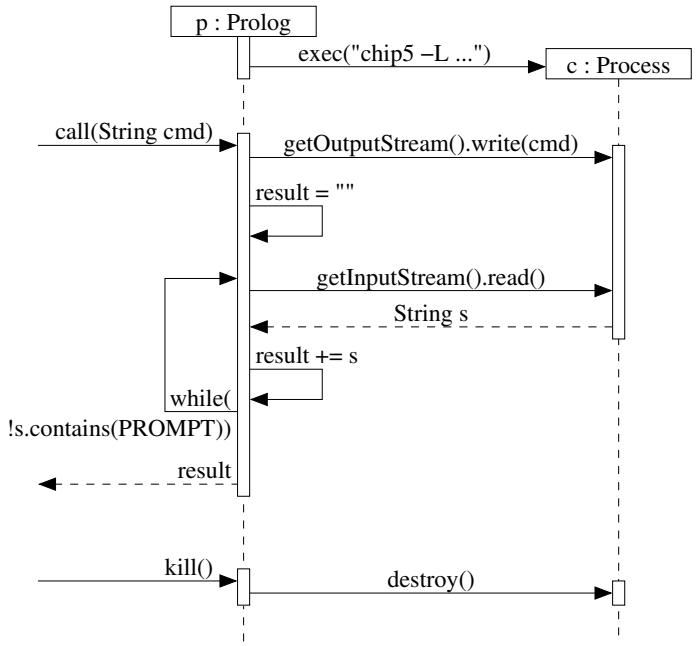


Abbildung 22.1: Sequenz-Diagramm: Prolog-Schnittstelle für die Einbindung externer Simulationsprogramme.

`SimuProlog` benutzt die Klasse `Prolog`: Das ist ein *Wrapper* um einen externen CHIP-Prozess, der diesen initiieren, steuern und beenden kann. Abbildung 22.1 [S. 146] zeigt die Abläufe der Klasse `Prolog`. Beim Anlegen eines `Prolog`-Objekts wird ein externer Prozess gestartet, in dem eine CHIP-Instanz läuft. Über die Kommandozeile zu dem externen Prozess können hierbei CHIP wichtige Laufzeitparameter mitgegeben werden, wie z.B. die Größe des

lokalen oder globalen Stacks etc. Über das `Process`-Objekt (eine Java-Standard-Klasse) hat das `Prolog`-Objekt Zugriff auf Ein- und Ausgabe-Kanäle des Prozesses. Und über diese wird der CHIP-Prozess gesteuert. Wenn vom Java-Programm die Anforderung kommt, im CHIP-Prozess ein Prädikat aufzurufen (`call(cmd)`), dann wird dieses an den Standard-Eingabe-Kanal des CHIP-Prozesses geschickt. Der Standard-Eingabe-Kanal ist übrigens gegenüber dem Java-Programm ein `OutputStream`. Der CHIP-Prozess bearbeitet dann den Aufruf des Prädikats, rechnet (was einige Zeit dauern kann) und gibt, je nachdem wie die Anwendung programmiert ist, während der Berechnung und am Schluss der Berechnung Ergebnisse auf der Standard-Ausgabe aus. Diese Ergebnisse werden vom `Prolog`-Objekt in der Methode `call()` Stück für Stück gelesen, bis der CHIP-Prozess einen vordefinierten Prompt-String ausgibt. Für `Prolog` ist das das Zeichen, dass die Bearbeitung des Prädikats beendet und das Resultat vollständig ist, und dieses von der Methode `call()` an den Aufrufer zurückgegeben werden kann.

Man beachte, dass die Bearbeitung des Prädikats im CHIP-Prozess parallel zur Ausführung der `call()`-Methode stattfindet, die Methode also die `while`-Schleife mehrmals durchläuft, während der CHIP-Prozess rechnet. Das kommt vor allem daher, dass CHIP in einem externen Prozess abläuft und die Aufrufe von `write()` und `read()` praktisch sofort zurückkehren. Das Betriebssystem sorgt im Übrigen dafür, dass die Prozesse CHIP und Java *quasi-parallel* ausgeführt werden.

Ein laufender CHIP-Prozess kann sofort beendet werden, indem `Process.destroy()` aufgerufen wird. Damit können CHIP-Berechnungen bei Problemen wie zu langer Laufzeit abgebrochen werden. Die Ergebnisse, die der Prozess bis dahin gemeldet hat, sind der `call()`-Methode aber weiterhin bekannt und werden von dieser an das aufrufenden Programm zurückgegeben. Dadurch bleiben Zwischen-Ergebnisse des CHIP-Prozesses erhalten.

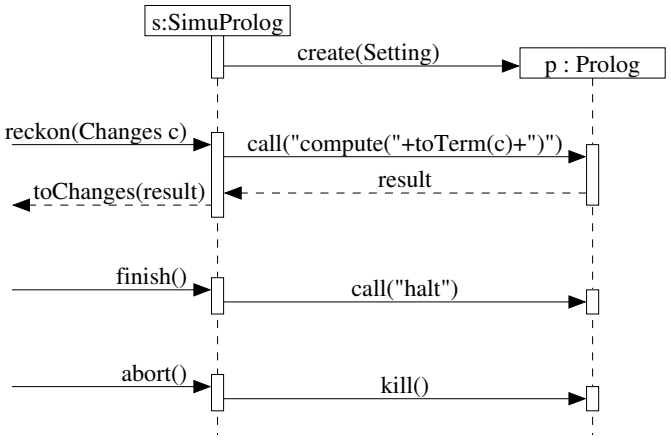


Abbildung 22.2: Sequenz-Diagramm: Steuerung des externen Simulators durch den Wrapper `SimuProlog`.

Abbildung 22.2 [S. 147] zeigt, wie `Prolog` in der Klasse `SimuProlog` benutzt wird (siehe auch die Simulations-Abläufe in Abbildung 19.9 [S. 123]). `reckon(Changes)` wandelt zunächst das `Changes`-Objekt in einen Prolog-Term um (bzw. dessen textuelle Repräsentation, hier vereinfacht durch die Methode `toTerm()` dargestellt) und ruft damit im CHIP-Prozess das Prädikat `compute()` auf. Wenn CHIP mit der Berechnung fertig ist, gibt es als Resultat wieder die textuelle Repräsentation eines `Changes`-Objekts zurück. Dieses muss in ein Java-Objekt vom Typ `Changes` umgewandelt werden (hier: `toChanges()`), welches an den Aufrufer (`Worker`) zurückgegeben wird. Die normale Beendigung der Simulation per `finish()` führt zur Beendigung des CHIP-Prozesses. `abort()` löst einen sofortigen Abbruch des CHIP-Prozesses aus.

Wir haben also mit der Klasse `Prolog` eine Java-Schnittstelle zu *beliebigen* Prolog-Systemen entwickelt. Mit ihr können solche Systeme in externen Prozessen gestartet werden, es ist sogar möglich, mehrere solche Prozesse gleichzeitig zu behandeln. Zur Steuerung dieser Prozesse werden von Java aus Prolog-Prädikate aufgerufen, deren *Ergebnis* – die textuelle Ausgabe während der Ausführung – wird als `String` zurückgegeben. Fehlerhafte Prolog-Programme können abgebrochen werden, wobei Zwischenergebnisse erhalten bleiben. Die Abstraktion `Simu` bietet die Möglichkeit einer Einbindung unterschiedlicher Simulatoren, `SimuProlog` ist eine davon, und realisiert unter Verwendung der Klasse `Prolog` die Einbindung des externen Simulators.

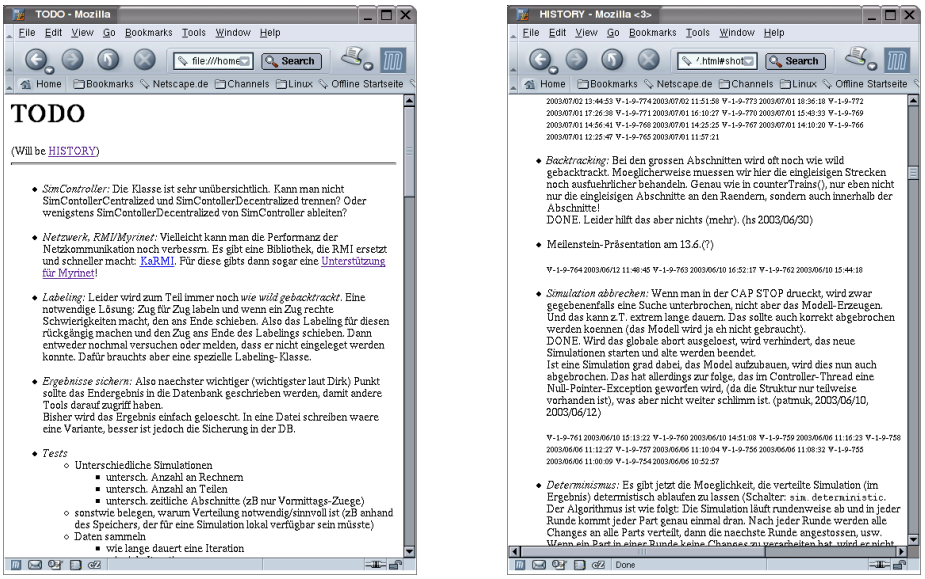
23 Entwicklungsprozess

Der in DRS integrierte Entwicklungsprozess erlaubt unter anderem den Austausch wichtiger Komponenten während der Laufzeit des Gesamt-Systems. Zusammen mit einer Aufgaben-Verwaltung, Versions-Management und automatischer Versions-Erstellung vereinfacht das deutlich die verteilte Entwicklung.

DRS ist ein verteiltes System und als solches erheblich schwieriger zu managen und zu entwickeln als ein monolithisch lokales System. In der Entwicklungsphase – und die ist gerade bei Forschungs- und Entwicklungsprojekten sehr lang – werden laufend neue Funktionen implementiert und vorhandene verbessert. Es entstehen also oft neue Versionen. Bei einem lokalen System kann man eine solche Version lokal erstellen und testen, indem man es lokal startet und beendet. Bei verteilten Systemen aber ist das nicht so einfach: Man muss da Komponenten des verteilten Systems austauschen, die irgendwo im Netz liegen und möglicherweise gerade von anderen Entwicklern oder Anwendern benutzt werden. Außerdem sind gerade an der Entwicklung von DRS mehrere Personen beteiligt.

Wir haben uns deshalb für die Entwicklung eines integrierten Entwicklungsprozesses entschieden. Die wesentlichen Anforderungen waren: Aufgabenverwaltung, Versionskontrolle, automatisierte Versionsgenerierung, verteilte Entwicklung und Online-Komponenten-Update.

23.1 Aufgabenverwaltung



(1) (2)

Abbildung 23.1: Todo-Liste (1) und Logbuch (2).

Welche Dinge noch zu tun sind und welche wann mit welchem Ergebnis erledigt wurden, sollte dokumentiert sein. Wir führen deshalb eine Todo-Liste, in

die eingetragen wird, welche Fehler aufgetaucht sind und welche sonstigen Verbesserungen noch zu tun sind. Wenn eine solche Aufgabe erledigt ist, kommt ihre Beschreibung von der Todo-Liste in eine Art Logbuch, zusammen mit der Beschreibung der Lösung und der Versionsnummer. Letztere wird von unserem Entwicklungswerkzeug automatisch generiert. Sowohl Todo-Liste als auch Logbuch werden als einfache HTML-Dateien geführt, in die jeder Entwickler Einträge macht. Abbildung 23.1 [S. 149] zeigt, wie das konkret aussieht.

23.2 Versionskontrolle

Aller Programm-Code muss verwaltet werden, insbesondere müssen alte Versionen wiederhergestellt werden können. Seit einiger Zeit hat sich dafür CVS etabliert, dieses System wird in fast allen Open-Source-Projekten im Internet verwendet. Ich habe in Abschnitt 18.9 [S. 115] CVS schon kurz als Teil von DRS dargestellt.

Im Projekt DRS hat also jeder Entwickler seine eigene Kopie des Programm-Codes, ein gemeinsames Repository dient der Synchronisation der unterschiedlichen Kopien und der Speicherung der Geschichte des Codes.

23.3 Automatische Versions-Erstellung

Wir benutzen für die Erstellung neuer Laufzeit-Versionen von DRS das System *Ant*, das ich in Kapitel 19 [S. 116] schon kurz dargestellt habe. Damit kann jeder Entwickler vollständig automatisiert neue Versionen erstellen. Wir haben für jede Komponente (DIR, SIM, etc.) und das übergeordnete Gesamtsystem DRS eigene Verzeichnisse, in denen eigene Definitionen vor allem für die *Ant-Tasks* `build` und `start` existieren.

Mit folgendem Kommando wird die Komponente WRK (im Verzeichnis `PRJDRS/WRK`) neu erzeugt:

```
hs@rubens:~/PRJDRS/WRK> ant clean build
```

Die Definition der entsprechenden Aufgaben `build` und `clean` in Abbildung 23.2 [S. 150] zeigt, dass dabei durchaus komplexe Dinge geschehen.

```
<target name="build" depends="time,compile,rmi,jar,distribute,war">
  <copy file="prjdrs/wmg/WMGServlet.class" overwrite="true"
     todir="${basedir}/CATALINA_BASE/webapps/wmg/WEB-INF/classes...
</target>
<target name="clean" depends="delete" />
<target name="upload">
  <java classname="${srcdom}.${srcpkg}.Upload" fork="true">
    <arg file="${basedir}/${webapp}.war"/>
    <classpath refid="project.class.path"/>
  </java>
</target>
```

Abbildung 23.2: Ausschnitt aus `PRJDRS/WRK/build.xml`.

Um eine neue Laufzeitversion zu *bauen*, wird also zunächst ein Zeitstempel erstellt (`time`), dann werden die veränderten Quell-Dateien übersetzt (`compile`),

anschließend werden einige Dateien für die RMI-Technik erzeugt (`rmi`), dann einige JAR-Archive erstellt (`jar`), diese auf verschiedene Verzeichnisse verteilt (`distribute`), und schließlich wird ein so genanntes *WAR*-Archiv erstellt (`war`), das für das Online-Update, siehe Abschnitt 23.5 [S. 152], benötigt wird. Alle Unteraufgaben (`time` etc.) sind übrigens wieder als Ant-Tasks realisiert, die hier aber nicht gezeigt sind. Der Task `clean` ruft einen anderen Task `delete` auf. Und `upload` benutzt eine eigene Klasse `Upload`, die eine neue Version in den DIR hochlädt, siehe Abschnitt 23.5 [S. 152].

```
<target name="commit" depends="destination.tmp,export">
  <ant dir="${destination}/PRJDRS/WRK" target="upload"/>
  <delete dir="${destination}" failonerror="false" />
  <echo message="Commit: updated DRS-Worker to ${version}!"/>
</target>
<target name="export" depends="time,cvsroot,cvstag,destination">
  <cvs cvsRoot="${cvsroot}"
       command="export -r ${version} PRJDRS"
       dest="${destination}" />
  <ant dir="${destination}/PRJDRS" target="build"/>
  <copy file="${destination}/PRJDRS/WRK/wrk.war"
        overwrite="true"
        todir="${destination}/PRJDRS/DIR/CATALINA_BASE/webapps/...
  <echo message="Exported PRJDRS ${version} to ${destination}!"/>
</target>
<target name="cvstag" depends="cvsroot,cvscommit,version,history">
  <cvs cvsRoot="${cvsroot}" command="tag -c ${version}"  />
</target>
```

Abbildung 23.3: Ausschnitt aus `PRJDRS/build.xml`.

Unter Verwendung derselben Mechanismen kann ein Entwickler mit folgendem einfachen Kommando seine Änderungen in das *laufende* System einbringen:

```
hs@rubens:~/PRJDRS> ant clean build commit
```

Abbildung 23.3 [S. 151] zeigt, was dann alles passiert: Zunächst muss ein temporäres lokales Verzeichnis erstellt werden (`destination.tmp`). In dieses wird eine neue Version exportiert (`export`). Dafür muss wieder ein Zeitstempel erstellt werden (`time`). Dann wird die aktuelle Version im CVS gekennzeichnet (`cvstag`). Hierfür muss zunächst der Speicherort des CVS ermittelt werden (`cvsroot`), dann wird die aktuelle Arbeitskopie in das zentrale Repository kopiert (`cvscommit`), anschließend wird eine neue Versionsnummer generiert (`version`), diese wird in die History-Datei (Logbuch) eingetragen (`history`), und dann wird per CVS-Kommando das zentrale Repository mit der neuen Versionsnummer gekennzeichnet. Damit kann diese Version später immer wieder konsistent hergestellt werden. Dann wird (wieder im Task `export`) das CVS-Kommando `export` aufgerufen, das aus dem zentralen Repository eine konsistente Version ins temporäre Verzeichnis kopiert, dort übersetzt (`build`), und schließlich das resultierende WAR-Archiv in die WRK-Komponente kopiert. Schließlich wird (jetzt wieder im Task `commit`) über den Sub-Task `upload` in der Komponente WRK die neuste Version zum DIR hochgeladen, der diese auf alle Worker verteilt. Wie das genau geht, steht in Abschnitt 23.5 [S. 152].

23.4 Verteilte Entwicklung

Wir haben gerade gesehen, wie die automatische Versions-Erzeugung das CVS benutzt, um jeweils die neuste konsistente Programmversion zu bekommen, diese zu übersetzen und in das laufende System einzupflegen.

CVS unterstützt darüber hinaus verteilte Entwicklung: Entwickler können an unterschiedlichen Orten jeweils an ihrer eigenen Programmkopie arbeiten und, immer wenn diese einen brauchbaren Zustand hat (als Minimal-Anforderung gilt hier in der Regel, dass das komplette System fehlerlos übersetzt werden kann), ihre Kopie mit dem zentralen Repository in Einklang bringen. Sie können dann sogar die neuste Version in das laufende System einbringen, alles von ihrem Arbeitsplatz aus.

23.5 Online-Update

Wir benötigen die Möglichkeit, ein laufendes System zumindest in wesentlichen Teilen erneuern zu können, ohne es dafür beenden zu müssen, also eine *Online-Update*-Möglichkeit. Wir haben ja schon gesehen, dass der Administrator den DIR herunterfahren kann, ohne dass das die Worker belastet, weil sich diese regelmäßig beim DIR melden. Damit kann man den DIR auf dem Server beenden, durch eine neue Implementierung ersetzen und dort neu starten. Nach etwa einer Minute – dem Intervall, in dem sich die Worker beim DIR melden – ist das System dann wieder komplett. Ähnlich kann man auf den Client-Rechnern einfach eine neue Version starten.

Wichtig und schwieriger ist es, laufende Versionen auf den im Netz verteilten Workern zu aktualisieren. Wir haben dafür eine automatische Update-Möglichkeit entwickelt. Diese nutzt die Fähigkeiten von Tomcat, *Web-Services* auszutauschen.

Wir haben schon weiter oben in Abbildung 23.3 [S. 151] gesehen, dass der Task `commit` eine Teilaufgabe `upload` hat, und in Abbildung 23.2 [S. 150], dass diese Teilaufgabe eine Klasse `Upload` benutzt. Abbildung 23.4 [S. 153] zeigt den genauen Ablauf beim Programm-Update. Die Klasse `Upload` schickt dem `DirectoryService` die Datei `wrk.war`, also den kompletten neuen Worker, verpackt in ein WAR-Archiv. Der `DirectoryService` ruft dann auf allen Workern, deren Zustand `w.state == IDLE` ist, `triggerUpdate()` auf. Das löst dort schlicht einen Aufruf von `register()` aus. Wir haben in Abschnitt 19.4 [S. 119] gesehen, dass sich die Worker regelmäßig am DIR per `register()` anmelden, und damit auch die Worker, die beim explizit ausgelösten Update gerade nicht `IDLE` sind, sondern eine Simulation rechnen. Irgendwann also werden alle Worker aktualisiert, indem sie sich beim DIR registrieren.

Der DIR überprüft nämlich jedes Mal, ob die Version eines sich registrierenden Workers noch aktuell ist oder aber *kleiner* als die aktuelle Version. Und wenn die Version nicht mehr aktuell ist, erzeugt der `DirectoryService` einen `Thread uw`, der das Update des Workers `w` durchführt. Er veranlasst den `Tomcat t`, der ja hier als Laufzeitumgebung für den Worker fungiert, den *Web-Service* des Workers zu entfernen (`remove()`). Der Tomcat Web-Server läuft ja immer, wie wir wissen, es wird für das Update des Workers nur die Komponente im laufenden Tomcat ausgetauscht, die den eigentlichen Worker realisiert.

`t` fährt dann den Worker herunter (`destroy()`), der sich ordnungsgemäß am DIR abmeldet. Damit ist der laufende Worker `w` beendet. Dann löst der

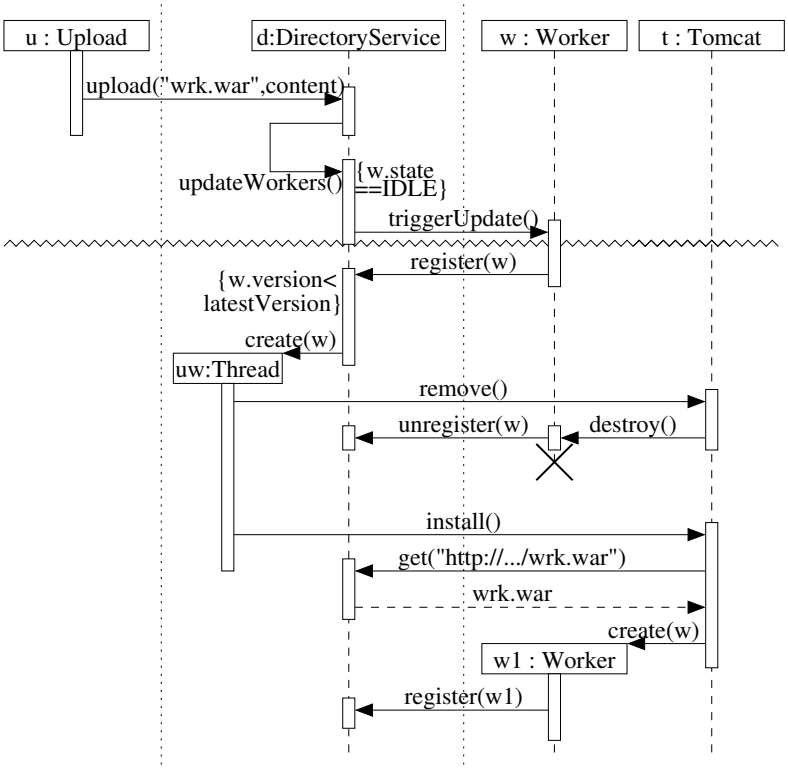


Abbildung 23.4: Ablauf Programm-Update. Der Teil oberhalb der Zickzack-Linie ist ein *möglicher* Auslöser für das eigentliche Update unterhalb der Linie, entscheidend ist der Aufruf von `register()`, der ja auch andere Ursachen haben kann (Abschnitt 19.4, S. 119).

Thread `uw` beim Tomcat ein `install()` aus, woraufhin `t` sich beim zentralen DIR die neue Version holt (`get("http://.../wrk.war")` und diese als neue Web-Applikation (oder Web-Service) startet. Der Tomcat benötigt übrigens die neue Web-Applikation in einem WAR-Archiv, deshalb wird ursprünglich ein solches erzeugt. Das Starten der neuen Web-Applikation führt dazu, dass ein neues `Worker`-Objekt erzeugt wird, also ein neuer Worker, der sich wieder beim DIR anmeldet (`register()`) und damit anstatt des alten Workers `w` im laufenden DRS-System zur Verfügung steht.

24 Zusammenfassung und Diskussion

Dieser Teil hat ausführlich die Realisierung des DRS-Systems dargestellt. Dabei wurde das Design anhand der System-Architektur und der internen Struktur vor allem der Komponenten DIR, CAP, SIM und WRK erläutert. Die unterschiedlichen Abläufe im System wurden anhand von UML-Sequenz-Diagrammen und auslösender Ereignisse der Kontroll-Applikation CAP dargestellt. Im Kapitel zur zentralen Kontrolle wurde vor allem formal gezeigt, warum die Realisierung den Algorithmus korrekt umsetzt. Wir haben gesehen, wie Simulationsprobleme konkret zerlegt und auf die Rechenknoten verteilt werden. Es wurde beschrieben, wie der im Projekt SIMONE entstehende externe Simulator eingebunden wird, und schließlich, warum wir einen eigenen Entwicklungsprozess benötigen und wie dieser ins System integriert ist. Das folgende Kapitel geht nochmal auf die in der Einführung definierten Motivationen und Probleme der verteilten Problemlösung ein.

In der Einführung hatte ich genau erläutert, welche Vorteile (Abschnitt 2.1, S. 7) der Einsatz verteilter Problemlösung bringt und welche Probleme (Abschnitt 2.2, S. 9) dabei zu lösen sind. Im Folgenden möchte ich die Realisierung von DRS anhand dieser Kriterien beurteilen.

24.1 Vorteile

Performanzgewinn

Der Performanzgewinn wird zum einen durch Problemzerlegung und damit erhebliche Reduktion der Komplexität der lokalen Simulationsprobleme, zum anderen durch die Ausnutzung verteilter Rechenkapazitäten erreicht. Dass daraus wirklich ein Gewinn resultiert, werde ich später in Kapitel 25 [S. 160] empirisch zeigen.

Fehlertoleranz und Ausfallsicherheit

Wir haben bei der Implementierung immer darauf geachtet, dass alle Komponenten möglichst unabhängig von Fehlern oder Ausfällen anderer sind. Wir haben sogar erste Ansätze realisiert, den kompletten Ausfall eines oder mehrerer Rechenknoten *während einer Simulation* zu verkraften. In diesem Fall würden die Aufgaben des ausgefallenen Rechners von anderen Knoten übernommen. Diese Ansätze sind aber noch etwas rudimentär.

Insgesamt ist die Implementierung aber mittlerweile sehr stabil. So konnten wir 900 Simulationen hintereinander auf einer laufenden WRK-Instanz durchführen, ohne dass diese hätte neu gestartet werden müssen, und ohne dass sich dort die Performanz verschlechtert hätte. DIR-Instanzen laufen typischerweise viele Wochen. Das ist vor allem auch der inkrementellen Erweiterbarkeit des Systems zu danken.

Inkrementelle Erweiterbarkeit

Das *Online-Update* erlaubt den Austausch wichtiger Komponenten zur Laufzeit des Gesamtsystems. Das erleichtert enorm die Entwicklung des Systems, wir können vor allem die Komponenten online austauschen, die sich während der Entwicklung besonders oft ändern. Wie gesagt trägt dieser Mechanismus auch zur Stabilität des Systems bei.

Örtliche Bereitstellung von Rechenleistung

Die Kontroll-Applikation CAP kann auf beliebigen Rechnern laufen, die mit dem laufenden DRS-System durch ein einfaches Standard-Netzwerk verbunden sind. Die Rechenknoten selbst können über ein großes Netz verteilt sein. Wir haben gerade bei der verteilten Kontrolle darauf geachtet, dass zwischen beliebigen Komponenten sehr wenig kommuniziert werden muss. Damit kann bei der Simulation per DRS eine große Menge an Rechenkapazität von fast jedem Internet-Arbeitsplatz genutzt werden.

Datensicherheit

Wie in Abschnitt 2.3.5 [S. 17] schon angedeutet, habe ich diesen Punkt bisher fast völlig außer Acht gelassen. Im Moment verlassen wir uns da auf die Komponenten Java und Tomcat. Allerdings sind diese relativ gut geschützt, schließlich wurden sie von vornherein für den Einsatz im Internet konzipiert. Für DRS selbst gibt es noch kein eigenes Sicherheitskonzept.

Kosteneffizienz

Ein Grundkonzept dieser Arbeit ist, auf Standard-Komponenten zu setzen, vor allem auch bei der benötigten Hardware. DRS läuft sehr gut auf aktuellen PCs, die allerdings nicht zu spärlich ausgestattet sein sollten. Auch zur Vernetzung genügt moderne Standard-Hardware völlig. Siehe hierzu auch den Vergleich mit Super-Computern in Abschnitt 2.1 [S. 7].

24.2 Probleme

Die in Abschnitt 2.2 [S. 9] definierten möglichen Probleme verteilter Systeme sind wie folgt *gelöst*.

Informationsdefizit

Der zentrale Informationsdienst DIR verwaltet die gemeinsamen Status-Informationen für die Worker, Kontrollanwendungen und Simulationen. Tatsächlich ist die Adresse des DIR die einzige Konfigurations-Information, die jede CAP oder jeder WRK beim Start benötigt. Alles andere wird über den gemeinsam bekannten DIR kommuniziert.

Inkonsistenzen

Ein weiteres Informationsdefizit besteht während der Simulation in den möglichen Inkonsistenzen zwischen lokalen Simulationsergebnissen. Dieses wird durch den Algorithmus DRS behoben.

Nichtdeterminismus und Determiniertheit

Das Verfahren DRS kann asynchron betrieben werden, um die vorhandene verteilte Rechenkapazität optimal auszunutzen. Das Verfahren ist dann nicht-determiniert. Die zentrale Kontrolle kann die verteilten Berechnungen auch synchronisieren, womit das gesamte Verfahren determiniert wird.

Terminierung

DRS konvergiert immer, wie ich in einer aufwändigen Untersuchung in Teil II [S. 51] gezeigt habe. Der verteilt vorliegende Endzustand wird entweder zentral oder dezentral erkannt (Kapitel 20, S. 130).

Kontrolle und Verantwortlichkeiten
Wie haben zwei verschiedene Kontroll-Möglichkeiten realisiert: zentrale und dezentrale Kontrolle. Die zentrale ist einfacher, hat aber das Problem, dass alle Knoten immer und ausschließlich mit der Zentrale kommunizieren müssen. Die dezentrale Kontrolle arbeitet auf einem verteilten Terminierungs-Baum und behebt das Kommunikationsproblem (s.a. Abschnitt 25.6, S. 170).

Konfiguration
Die Konfiguration der Problem-Zerlegung und -Verteilung wird durch die abstrakte Klasse `DisConfig` realisiert. Zur Zeit haben wir dafür nur die Implementierung der statischen Zerlegung in vordefinierte Teile und der trivialen Verteilung auf Worker. Bessere Zerlegungen und Verteilungen erwarten wir durch die *adaptive Konfiguration* ([Muk04]).

Kommunikation
Die Kommunikation der verteilten Komponenten wird fast ausschließlich auf einer relativ abstrakten Ebene per Java-RMI (Abschnitt 19.1, S. 117) gemacht. Diese Technik hat sich als sehr anwendungsfreundlich (starke Typisierung, brauchbare Performanz) erwiesen. Nur an zwei Stellen stand sie uns nicht zur Verfügung: beim Online-Update und bei der Einbindung des externen Simulators.

Algorithmen und Programmierung
Verteilte Systeme sind besonders schwierig zu entwerfen und zu implementieren, insbesondere bezüglich algorithmischer Korrektheit. Wir haben deshalb auf ein sauberes Design geachtet. Vor allem aber habe ich die Korrektheit der Algorithmen und der Implementierung formal gezeigt.

Heterogenität
Zum einen unterstützen wir mit DRS heterogene verteilte Systeme indem wir als Laufzeitumgebung Java benutzen und damit relativ Plattform-unabhängig sind. Zum anderen ist keine spezielle Hardware nötig, DRS läuft sehr gut auf allen kompatiblen Standard-Komponenten. Außerdem kann sogar der Simulations-Kern aus einer heterogenen, externen Komponente bestehen, die einfach *einzustecken* ist.

Bewertung
Zur empirischen Bewertung der Laufzeit-Eigenschaften des Systems haben wir ein `Statistics`-Modul entwickelt, mit dem man dezentral Daten sammeln und schließlich zentral verdichten kann. Darüber hinaus können die verteilten Komponenten in Web-Oberflächen etwa zum Debugging überwacht werden.

Teil IV

Ergebnisse

Dieser Teil setzt den Schlusspunkt der vorliegenden Arbeit. Hier werden zunächst der Algorithmus und die Realisierung empirisch evaluiert: Das Kapitel 25 [S. 160] beleuchtet am Beispiel einer Fallstudie die Laufzeit-Eigenschaften von ganz unterschiedlichen Seiten.

Zusammenfassung (Kapitel 26, S. 177) und Ausblick (Kapitel 27, S. 179) runden die Arbeit ab.

25 Fallstudie

Hier sollen die Fähigkeiten des Ansatzes und der Implementierung empirisch untersucht werden. Dazu wird zunächst das Fallbeispiel vorgestellt, das im Rahmen des Projekts SIMONE zur Verfügung stand. Darauf aufbauend werden die Laufzeit-Eigenschaften von DRS evaluiert. Dazu haben wir Hunderte von Tests durchgeführt, wobei folgende Parameter variiert wurden: Größe des Simulationsproblems, Anzahl verwendeter Rechenknoten, Art der Partitionierungen, deterministischer und nichtdeterministischer Modus, zentrale und verteilte Kontrolle, und der verwendete lokale Simulator.

Im letzten Abschnitt des Kapitels werden die Laufzeit-Ergebnisse komprimiert bewertet.

25.1 Fallbeispiel

	Gesamt	Montag	Deutschland
Gleislänge [km]	1006,503		65005
Betriebsstellen (BST)	104		
∅ Gleislänge / BST [km]	9,678		
Gleisabschnitte	7111		
∅ Länge / Gleisabschnitt [km]	0,142		
∅ Gleisabschnitte / BST	68		
∅ Züge / Tag	795	781	34950
∅ Züge / BST	123		
∅ BST / Zug	11		
∅ Gleisabschnitte / Zug	266		
∅ Laufweg / Zug [km]	43,509	37,844	
∅ Laufweg / Tag [km]	34583,981	29556,384	

Tabelle 25.1: Das Fallbeispiel im Überblick.

Wir hatten im Projekt SIMONE einige Beispieldaten zur Verfügung, die alle auf realen Daten der deutschen Eisenbahn basieren. Die folgende Fallstudie basiert auf unserem *großen Fallbeispiel*. Tabelle 25.1 [S. 160] zeigt dieses im Überblick. Es besteht aus mehr als 1000 Kilometer Gleis, die sich auf 104 *Betriebsstellen* verteilen. Eine *Betriebsstelle* ist eine vom Bahnbetreiber definierte Menge von Gleisen, oft bezieht sich diese auf einen Bahnhof oder ein Stellwerk. Das deutsche Netz ist disjunkt in Betriebsstellen aufgeteilt. Die durchschnittliche Gleislänge pro Betriebsstelle in unserem Beispiel ist demnach 9,678 Kilometer.

Das Gleisnetz besteht aus 7111 *Gleisabschnitten*. Wie weiter oben in Abschnitt 6.1 [S. 47] schon angedeutet, sind die *Gleisabschnitte* begrenzt durch Signale, Zugschlusstellen, Weichen, Geschwindigkeitswechsel, Steigungsänderungen oder ähnliches. Das heißt, wenn am Gleisnetz ein Signal steht, endet ein

Gleisabschnitt und ein neuer beginnt. Die durchschnittliche Länge pro Gleisabschnitt ist (im Beispiel) 142 Meter. Jede Betriebsstelle besteht im Schnitt aus 68 Gleisabschnitten.

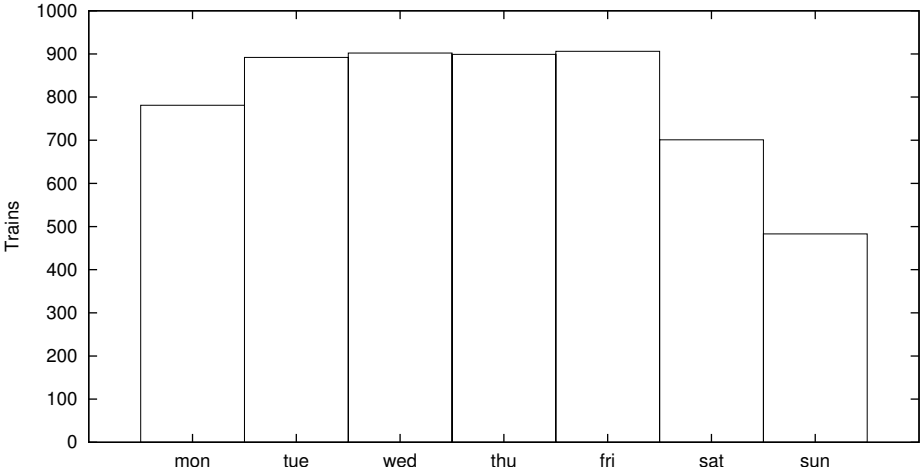


Abbildung 25.1: Gesamtzahl der Züge im Fallbeispiel an den unterschiedlichen Wochentagen.

Das Fallbeispiel enthält neben den Daten zur Infrastruktur natürlich auch Daten zum Fahrplan. Dieser definiert insgesamt 1118 Züge. Nicht jeder dieser Züge verkehrt aber an jedem Tag, einige fahren nur Montag bis Freitag oder nur sonntags. Im Schnitt definiert der Fahrplan für jeden beliebigen Tag der Woche 795 Züge. Abbildung 25.1 [S. 161] zeigt, wie unterschiedlich die Anzahl der Züge an den verschiedenen Wochentagen ist. Am Montag des Beispiels fahren 781 Züge (Spalte *Montag* in Tabelle 25.1, S. 160). Natürlich ist die Anzahl der Züge an einem Tag auch nicht gleichverteilt, in der Nacht fahren viel weniger Züge als zu den Stoßzeiten morgens und abends. Abbildung 25.1 [S. 161] gibt einen Überblick über die Verteilung der Züge (im Beispiel) über den Tag.

Durch jede Betriebsstelle fahren im Schnitt 123 Züge, jeder Zug befährt durchschnittlich 11 Betriebsstellen und 266 Gleisabschnitte. Die Züge fahren damit im Schnitt 43,509 Kilometer, alle zusammen an einem durchschnittlichen Tag im Beispiel fast 34600 Kilometer. Am Montag fahren die Züge im Schnitt 37,844 Kilometer und zusammen genau 29566,384 Kilometer.

Das Beispiel beschreibt natürlich nur einen Ausschnitt des gesamten deutschen Eisenbahn-Systems. Zum Vergleich: Das Gesamtsystem umfasst 65005 Kilometer Gleis und im Schnitt 34950 Züge pro Tag [Deua].

Abbildung 25.3 [S. 162] zeigt das vollständige Gleisnetz des Fallbeispiels, bestehend aus den 7111 Gleisabschnitten. In Abbildung 25.4 [S. 162] sehen wir entsprechend das gröbere Netz der Betriebsstellen (s.a. Abbildung 21.1, S. 144). Die Position der Gleisabschnitte bzw. Betriebsstellen entspricht übrigens nicht genau der Wirklichkeit, sie sind vom Bearbeiter des Netzes so angeordnet, dass er in seinem Werkzeug mit diesem Netz gut hantieren kann.

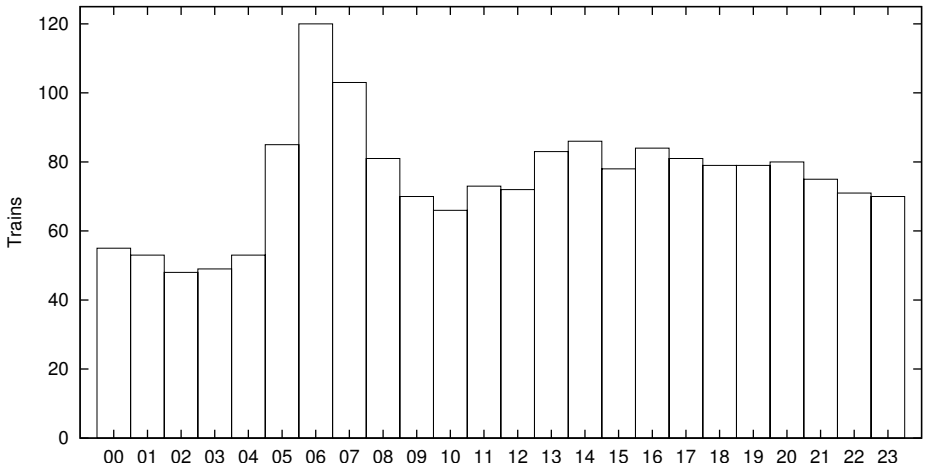


Abbildung 25.2: Gesamtzahl der Züge im Fallbeispiel, aufgetragen über der Tageszeit.

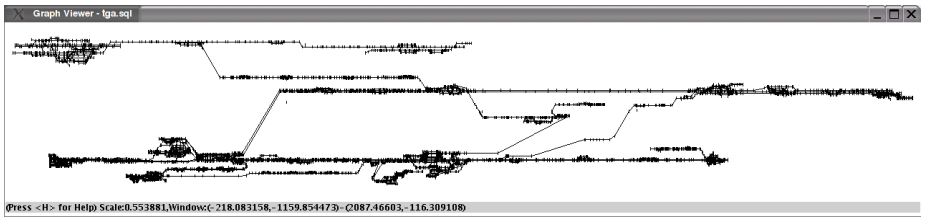


Abbildung 25.3: Gleisabschnitts-Netz des Fallbeispiels.

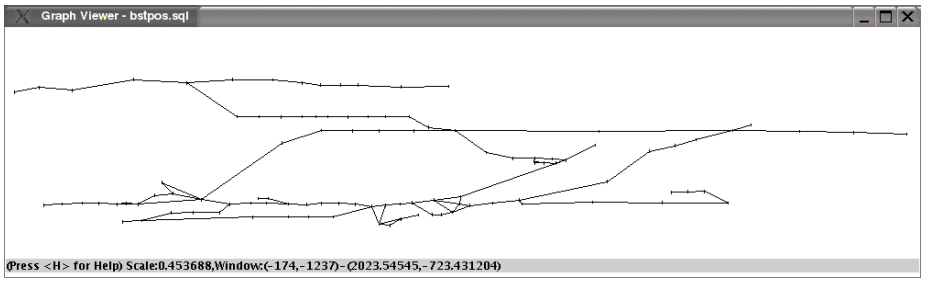


Abbildung 25.4: Betriebsstellen-Netz des Fallbeispiels.

25.2 Problemgröße

Wie wirkt sich die Größe des Simulationsproblems auf das Laufzeitverhalten von DRS aus? Wir haben unterschiedliche Tests gemacht, wobei wir sowohl die Netzgröße als auch die Anzahl der darauf fahrenden Züge variiert haben.

25.2.1 Anzahl der Züge

Zeitscheibe	Züge	Dichte [Züge/Std]
03-09	189	31,5
09-15	233	38,8
15-21	249	41,5
21-03	121	20,2
03-15	404	33,7
03-21	641	35,6
03-03	741	30,9

Tabelle 25.2: Zeitscheiben (jeweils erste und letzte Stunde aus dem Montags-Fahrplan, also etwa alle Züge, die montags zwischen 03:00 und 09:00 Uhr verkehren) und Anzahl der enthaltenen Züge.

Um die Anzahl der Züge zu variieren, haben wir aus allen Fahrplandaten zunächst die Züge genommen, die an einem normalen Montag fahren und daraus dann verschiedene Zeitscheiben. Tabelle 25.2 [S. 163] zeigt die Zeitscheiben und wie viele Züge in jeder einzelnen unterwegs sind. In der Zeitscheibe 03-15 befinden sich übrigens mit 404 Zügen weniger als die Summe der Zeitscheiben 03-09 und 09-15 unterwegs (189+233=422), weil einige Züge sowohl vor als auch nach 09:00 Uhr fahren. Die ersten vier Zeitscheiben sind jeweils gleich lang (6 Stunden), haben aber unterschiedlich viele Züge. Dort ist deshalb auch die Dichte relativ unterschiedlich. Die größeren Zeitscheiben enthalten vor allem deutlich mehr Züge.

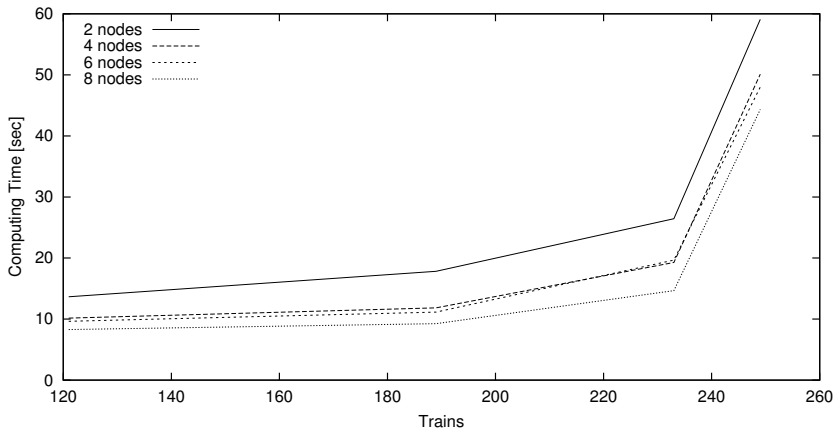


Abbildung 25.5: Laufzeiten in Abhängigkeit der Anzahl der Züge.

Abbildung 25.5 [S. 163] zeigt Gesamt-Laufzeiten der Simulation in Abhängigkeit der Anzahl der Züge. Wir haben für diese Simulationen (und die meisten folgenden) den Java-Simulator-Prototypen verwendet, nicht den externen SIMONE-Simulator. Der Prototyp war einfach leichter zu handhaben, als der externe CHIP-basierte Simulator, siehe hierzu die Vergleiche weiter unten in Abschnitt 25.7 [S. 172]. Außerdem haben wir hier zunächst nur den zentralen, synchronisierten (und damit deterministischen) Modus verwendet, und die Zerlegung in 26 Teile, die sich als die im Schnitt beste heraus gestellt hat, wie wir später noch sehen werden. In Abbildung 25.5 [S. 163] wurde also die Anzahl der Züge variiert und die Anzahl der Rechenknoten: die vier Kurven beschreiben die Laufzeiten für 2, 4, 6 und 8 Rechner.

Jeder Rechner ist ausgestattet mit 1GB Hauptspeicher und 2 AMD K7 Prozessoren mit 1,2 GHz Taktfrequenz. Die Rechenknoten sind also Doppel-Prozessor-Systeme! Auf jedem Rechner läuft Linux 2.4 und das Java-Laufzeitsystem J2SE 1.4.0 von Sun [Jav].

Die Vergleiche zeigen, dass die Gesamtlaufzeit nicht linear von der Anzahl der Züge abhängt. Kleinere Probleme aber skalieren durchaus: Die Laufzeit für 121 Züge auf 2 Knoten ist etwa dieselbe wie die für knapp 233 Züge auf allerdings 8 Knoten. Wenn die Probleme größer werden, skaliert das Verfahren nicht mehr so schön, hier eben ab 249 Zügen.

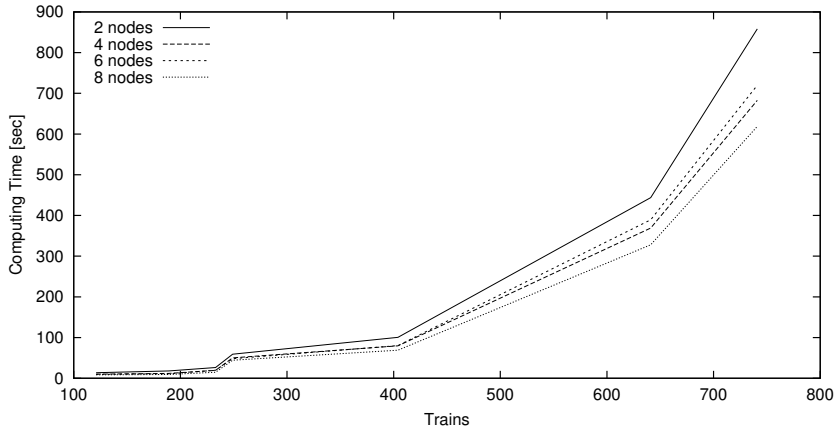


Abbildung 25.6: Laufzeiten in Abhängigkeit der Anzahl der Züge.

Insbesondere bei noch größeren Problemen ist die Laufzeit nicht mehr linear von der Problemgröße abhängig: Abbildung 25.6 [S. 164] zeigt Laufzeiten für alle Zeitscheiben, angeordnet nach der Anzahl der Züge (121-741).

Allerdings ist die Problem-*Komplexität* nicht linear abhängig von der Problem-*Größe* (z.B. Anzahl der Züge oder der Gleisabschnitte) abhängig, schließlich ist ein Teil des Simulations-Problems ein Job-Shop-Scheduling-Problem und das ist NP-vollständig (Abschnitt 3.3, S. 23).

25.2.2 Netzgröße

Zusätzlich zur Anzahl der Züge haben wir auch die Netzgröße variiert. Abbildung 25.7 [S. 165] zeigt Laufzeiten für das ganze Netz und für ca. zwei Drittel

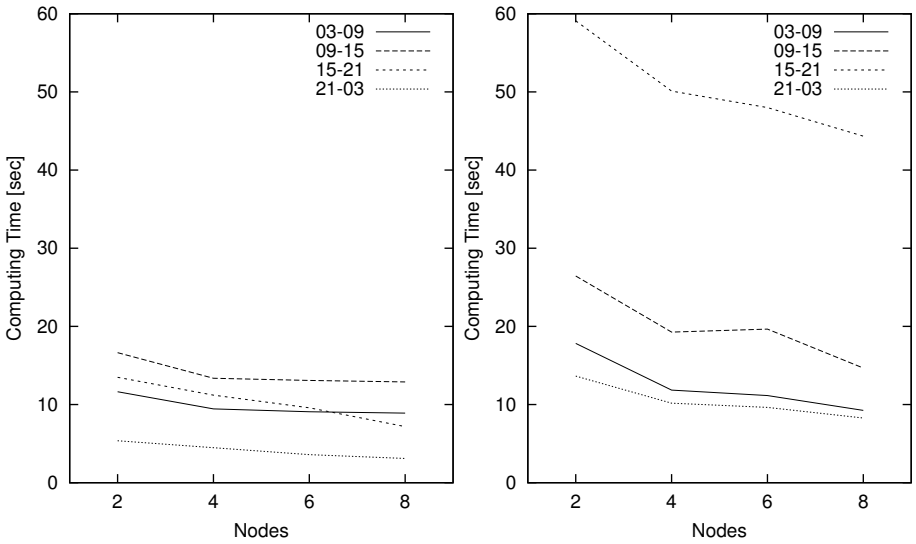


Abbildung 25.7: Laufzeiten für unterschiedlich große Netze: 67 Betriebsstellen (1) und alle 104 Betriebsstellen (2).

des Netzes. In Abbildung 25.7 (2) [S. 165] habe ich das ganze Netz genommen und es wie oben in 26 Teile geteilt, während ich in Abbildung 25.7 (1) [S. 165] nur 15 dieser 26 Teile simuliert habe. Unter diesen 15 waren auch einige sehr komplexe Teile, so dass in Abbildung 25.7 (1) [S. 165] genau 67 Betriebsstellen gegenüber 104 in Abbildung 25.7 (2) [S. 165] simuliert wurden. Auch hier kann man wieder sehen, das DRS nicht optimal skaliert: obwohl (2) nur etwa 50% größer ist als (1), dauert die Berechnung mehr als doppelt so lange: vergleichbare Kurven in (2) liegen entsprechend höher als die in (1).

25.3 Anzahl Rechenknoten

Im vorigen Abschnitt haben wir die Skalierbarkeit von DRS anhand der Variation der Problemgröße betrachtet. Hier will ich zeigen, wie sich für ein gegebenes Problem die Verwendung von DRS und der Einsatz unterschiedlich vieler Rechenknoten auswirkt. Davon kann ich dann später Aussagen zum *Speedup* (Abschnitt 2.1, S. 7) ableiten.

Sehen wir uns zunächst das Laufzeitverhalten für kleinere Probleme an, die 6-Stunden-Zeitscheiben: In Abbildung 25.8 [S. 166] sehen wir, wie sich für diese die Verwendung mehrerer Knoten auszahlt, hier für die unterschiedlichen Zeitscheiben. Ich habe hier wieder die Partitionierung mit 26 Teilen verwendet.

Abbildung 25.9 [S. 166] zeigt demgegenüber die Verwendung unterschiedlicher Partitionierungen (mit 16, 26 und 37 Teilen), hier wurden jeweils die Züge von 03-09 berechnet. Hier kann man übrigens schon sehen, dass die Anzahl der Teil einen relativ großen Einfluss auf die Laufzeit hat, größer jedenfalls als die Zahl der Rechenknoten. Die Partitionierung mit den kürzesten Laufzeiten – damit die *beste* Partitionierung – hat 26 Teile. Deshalb habe ich diese auch meist für die anderen Vergleiche (zum Beispiel im vorigen Abschnitt) benutzt.

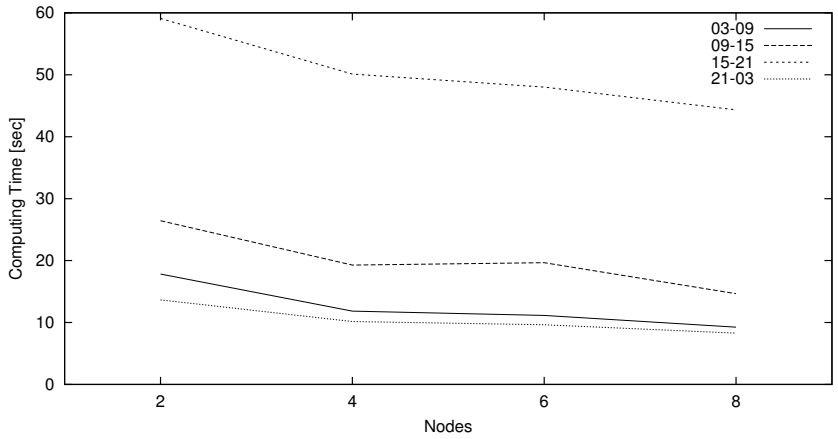


Abbildung 25.8: Laufzeiten abhängig von der Anzahl der Rechner und für unterschiedliche 6-Stunden-Ausschnitte.

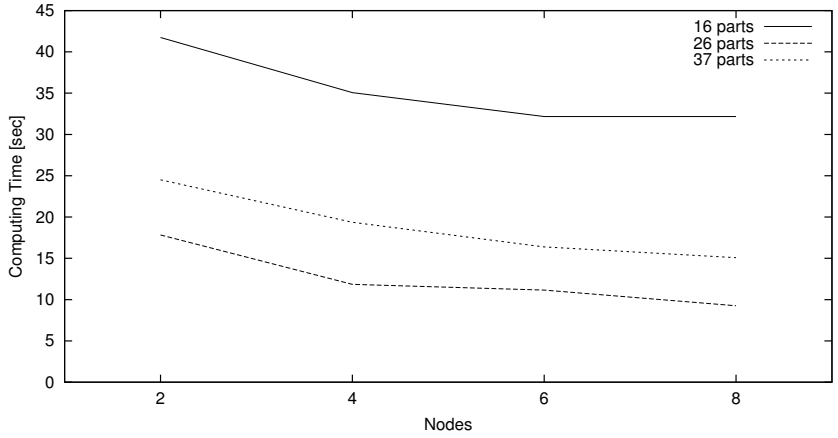


Abbildung 25.9: Laufzeiten abhängig von der Anzahl der Rechner und für unterschiedliche Partitionierungen.

Beide Vergleiche zeigen, dass sich die Anzahl der Knoten zwar auf die Laufzeit auswirkt, was zu erwarten war, aber weniger stark als vielleicht erhofft.

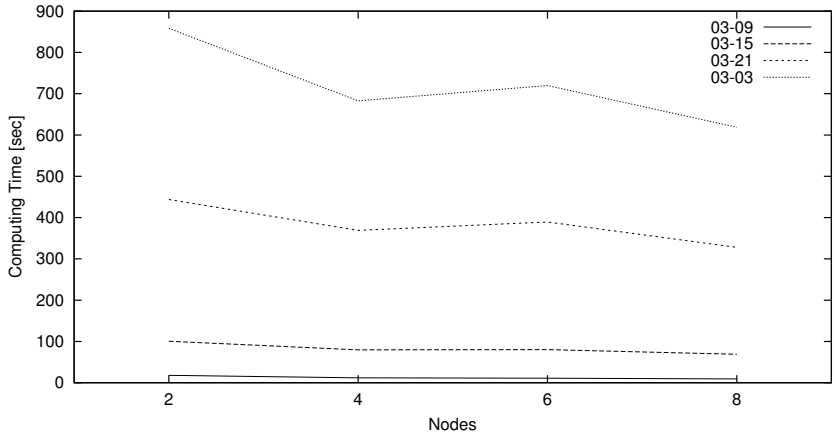


Abbildung 25.10: Laufzeiten abhängig von der Anzahl der Rechner und für unterschiedliche Tages-Ausschnitte.

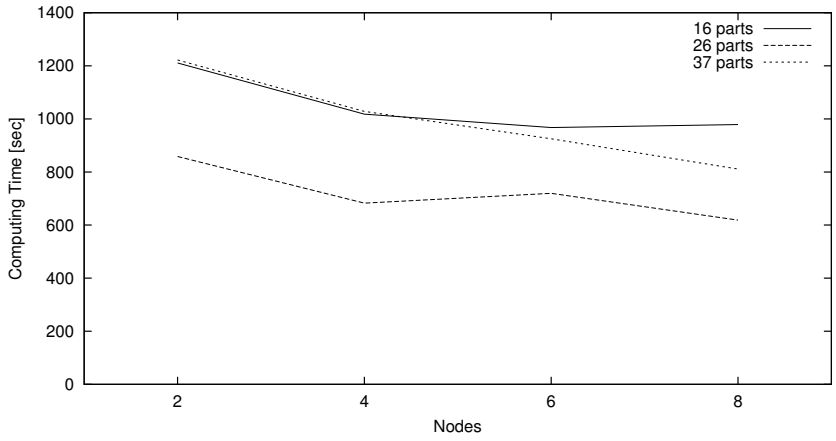


Abbildung 25.11: Laufzeiten abhängig von der Anzahl der Rechner und für unterschiedliche Partitionierungen.

Für größere Beispiele, i.e. solche mit mehr Zügen, ergibt sich ein ähnliches Bild: siehe Abbildung 25.10 [S. 167] (wieder für 26 Teile) und Abbildung 25.11 [S. 167] (für den ganzen Tag, 03-03).

Auffallend ist hier, dass die Laufzeiten bei 26 Teilen mit 4 Knoten oft besser sind als mit 6 Knoten. Das kommt zum einen daher, dass die Partitionierung gerade bei so vielen Teilen bzgl. der Komplexitätsverteilung nicht optimal ist, ich hatte das ja in Kapitel 21 [S. 141] schon angedeutet. Außerdem verteilen wir die Teile trivial auf die Knoten, per *Round-Robin*-Verfahren. Tabelle 25.3 [S. 168] zeigt die konkrete Verteilung. Und damit kann es sein, dass bei den sechs

Knoten die Teile 5 Teile 1,7,13,19,25 des ersten Knotens zusammen komplexer sind, als bei 4 Knoten etwa die 6 Teile 3,7,11,15,19,23 des dritten Knotens. Und dadurch kann die Gesamtlaufzeit, die ja unmittelbar vom Maximum der Laufzeiten auf den einzelnen Knoten abhängt, mit 6 Knoten schlechter sein als mit 4.

Knoten	4 Knoten	6 Knoten
1	1,5,9,13,17,21,25	1,7,13,19,25
2	2,6,10,14,18,22,26	2,8,14,20,26
3	3,7,11,15,19,23	3,9,15,21
4	4,8,12,16,20,24	4,10,16,22
5		5,11,17,23
6		6,12,18,24

Tabelle 25.3: Round-Robin-Verteilung von 26 Teilen auf 4 bzw. 6 Knoten.

25.4 Partitionierungen

Wie gesagt, haben wir unterschiedliche Partitionierungen berechnet: mit 1, 2, 4, 16, 26 und 37 Teilen. Die Anzahl der Teile wirkt sich *dramatisch* auf die Gesamt-Rechenzeit aus: In Abbildung 25.12 [S. 168] sehen wir die Ergebnisse für 1, 2 und 4 Teile, jeweils auf 4 Rechenknoten. Und hier kann man sehen, dass Zerlegung alleine schon Vorteile bringt, schließlich ist der Geschwindigkeits-Zuwachs super-linear: durch Verdoppelung der Teile (und damit der tatsächlich verwendeten Rechner) wird die Rechenzeit in den meisten Fällen mehr als halbiert!

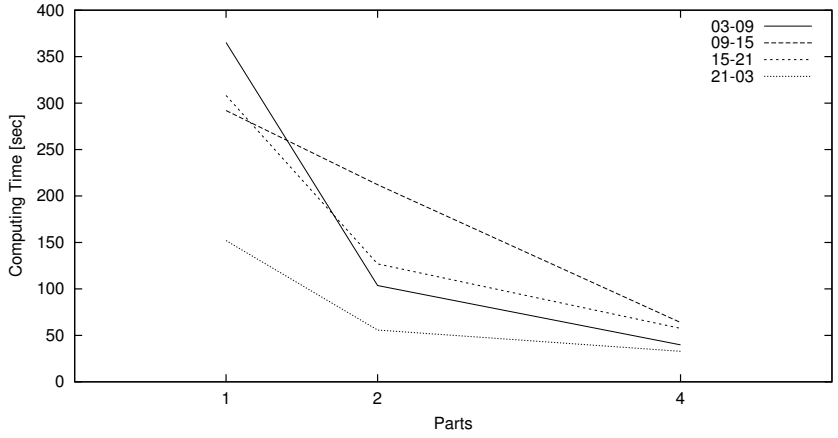


Abbildung 25.12: Vergleich der Laufzeiten unterschiedlicher Zerlegungen auf 4 Rechnern.

Noch drastischer sieht der Vergleich *aller* Zerlegungen aus: siehe Abbildung 25.13 [S. 169]. Hier fällt wieder auf, dass sich eine Linie deutlich nicht-monoton verhält. Ähnlich wie oben wird hier die Gesamtlaufzeit wesentlich von der Lauf-

zeit eines Knotens bestimmt. Und im Fall 15-21 gibt es wieder einen, der aus der Reihe fällt, weil er deutlich mehr zu berechnen hat als die anderen Knoten.

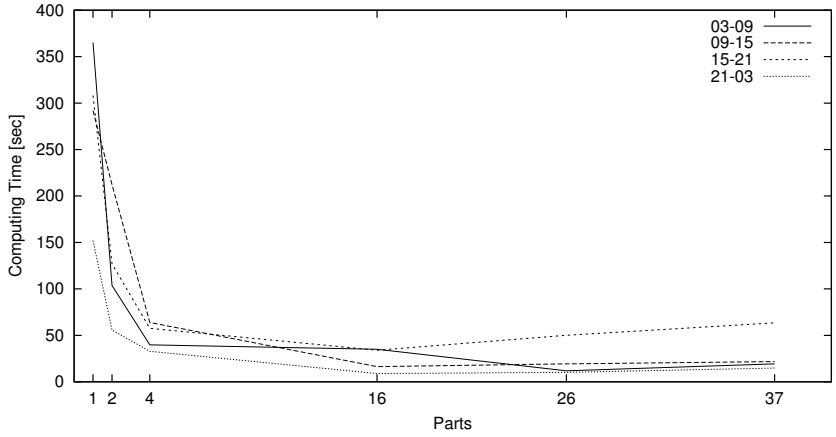


Abbildung 25.13: Vergleich der Laufzeiten unterschiedlicher Zerlegungen auf 4 Rechnern.

Man beachte, dass die *Zerlegung in einen Teil* natürlich keine Zerlegung bedeutet und hier also das Problem unzerlegt berechnet wird, also komplett ohne den verteilten Algorithmus. Dadurch sind natürlich keine Iterationsschritte zum Abgleich nötig. Alleine die Reduktion der lokalen Komplexität aber bringt eine erhebliche Reduktion der Rechenzeiten!

25.5 Nicht- / Determinismus

DRS kann synchron oder asynchron ausgeführt werden (siehe Kapitel 20, S. 130). Der Vorteil der synchronen Ausführung ist die Determiniertheit, damit läuft jede Simulation auf denselben Eingabedaten immer gleich ab. Allerdings ist dieser Modus langsamer, laut Abbildung 25.14 (1) [S. 170] im Schnitt etwa 25%. Offenbar kann beim asynchronen Modus mehr parallele Kapazität ausgenutzt werden, schließlich müssen die einzelnen Teile nicht aufeinander warten, sondern können immer gleich nach Beendigung einer Berechnung mit den von den Nachbarn erhaltenen Änderungen neu beginnen.

Tatsächlich werden asynchron sogar etwas weniger Iterationen berechnet, wie Abbildung 25.14 (2) [S. 170] zeigt: hier sehen wir die durchschnittliche Anzahl von lokalen Simulationsberechnungen. Das hat seinen Grund darin, dass im synchronen Modus immer alle auf die Beendigung aller anderen warten müssen, und damit immer auch auf den komplexesten Teil. Im asynchronen Modus aber können kleinere Teile schon miteinander konsistent gemacht werden, während die komplexen Teile das erste Mal berechnet werden. Die konsistenten Teilergebnisse fließen dann gemeinsam in die komplexeren Teile ein, wodurch letztere deutlich seltener berechnet werden müssen.

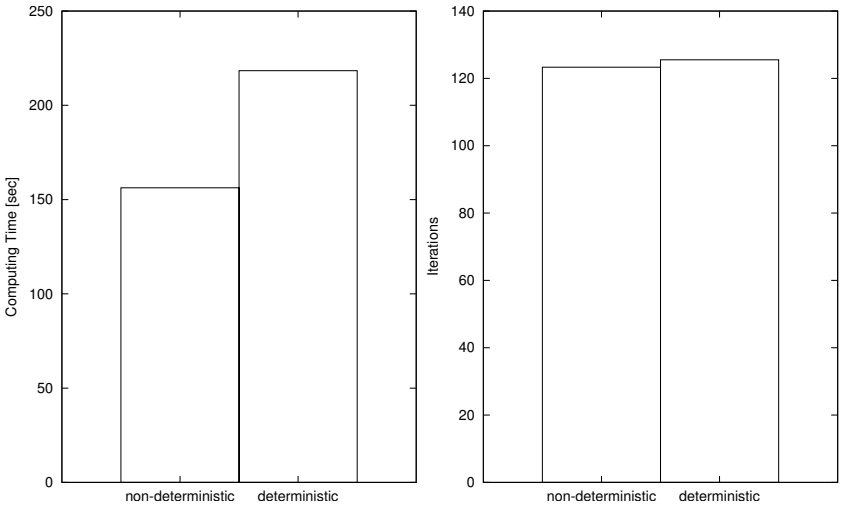


Abbildung 25.14: Durchschnittliche Laufzeiten und Iterationen im asynchronen, nicht-deterministischen, und im synchronen, deterministischen, Modus.

25.6 Verteilte Kontrolle

Für sehr große Simulationen kann die zentrale Kontrolle ein Engpass werden: hier kommunizieren ja alle lokalen Simulationen immer und ausschließlich mit der zentralen CAP. Um mit diesem Problem umgehen zu können – schließlich ist in dem Forschungsprojekt SIMONE ja angestrebt, das ganze deutsche Eisenbahnnetz zu simulieren – haben wir eine dezentrale Kontrolle implementiert, siehe Abschnitt 20.3 [S. 137]. Die folgenden Laufzeituntersuchungen sind in [Ruh04] ausführlicher erläutert.

25.6.1 Kommunikation

Diese dezentrale Kontrolle verringert die Kommunikation mit der Zentrale drastisch: Abbildung 25.15 [S. 171] zeigt die kumulierten Größen aller Nachrichten mit der zentralen CAP in Abhängigkeit der Anzahl der Rechenknoten, im Durchschnitt aller unserer Berechnungen.

Dieser Unterschied ist tatsächlich je größer, je größer die Probleme sind: in Abbildung 25.16 [S. 171] sehen wir wieder die kumulierte Größe aller Nachrichten mit der Zentrale, hier in Abhängigkeit der Anzahl der Züge.

Die dezentrale Kontrolle reduziert nicht nur die Kommunikation mit der Zentrale sondern auch die gesamte Kommunikation, wie man an Abbildung 25.17 [S. 172] sehen kann. Bei der dezentralen Kontrolle können nämlich Simulationen, die in demselben Worker laufen, direkt miteinander kommunizieren, und müssen nicht langsame und teure Netzwerkverbindungen nutzen. Das ist beim zentralen Ansatz nicht möglich.

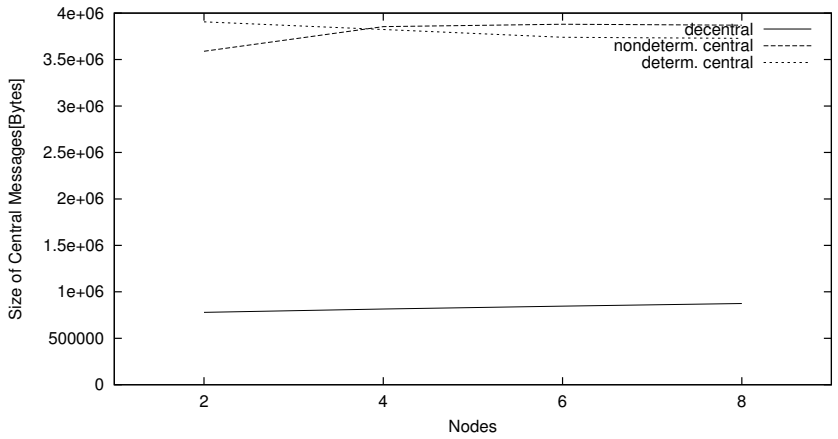


Abbildung 25.15: Summe der Größen aller Nachrichten mit der zentralen CAP bei den unterschiedlichen Verfahren: zentral deterministisch, zentral nicht-deterministisch, dezentral (nicht-deterministisch).

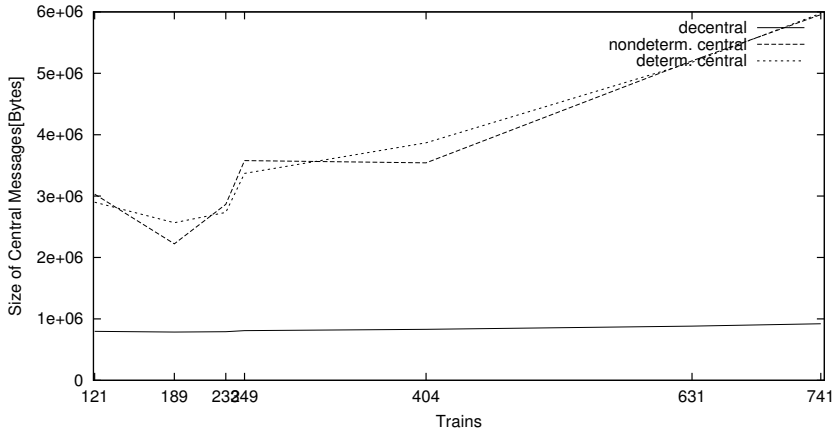


Abbildung 25.16: Größe aller Nachrichten mit der zentralen CAP in Abhängigkeit der Problemgröße.

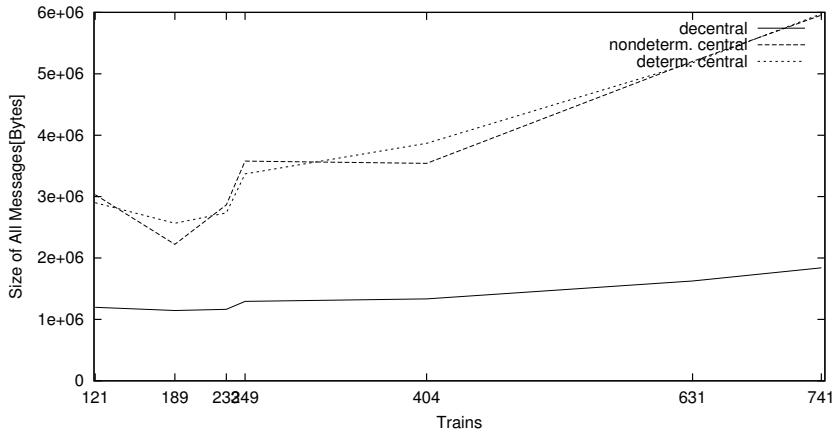


Abbildung 25.17: Größe *aller* Nachrichten (nicht nur mit der Zentrale) in Abhängigkeit der Problemgröße.

25.6.2 Laufzeiten

Die dezentrale oder verteilte Kontrolle schafft also die deutliche Reduktion von Netzwerk-Kommunikation. Das wirkt sich für unser Fallbeispiel allerdings nicht in einer besseren Laufzeit aus, dafür ist es offenbar zu klein. Abbildung 25.18 [S. 172] zeigt die Laufzeiten in Abhängigkeit der Problemgröße.

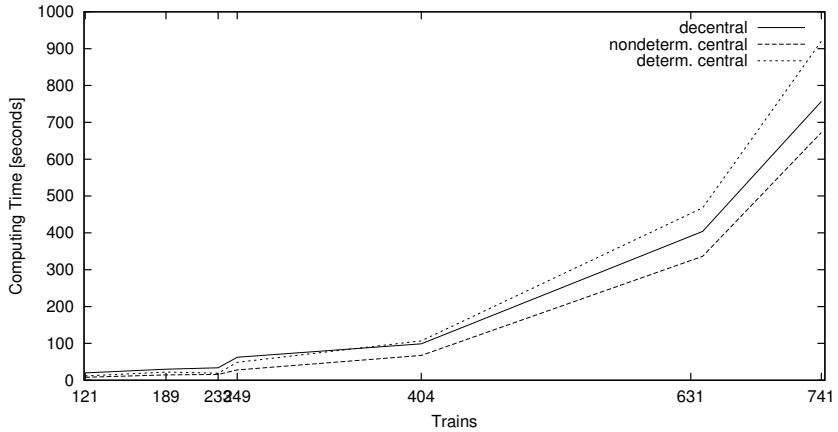


Abbildung 25.18: Laufzeiten in Abhängigkeit der Problemgröße.

25.7 Simulatoren

In obigen Untersuchungen haben wir immer den in DRS integrierten Prototypen verwendet. Wie verhält sich nun der SIMONE-Simulator, in CHIP-Prolog [CHI01] realisiert?

Hier hatten wir weniger Rechner zur Verfügung, weil für die Ausführung dieses Simulators eine relativ teure Lizenz erforderlich ist. Und deshalb konnten wir nur vier einigermaßen vergleichbare *Knoten* verwenden: ein Rechner mit zwei Prozessoren und zwei weitere mit jeweils einem Prozessor. Die Prozessoren sind alle vom Typ Pentium 4 mit 3GHz. Außerdem haben wir jeweils Windows XP benutzt und das CHIP System in der Version 5.4.1.

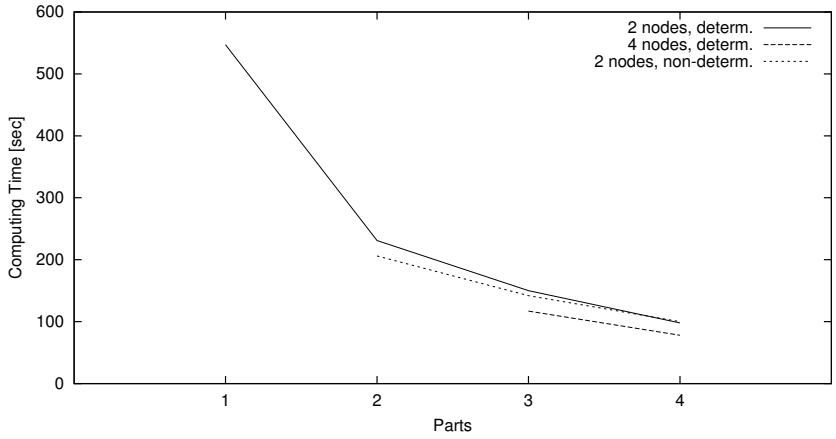


Abbildung 25.19: Laufzeiten mit dem CHIP-basierten SIMONE-Simulator für die Zeitscheibe 15-16 Uhr (59 Züge).

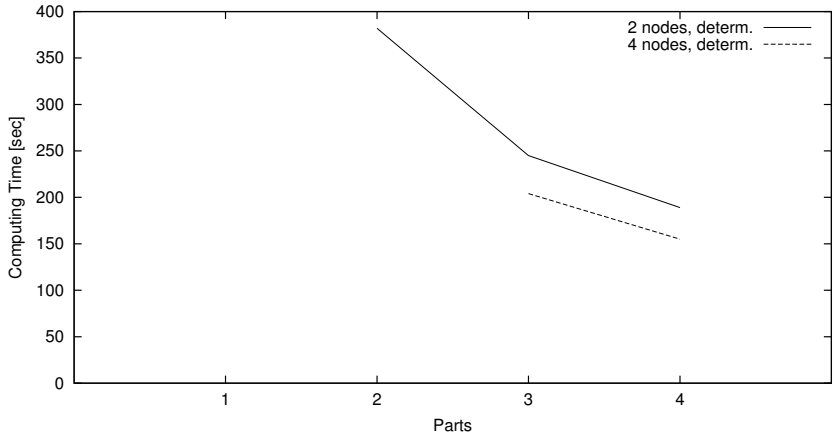


Abbildung 25.20: Laufzeiten mit dem CHIP-basierten SIMONE-Simulator für die Zeitscheibe 03-07 Uhr (114 Züge).

Abbildung 25.19 [S. 173] und Abbildung 25.20 [S. 173] zeigen die Laufzeitergebnisse, jeweils in Abhängigkeit der Anzahl der Rechenknoten (hier: Prozessoren). Das Verhältnis der Laufzeiten zueinander entspricht den obigen Beobachtungen.

Allerdings fällt auf, dass die absoluten Zahlen doch deutlich über denen des

Java-Prototypen liegen, insbesondere wenn man die verschiedenen Rechnertypen und die unterschiedlichen Problemgröße in Betracht zieht. Der Laufzeitunterschied ist wohl im wesentlichen darin begründet, dass der SIMONE-Simulator aufwändigere Berechnungen durchzuführen hat, als der Java-Prototyp. Ein anderes Problem sind hier die sehr hohen Lizenzkosten, die in unserem Fall eine breitere Testbasis verhindert haben. Leider hat sich außerdem die Kombination der verteilten Umgebung mit dem CHIP-Simulator als weniger stabil als die integrierte Lösung herausgestellt. So hängte sich das CHIP-Programm während einer Präsentation aus ungeklärter Ursache auf, was mit dem Umstand zu tun haben könnte, dass die Lizenz für den entsprechenden Rechner einige Tage später ungültig wurde.

25.8 Speicherbedarf

DRS ist ja nicht nur wegen der schnelleren Berechnung interessant, sondern auch wegen des *verteilten Speicherbedarfs*. So benötigt der CHIP-Simulator für ein Viertel des Netzes des Fallbeispiels und die Züge von 07 bis 13 Uhr etwa 1,35 GB Speicher! Wenn man bedenkt, dass ein CHIP-Programm unter Windows maximal ca. 1,8 GB virtuellen Speicher nutzen kann, sieht man schon, dass man hier sehr schnell an natürliche Grenzen stößt. Genauso mussten wir den Vergleich in Abbildung 25.19 [S. 173] auf die Züge einer einstündigen Zeitscheibe begrenzen, weil wir diesen Test sonst nicht auf einem Rechner hätten durchführen können.

Ähnliche – nämlich endliche – Grenzen gelten für die Version mit dem Java-Prototypen, auch wenn dort die Situation durch ein dynamisches Speichermanagement weniger dramatisch ist. In jedem Fall bietet DRS eine Möglichkeit, Simulationsprobleme zu rechnen, die allein wegen des begrenzt verfügbaren Speichers nicht auf einem Rechner möglich wären.

25.9 Zusammenfassung und Diskussion

Für das Fallbeispiel haben wir verschiedene Laufzeituntersuchungen angestellt. Dabei haben wir folgende Parameter variiert:

Problemgröße Ganzes Fallbeispiel bzw. zwei Drittel davon, Züge unterschiedlicher Zeitausschnitte eines Tages.

Anzahl an Rechenknoten 1,2,4,6,8.

Partitionierungen Das Fallbeispiel zerlegt in 1, 2, 4, 16, 26 und 37 Teile.

Kontroll-Modus Zentral synchron, zentral asynchron, dezentral (asynchron).

Simulatoren Intern, Java-basiert, und extern, CHIP-basiert.

Die Ergebnisse lassen sich bezüglich Performanzgewinn (*Speedup*) und Skalierbarkeit (*Scaleup*) zusammenfassen.

25.9.1 Speedup

Hinter *Speedup* steht die Frage: Kann ich per DRS mein gegebenes Simulationsproblem schneller als ohne DRS berechnen?

Diese Frage kann ich aufgrund der Fallstudie eindeutig mit Ja beantworten.

Wir haben gesehen, dass allein die Zerlegung des gesamten Simulationsproblems in kleine Teilprobleme einen deutlichen Geschwindigkeitszuwachs bringt und das gesamte Problem bei einer gegebenen Zahl an Rechnern (z.B. 4) schneller berechnet werden kann, wenn die Teilprobleme kleiner werden.

Der Speedup durch die Zerlegung ist mehr als linear. Durch die Hinzunahme weiterer Rechenknoten kann man aber keinen weiteren *linearen* Speedup erwarten.

25.9.2 Scaleup

Scaleup bedeutet: Kann ich erwarten, dass ich durch Verwendung von DRS und evtl. die Hinzunahme weiterer Rechenknoten entsprechend größere Probleme besser simulieren kann?

Auch diese Frage kann man deutlich mit Ja beantworten.

Das liegt im wesentlichen daran, dass bei Verwendung von DRS gegenüber einer lokalen Simulation viele Rechner benutzt werden können und damit die gesamte verteilte Speicherkapazität. Das löst das Problem, dass Simulationsprobleme leicht zu groß werden können, um überhaupt in einem Rechner berechnet werden zu können. Das gilt natürlich zunächst nur für den im Projekt SIMONE verfolgten Constraint-basierten Ansatz. Bei klassischen Ansätzen kann man sich vorstellen, das Modell des zu simulierenden Eisenbahnnetzes nicht auf einmal im Speicher zu halten, sondern dynamisch aufzubauen. Dazu existieren aber bislang keine veröffentlichten Ansätze. Es ist auch zweifelhaft, ob man damit schneller Ergebnisse bekäme.

Man kann also per DRS Probleme simulieren, die für eine lokale Simulation zu groß wären. Bezüglich der zu erwartenden Rechenzeit kann man allerdings keinen konstanten Scaleup erwarten, also etwa ein doppelt so großes Problem durch Verwendung doppelter Rechenkapazität in derselben Zeit berechnen zu können. Das liegt aber im wesentlichen daran, dass die Problem-*Komplexität* nicht linear von der reinen Problem-*Größe* (z.B. Anzahl der Züge oder der Gleisabschnitte) abhängt, schließlich ist der Kern des Problems ein Job-Shop-Scheduling Problem und das ist NP-vollständig.

25.9.3 Weitere Beobachtungen

Zerlegung und Verteilung

Bisher haben wir das Fallbeispiel statisch zerlegt und trivial auf die Rechenknoten verteilt.

Wir haben gesehen, dass für das Fallbeispiel und die meisten Rechen-Konfigurationen die Zerlegung in 26 Teile die schnellsten Ergebnisse geliefert hat. Das haben wir aber durch Versuche ermittelt. Es hat sich auch gezeigt, dass die triviale *Round-Robin*-Verteilung schlecht sein kann: So waren manche Berechnungen auf 6 Knoten langsamer als auf 4 Knoten.

Wir brauchen also ein Verfahren, das beliebige Simulationsprobleme für die konkret gegebene Rechen-Konfigurationen optimal zerlegt und optimal verteilt.

Sehr stabile Implementierung

Jeder der Tests wurde 5 Mal durchgeführt, außerdem fast alle möglichen Kombinationen. Insgesamt beruhen die Auswertungen damit auf 1490 einzelnen Tests! Dabei hat sich gezeigt, dass die Implementierung von DRS sehr stabil ist: 900

Simulationen hintereinander wurden auf ein und derselben WRK-Instanz gemacht, ohne dass eine davon neu gestartet werden musste (mehr als 900 konnten wir nicht hintereinander in einer Nacht durchführen und mussten die Cluster-Rechenknoten tagsüber für andere Aufgaben abgeben). Die durchschnittlichen Zeiten in 5 aufeinander folgenden Runden mit eben je ca. 180 Tests waren in Sekunden: 26,26, 26,34, 25,72, 25,85, 26,14. Daraus kann man ablesen dass sich das Laufzeitverhalten der Worker während aller Tests nicht verändert hat, also unter anderem keine Speicher-*Lecks* aufgetreten sind.

Externer Simulator

Die Verwendung des externen Simulators funktioniert, die Schnittstelle hat sich auch als performant erwiesen. Allerdings sollte man wenn möglich auf eine solche Kombination verzichten. Der interne Simulator läuft deutlich stabiler, das integrierte System ist leichter zu entwickeln, zu installieren und zu pflegen.

Job-Shop-Scheduling

Wie weiter oben angedeutet, enthält das Simulationsproblem ein Job-Shop-Scheduling-Problem, nämlich bezüglich der überlappungsfreien Belegung der Gleise mit Zugfahrten. Das gesamte Netz besteht aus 7111 Gleisabschnitten, mithin 7111 Maschinen. Jeder Zug fährt im Durchschnitt über 266 Gleisabschnitte und bildet damit einen Job aus 266 Tasks. Damit müssen etwa im Fall der Zeitscheibe 03-09 189*266=50274 Tasks auf 7111 Maschinen verteilt werden, was im optimalen Fall – bei optimaler Verteilung, Verwendung des Java-Prototypen und unseres Solvers `firstcs` – ganze 9,26 Sekunden dauerte!

Tabelle 25.4 [S. 176] zeigt die entsprechenden Problemgrößen und Rechenzeiten für alle Zeitscheiben.

Zeitscheibe	Züge bzw. *Jobs*	*Tasks*	min. Rechenzeit [Sek]
03-09	189	50274	9,26
09-15	233	61978	12,46
15-21	249	66234	29,50
21-03	121	32186	7,00
03-15	404	107464	68,82
03-21	641	170506	327,81
03-03	741	197106	618,87

Tabelle 25.4: Job-Job-Scheduling als Teilproblem der Simulation: entsprechende Problemgrößen und Laufzeiten.

26 Zusammenfassung

DRS ist ein ganz neuartiges Verfahren zur Eisenbahn-Simulation, theoretisch fundiert, solide umgesetzt, und erfüllt praktisch die gesetzten Ziele.

Ziel dieser Arbeit war es, einen Algorithmus zur verteilten, Constraint-basierten Eisenbahn-Simulation zu entwickeln. Die Arbeit fand im Rahmen des Forschungsprojekts SIMONE (*Simulation von Trassen-Slots und Zuglagen in Eisenbahnnetzen*) statt. Die Motivation für dieses Forschungsprojekt ist es, erstmals Constraint-Programmierung (*CP*) zur Eisenbahn-Simulation einzusetzen, um damit schneller zu besseren Simulationsergebnissen zu kommen. CP kann hier schon in einem monolithischen Simulator die Bearbeitungsgeschwindigkeit erhöhen. Es sollten aber nicht nur existierende Simulationen schneller berechnet, sondern auch deutlich größere Netze als bisher behandelt werden können. Auch hier wollen wir in der Eisenbahn-Simulation praktisch Neuland betreten und verwenden Verteilte Problemlösung. Und dafür habe ich einen Algorithmus entwickelt: DRS (*Distributed Railway Simulation*).

DRS zerlegt ein gegebenes Simulationsproblem in Teile, lässt diese von lokalen, Constraint-basierten Simulatoren lösen, und macht die lokalen Ergebnisse durch wiederholte Iterationen global konsistent. Jede lokale Simulation wird auf ihrem Simulationsteil durchgeführt, bei der ersten Berechnung werden nur die lokalen Daten (v.a. Fahrplaneinträge, Zugdaten) benutzt. Jede lokale Berechnung wird immer vollständig durchgeführt, am Schluss jeder Berechnung werden die Daten der Züge, die durch mehrere Teile fahren, an die anderen Simulationsteile kommuniziert. Bei folgenden Iterationen müssen die lokalen Simulationen die Daten der Nachbarabschnitte (v.a. Ein- und Ausfahrzeiten der grenzüberschreitenden Züge) berücksichtigen. Wenn eine Iteration bezüglich bisherigen Simulationen Änderungen an Ein- oder Ausfahrzeiten vornimmt, muss sie diese wieder an die Nachbarn kommunizieren, und die Nachbarn müssen diese neuen Änderungen ihrerseits in einer weiteren Iteration in ihre Simulation einbauen.

Es konnte gezeigt werden, dass dieser globale Iterationsprozess immer nach endlich vielen Schritten terminiert. Dazu muss der lokale Simulator bei gegebenen Eingabedaten immer dasselbe Resultat erzeugen, also determiniert sein. Außerdem ist eine globale Reihenfolge der Züge notwendig, die lokal in jeder Simulation eingehalten werden muss. Es passiert nämlich häufig, dass simulierte Züge einander verdrängen. Und Verdrängungen müssen in allen Teilen konsistent passieren, damit sich nicht zwei Züge in einem Konflikt gegenseitig immer wieder endlos verdrängen können. Wenn am Ende des Iterationsprozesses eine konsistente Lösung für das gesamte Simulationsproblem vorliegt, ist diese auch korrekt: Sie beschreibt eine mögliche Fahrt aller Züge durch das Gleisnetz, wobei alle harten Bedingungen, wie Fahrplaneinträge oder die Gleisausschlussbedingung, eingehalten sind.

Das Verfahren ist grundsätzlich nicht-deterministisch: Unterschiedliche Kommunikations- oder Ausführungsreihenfolgen können am Ende zu unterschiedlichen Ergebnissen führen. Es kann aber sehr leicht deterministisch gemacht werden, indem die globalen Iterationen synchronisiert werden. Ein Grundprinzip von DRS ist, Züge (global) immer nur in die Zukunft zu verschieben, bis sie konsistent sind. Gerade für die mögliche Anwendung der Fahrplankonstruktion müsste hier optimiert werden. Global aber kann man nicht generell inkonsistente Zugfahrten in die Vergangenheit verschieben, weil hier in der Regel

(v.a. durch die Fahrplaneinträge, die erfordern, dass kein Zug früher als vorgesehen den Bahnhof verlässt) keine Varianz besteht und man damit generell sehr schnell widersprüchliche Constraints bekommt. Eher bietet sich lokale Optimierung an, da muss man aber darauf achten, dass die globale Zugreihenfolge nicht verletzt wird, was etwa bei einer lokalen Suche einer optimalen Reihenfolge der Fahrten passieren kann. Ohne diese Reihenfolge hätte man wohl keine automatische Terminierung mehr, müsste Terminierung dann ad hoc erzwingen. Weil für den globalen Algorithmus zwischen den lokalen Simulationen Bedingungen ausgetauscht werden, die dann lokal eingehalten werden müssen, sollte die lokale Simulation mit Constraint-Propagation arbeiten.

Das theoretisch nachgewiesene Konzept wurde erfolgreich umgesetzt. Dazu haben wir die Haupt-Komponenten DIR, WRK, CAP und SIM entworfen, die zusammen mit den externen Komponenten Datenbank, Tomcat Web-Server, und CVS das DRS-System bilden. Der DIR realisiert den zentralen Informationsdienst, die WRK sind die Arbeitspferde, die die in der Komponente SIM realisierten Simulationen durchführen, alles das wird gesteuert vom Anwender und der von ihm benutzen Kontroll-Anwendung CAP. In der Datenbank liegen vor allem die Problem-Spezifikationen, die Tomcat Web-Server dienen als Laufzeitumgebung für die WRK, und das CVS hilft im integrierten Entwicklungsprozess die Programm-Quellen konsistent zu halten. Die Kontrolle des Algorithmus findet entweder zentral in der CAP statt oder dezentral auf die WRK verteilt. Dezentral muss ein spezieller verteilter Algorithmus den globalen Endzustand feststellen. Das DRS-System besitzt einen eigenen rudimentären lokalen Simulator, der im SIMONE-Projekt entwickelte Simulator kann über eine generische Schnittstelle als externes Programm eingebunden werden. Ein spezieller integrierter Entwicklungsprozess erlaubt eine verteilte Entwicklung und den Austausch wichtiger DRS-Komponenten während der Laufzeit des Gesamtsystems durch ein Online-Update.

Damit profitiert DRS von den Vorteilen verteilter Systeme wie Performanzgewinn, Fehlertoleranz und Ausfallsicherheit, inkrementelle Erweiterbarkeit, örtliche Bereitstellung von Rechenleistung, Kosteneffizienz. Einige davon stehen freilich nicht automatisch zur Verfügung, sondern werden von der Implementierung effektiv realisiert. Genauso löst die Implementierung viele Probleme wie Informationsdefizit, Inkonsistenzen, Nichtdeterminismus, Terminierung, Kontrolle, Konfiguration, Kommunikation, Programmierung, Heterogenität und Bewertung.

Umfangreiche Laufzeittests belegen, dass Konzept und Realisierung wie gewünscht arbeiten. Wir haben die Problemgrößen variiert, die Anzahl an Rechenknoten, die Partitionierungen, zentrale und dezentrale sowie synchrone und asynchrone Kontrolle verwendet, und den internen mit dem externen Simulator verglichen. Dabei hat sich das System als sehr stabil erwiesen, so konnten einige hundert Simulationen auf einer laufenden WRK-Instanz ausgeführt werden.

Die empirischen Untersuchungen haben gezeigt, dass alleine die Problem-Zerlegung zu einem oft mehr als linearen Speedup führt und durch die Verteilung vor allem wegen des dadurch beliebig vorhandenen Speichers Probleme gelöst werden können, die für einen Rechenknoten zu groß wären.

DRS ist ein ganz neuartiges Verfahren zur Eisenbahn-Simulation, theoretisch fundiert, solide umgesetzt, und erfüllt praktisch die gesetzten Ziele.

27 Ausblick

Dieses Kapitel blickt in die Zukunft und beschäftigt sich mit folgenden Fragen: Welche Erweiterungen des grundlegenden Algorithmus sind viel versprechend? Wo kann die Realisierung noch verbessert werden? Welche zusätzlichen Anwendungen sind denkbar?

27.1 Algorithmus

Hierarchische Simulation

Die verteilte Simulation könnte hierarchisiert werden: Auf den verschiedenen Stufen der Hierarchie würde das globale Netz unterschiedlich simuliert. So könnten auf einer höheren Ebene weniger, v.a. überregionale Züge simuliert werden, auf den unteren Ebenen hingegen entsprechend mehr Züge in kleineren Teilnetzen, einschließlich der regionalen oder lokalen. Die Ebenen hätten dann entsprechende Prioritäten: Entscheidungen (z.B. Fahrzeiten) der höheren Ebenen müssten auf den niedrigeren Ebenen akzeptiert werden, die regionalen Züge müssten sich den überregionalen anpassen.

Ähnlich könnte man auf den Stufen der Hierarchie unterschiedliche Simulations-Granularität benutzen: Auf der höchsten Ebene etwa gäbe es nur Bahnhöfe, die durch ein- oder mehrgleisige Strecken verbunden sind, und auf der untersten Ebene die von der Bahn geforderte mikroskopische Simulation mit Gleisabschnitten, Weichen, etc. Damit könnten Dispositions-Entscheidungen wie z.B., wann ein Zug in eine Betriebsstelle einfahren darf, schon auf abstrakter Ebene gelöst werden.

Optimierungsmöglichkeiten

In Abschnitt 16.5 [S. 94] hatte ich angedeutet, dass es insbesondere für die mögliche Anwendung der Fahrplankonstruktion nicht genügt, Züge bei Konflikten bzw. Inkonsistenzen immer nur in die Zukunft zu verschieben. Es sollte auch Möglichkeiten geben, Zugfahrten zu optimieren. Dafür könnte man die Iterationsergebnisse Stück für Stück festschreiben, das System quasi *langsam abkühlen*: man könnte lokal entscheiden, welche Zugfahrten schon festliegen, und die *größeren* (mit einer größeren Prioritätsnummer; die kleinste Nummer 0 steht für die höchste Priorität) dann nochmal früher versuchen. Dafür könnte man eine *Zugnummerngrenze* z einführen, diese während der Simulation immer weiter erhöhen und lokal immer solche Züge, deren Nummer unterhalb der Grenze z liegen, nicht mehr verschieben (natürlich müssen diese Zugfahrten bereits konsistent sein), die Züge jenseits von z aber auch in die Vergangenheit verschieben. Dies würde natürlich den Rechenaufwand erhöhen, aber möglicherweise erhielte man wenigstens für einige Züge ein (bzgl. Verspätung) besseres Ergebnis.

Adaptive Zerlegung

Die empirischen Untersuchungen haben gezeigt, dass es nicht trivial ist, für ein gegebenes Simulationsproblem und eine konkret gegebene Rechen-Infrastruktur die Zerlegungs- und Verteilungs-Konfiguration zu finden. Ein adaptives Verfahren, das eben auf die konkreten Gegebenheiten eingeht, sollte hier Abhilfe schaffen (siehe auch Kapitel 21, S. 141 und Abschnitt 25.9.3, S. 175). Dazu werden bald in [Muk04] Ergebnisse vorliegen.

Unmittelbarer Austausch

Ich habe beim Entwurf von DRS darauf geachtet, die Kommunikation zwischen den lokalen Simulationen zu reduzieren. Neuere Entwicklungen der Hardwaretechnik könnten es möglich machen, deutlich schneller und intensiver zwischen den Knoten zu kommunizieren. Damit könnte man von dem in DRS beschriebenen *Batch*-Verfahren – Interaktion nicht während, sondern nur am Ende jeder Berechnung – etwas abrücken und auch Daten (Ein- und Ausfahrzeiten) zwischen den Simulationsteilen kommunizieren, *während* diese berechnet werden. Die Simulationen müssten beim vorausschauenden, Constraint-basierten Ansatz, anders als in [Kla94] (siehe Abschnitt 5.3, S. 43), nicht komplett Zeitsynchronisiert werden.

27.2 Realisierung

Fehlertoleranz

Das Gesamtsystem ist schon jetzt einigermaßen fehlertolerant (s.a. Kapitel 24, S. 155). So läuft eine Simulation problemlos weiter, wenn ein Worker ausfällt, der an dieser Simulation gerade nicht beteiligt ist. Eine laufende Simulation sollte aber auch den Ausfall eines beteiligten Workers verkraften. Dazu sollten bei der zentralen Kontrolle die bekannten kommunizierten Daten der grenzüberschreitenden Züge gespeichert werden, und beim Ausfall eines Workers diese Daten an einen anderen übergeben werden, damit dieser den Teil konsistent zu den Daten der Nachbarn neu simulieren kann. Erste Ansätze dazu sind bereits implementiert.

Datensicherheit

Wir benötigen auch ein Sicherheitskonzept für DRS. Im Moment verlassen wir uns wie gesagt (Abschnitt 2.3.5, S. 17 und Kapitel 24, S. 155) auf die grundlegenden Mechanismen, die in Java und Tomcat bereits eingebaut sind und standardmäßig verwendet werden. Beide bieten jedoch weiter Möglichkeiten, die wir für DRS noch nutzen müssen.

Entwicklungsprozess

Beim integrierten Entwicklungsprozess fehlt ein konsistentes Test-Verfahren. Im Moment wird bei einer neuen Version, die per Online-Update in das laufende System eingespielt wird, nur überprüft, ob diese kompilierbar ist. Die Komponenten sollen eigene Test-Methoden bekommen, die vor dem Update immer überprüfen, ob der neue Code auch korrekt ist. Automatische Tests sind eine sehr elegante Methode der Softwaretechnik, um korrekte Programme zu entwickeln. Siehe hierzu zum Beispiel [KP00].

27.3 Anwendungen

Fahrplan-Konstruktion

Wie in Abschnitt 16.4 [S. 93] schon angedeutet, ist Fahrplan-*Konstruktion* eine zu unserer *Simulation* sehr ähnliche Anwendung. Tatsächlich kann auch sie sehr gut mit dem Constraint-Ansatz realisiert werden. Dabei wird anders als bei der Simulation nicht ein Fahrplan gegeben, der möglichst einzuhalten ist, die Züge werden vielmehr relativ frei losgeschickt, um sich ihre Trasse selbst zu suchen. In DRS könnte das einfach dadurch realisiert werden, dass der *lokale Simulator* Fahrplankonstruktion durchführt. Global – am verteilten Verfahren –

müsste sich nicht notwendigerweise etwas ändern. Allerdings wäre es für Fahrplankonstruktion wünschenswert, die Ergebnisse optimieren zu können (siehe oben Abschnitt 27.1, S. 179).

Internationale Eisenbahnnetze

DRS löst ein gegebenes Simulationsproblem, indem es dieses zerlegt und verteilt berechnet. Man könnte das Verfahren aber auch für schon verteilte Eisenbahnnetze benutzen, beispielsweise das europäische. Dafür bräuchte man dann für jedes Land ein eigenes System, das die Länder-spezifischen Gegebenheiten berücksichtigt. Die Systeme der Länder würden dann per DRS-Verfahren kooperativ eine Simulation oder einen neuen Fahrplan für das gesamte Netz berechnen.

Dieses Verfahren würde die *sozialen Grenzen* ([Han01]) berücksichtigen: Schnittstelle zwischen den Ländern sind die grenzüberschreitenden Züge, innerhalb jedes Landes kann disponiert werden, ohne dass die entsprechenden Regeln nach außen sichtbar gemacht werden müssen. Dadurch könnten die nationalen Bahnbetreiber ihre Daten und Regeln geheim halten.

Prozess-Planung

In Abschnitt 25.9.3 [S. 175] hatte ich darauf hingewiesen, dass jedes Eisenbahn-Simulationsproblem ein Job-Shop-Scheduling-Problem als Teilproblem enthält. Abbildung 27.1 [S. 181] zeigt beispielhaft einen Software-Entwicklungs-Prozess. Auch hier gibt es Aufgaben, die miteinander zusammenhängen und die unter Berücksichtigung einiger Rahmenbedingungen zeitlich angeordnet werden müssen. Auch dafür wäre DRS ein vielversprechender Ansatz.

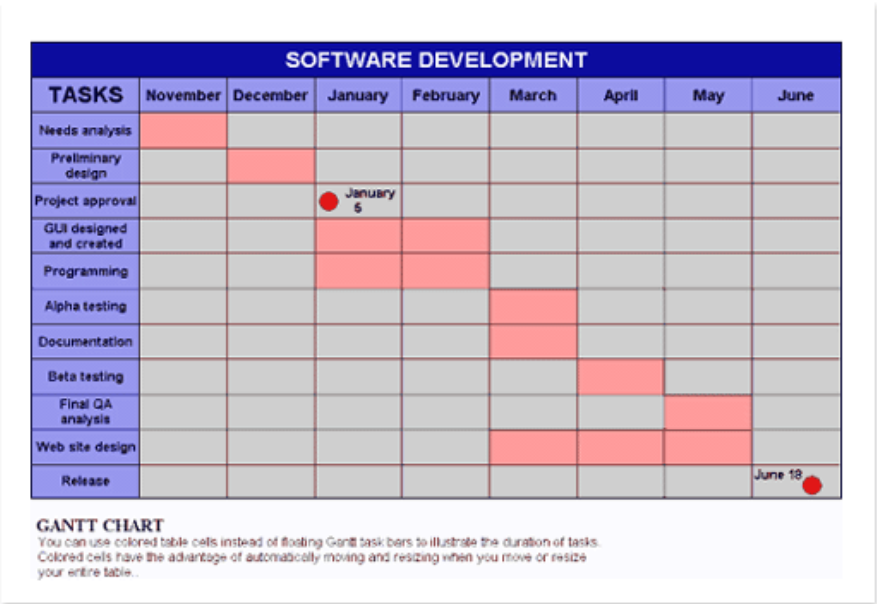


Abbildung 27.1: Zeitplan eines Software-Eintwicklungs-Prozesses [Sma].

Literaturverzeichnis

[AB93] A. Aggoun und N. Beldiceanu. Extending CHIP in Order to Solve Complex Scheduling and Placement Problems. *Mathematical and Computer Modelling*, 1993.

[ABC+99] Eleutherios Athanassiou, Peter Barrett, Delia Chirichescu, Marie Pierre Gleizes, Pierre Glize, Dimitrios Katsoulas, Alain Leger, José Ignacio Moreno und Hans Schlenker. Abrose: A Cooperative Multi-agent Based Framework for Marketplace. In *Proc. IATA'99*. Springer LNCS 1699, 1999.

[Abe90] Dirk Abel. *Petri-Netze für Ingenieure*. Springer, 1990.

[ABG02] Martin Alt, Holger Bischof und Sergei Gorlatch. Algorithm Design and Performance Prediction in a Java-Based Grid System with Skeletons. In *Proc. European Conference on Parallel Processing Euro-Par*, 2002.

[AFM99] Slim Abdennadher, Thom W. Frühwirth und Holger Meuss. Confluence and Semantics of Constraint Simplification Rules. *Constraints*, 4(2), 1999.

[AL97] E. Aarts und J.K. Lenstra, Hrsg. *Local Search in Combinatorial Optimization*. John Wiley and Sons, 1997.

[ALL96] M. Adamy, I. Lalis und M. Liska. Simulation of Large and Complex System. In *Proc. Advanced Simulation of Systems ASS'96*, Zabreh, 1996.

[And01] David Anderson. SETI@home. In Oram [Ora01].

[Apaa] Apache Software Foundation. *Apache Ant*. `http://ant.apache.org/`.

[Apab] Apache Software Foundation. *Apache Tomcat*. `http://jakarta.apache.org/tomcat/`.

[APdLO97] M. Alfonseca, E. Pulido, J. de Lara und R. Orosco. OOCSMP: An Object-Oriented Simulation Language. In *Proc. 9th European Simulation Symposium ESS97*, 1997.

[Apt84] Krzysztof R. Apt. Proving Correctness of CSP Programs – A Tutorial. In Broy [Bro84].

[Apt99] Krzysztof R. Apt. The Essence of Constraint Programming. *Theoretical Computer Science*, 1999.

[Apt03] Krzysztof R. Apt. *Principles of Constraint Programming.* Cambridge University Press, 2003.

[AR01] Slim Abdennadher und Christophe Rigotti. Towards Inductive Constraint Solving. In Walsh [Wal01].

[AS97] Slim Abdennadher und Hans Schlenker. INTERDIP - Ein Interaktiver Constraint-basierter Dienstplaner für Krankenstationen. In *Proc. 12. Workshop Logische Programmierung*, München, 1997.

[AS99a] Slim Abdennadher und Hans Schlenker. INTERDIP - An Interactive Constraint Based Nurse Scheduler. In *Proc. First International Conference and Exhibition on The Practical Application of Constraint Technologies and Logic Programming, PACLP99*, London, 1999.

[AS99b] Slim Abdennadher und Hans Schlenker. Nurse Scheduling using Constraint Logic Programming. In *Proc. Eleventh Annual Conference on Innovative Applications of Artificial Intelligence, IAAI-99.* The MIT Press, 1999.

[Bae91] J. C. M. Baeten, Hrsg. *Applications of Process Algebra.* Cambridge University Press, 1991.

[Bar] Roman Barták. *(Online) Guide to Prolog Programming.* `http://kti.ms.mff.cuni.cz/~bartak/prolog/`.

[Bau96] Bernd Baumgarten. *Petri-Netze.* Spektrum Akademischer Verlag, 1996.

[BB98] Greg J. Badros und Alan Borning. The Cassowary Linear Arithmetic Constraint Solving Algorithm: Interface and Implementation. Technical Report UW-CSE-98-06-04, University of Washington, 1998.

[BBS] N. Beldiceanu, E. Bourreau und H. Simonis. A Note on Perfect Square Placement. Technical report, Cosytec.

[BC93] Frédéric Benhamou und Alain Colmerauer, Hrsg. *Constraint Logic Programming.* The MIT Press, 1993.

[BC94] N. Beldiceanu und E. Contejean. Introducing Global Constraints in CHIP. *Mathematical and Computer Modelling*, 20, 1994.

[BC01] Nicolas Beldiceanu und Mats Carlsson. Sweep as a Generic Pruning Technique Applied to the Non-overlapping Rectangles Constraint. In Walsh [Wal01].

[BC02] Nicolas Beldiceanu und Mats Carlsson. A New Multi-resource *cumulatives* Constraint with Negative Heights. In Van Hentenryck [Van02].

[BD94] Brigitte Bärnreuther und Heinrich Dietsch. Verteilte Rechnersysteme als Voraussetzung Rechnerintegrierter Produktion. In Wedekind [Wed94].

[BDM+99] Peter Brucker, Andreas Drexl, Rolf Möhring, Klaus Neumann und Erwin Pesch. Resource-Constrained Project Scheduling: Notation, Classification, Models and Methods. *European Journal of Operational Research*, 112, 1999.

[Ben01] Günther Bengel. *Verteilte Systeme*. Vieweg, 2001.

[Bes94] Christian Bessière. Arc-Consistency and Arc-Consistency Again. *Artificial Intelligence*, 65, 1994.

[BFBW92] Alan Borning, Bjørn N. Freeman-Benson und Molly Wilson. Constraint Hierarchies. *Lisp and Symbolic Computation*, 5(3), 1992.

[BFR95] Christian Bessière, Eugene C. Freuder und Jean-Charles Régin. Using Inference to Reduce Arc-Consistency Computation. In *Proc. Fourteenth International Joint Conference on Artificial Intelligence, IJCAI-95*, 1995.

[BG02] Thomas Barth und Manfred Grauer. Grid-Computing-Ansätze für verteiltes virtuelles Prototyping. In Schoder et al. [SFT02].

[BHK03] Klaus-Rainer Bräutigam, Günter Halbritter und Christel Kupsch. Simulationsrechnungen zu einigen Telematiktechniken und -diensten. *Internationales Verkehrswesen*, 55, 2003.

[BKL03] Ulrike Brüggemann, Stefan Kröpel und Harry Lehmann. Das Verkehrsnachfragemodell AVENA und seine Anwendung. *Internationales Verkehrswesen*, 55, 2003.

[BL97] Philippe Baptiste und Claude Le Pape. Constraint Propagation and Decomposition Techniques for Highly Disjunctive and Highly Cumulative Project Scheduling Problems. In Smolka [Smo97].

[BL00] Jens Braband und Karl Lennartz. Risikoorientierte Aufteilung von Sicherheitsanforderungen – ein Beispiel. *Signal und Draht*, 92, 2000.

[Bla83] Andrew Blake. The Least Disturbance Principle and Weak Constraints. *Pattern Recognition Letters*, 1, 1983.

[BM02] Maren Bolemant und Dirk Matzke. Modellierung und Simulation von Fahrplänen für den Eisenbahnverkehr mit constraint-basierten Verfahren. In *ASIM – Symposium Simulationstechnik*. SCS Europe, 2002.

[BMR97] S. Bistarelli, U. Montanari und F. Rossi. Semiring-based Constraint Solving and Optimization. *Journal of the ACM*, 44(2), 1997.

[BN00] Roberto Beraldi und Libero Nigro. Exploiting Temporal Uncertainty in Time Warp Simulations. In *Proc. Fourth IEEE International Workshop on Distributed Simulation and Real-time Applications DS-RT'00*, 2000.

[Bos94] H. Bossel. *Modellbildung und Simulation: Konzepte, Verfahren und Modelle zum Verhalten dynamischer Systeme.* Vieweg, 1994.

[BP02a] Randolf Berglehner und Johann Polz. Simulation des Sperrzeitverhaltens von Bahnübergängen. *Signal und Draht*, 94, 2002.

[BP02b] Norbert Bolln und Ferenc Paradi. Stellwerks- und Betriebssimulation für die Fahrdienstleiter-Schulung. *Signal und Draht*, 94, 2002.

[BP03] Jens Braband und Harald Peters. Experience with Quantified Safety Analyses. *Signal und Draht*, 95, 2003.

[BPA01] Alexander Bockmayr, Nicolai Pisaruk und Abderrahmane Aggoun. Network Flow Problems in Constraint Programming. In Walsh [Wal01].

[BPS01] Johann Berger, Norbert Plaminger und Péter Ernö Szilva. Stellwerksplanung der ÖBB wird durch BEST-Simulation unterstüzt. *Signal und Draht*, 93, 2001.

[BR75] J. R. Bitner und E. M. Reingold. Backtrack programming techniques. *Communications of the ACM*, 18, 1975.

[Bra93] Johannes Garrelt Braker. *Algorithms and Applications in Timed Discrete Event Systems.* Dissertation, Technische Universiteit Delft, 1993.

[BRJ99] Grady Booch, James Rumbaugh und Ivar Jacobson. *The Unified Modelling Language User Guide.* Addison-Wesley, 1999.

[Bro84] Manfred Broy, Hrsg. *Control Flow and Data Flow: Concepts of Distributed Programming.* NATO ASI Series, Springer, 1984.

[Bry84] R. E. Bryant. A Switch-Level Model and Simulator for MOS Digital Systems. *IEEE Transactions on Computers*, 33(2), 1984.

[Brü96] O. Brünger. *Konzeption einer Rechnerunterstützung für die Feinkonstruktion von Eisenbahnfahrplänen.* Dissertation, RWTH Aachen, 1996.

[BS03] Eduard G.J. Bouwman und Henk B. Scholten. Risk Management in Signalling Projects at ProRail. *Signal und Draht*, 95, 2003.

[Bun] Bundesrat. *Bundesrat, Länder.* `http://www.bundesrat.de`.

[BW90] J. C. M. Baeten und W. P. Weijland. *Process Algebra.* Cambridge University Press, 1990.

[Cal00] B. Callaghan. *NFS Illustrated.* Addison-Wesley, 2000.

[CD96] Philippe Codognet und Daniel Diaz. Compiling Constraints in clp(FD). *Journal of Logic Programming*, 27(3), 1996.

[CD01] Philippe Codognet und Daniel Diaz. Yet another Local Search Method for Constraint Solving. In *Proc. 1st Symposium on Stochastic Algorithms and Applications, SAGA01.* Springer, 2001.

[CDK01] George Coulouris, Jean Dollimore und Tim Kindberg. *Verteilte Systeme.* Pearson Education, 2001.

[Che03] Gilbert Chen. *New Methods for Parallel Discrete Event Simulation.* Dissertation, Rensselaer Polytechnic Institute, New York, 2003.

[CHI01] Cosytec. *CHIP System Documentation 5.4*, 2001.

[Chu99] Andy Hon Wai Chun. Constraint Programming in Java with JSolver. In *Proc. Practical Application of Constraint Technologies and Logic Programming, PACLP99*, 1999.

[Chv83] Vasek Chvatal. *Linear Programming.* W.H.Freeman, New York, 1983.

[CJ96] Assef Chmeiss und Philippe Jégou. Two New Constraint Propagation Algorithms Requiring Small Space Complexity. In *Proc. 8th IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, 1996.

[CJ97] Randy Chow und Theodore Johnson. *Distributed Operating Systems and Algorithms.* Addison-Wesley, 1997.

[CJL99] Yves Caseau, François-Xavier Josset und François Laburthe. CLAIRE: Combining Sets, Search, and Rules to Better Express Algorithms. In *Proc. International Conference on Logic Programming*, 1999.

[CL85] K. Mani Chandy und Leslie Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Science*, 3(1), 1985.

[CL96] Yves Caseau und François Laburthe. Cumulative Scheduling with Task Intervals. In *Logic Programing, Proceedings of the 1996 Joint International Conference and Syposium on Logic Programming.* The MIT Press, 1996.

[CM79] K. M. Chandy und J. Misra. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Transactions on Software Engineering*, 5(5), 1979.

[Col70] Alain Colmerauer. Les systèmes-q ou un formalisme pour analyser et synthétiser des phrases sur ordinateur. Technical Report 43, Département d'Informatique de l'Université de Montréal, 1970.

[Col84] Alain Colmerauer. Equations and Inequations on Finite and Infinite Trees. In *Proc. International Conference on Fifth Generation Computer Systems 1984.* North-Holland, 1984.

[CR93] Alain Colmerauer und Philippe Roussel. The Birth of Prolog. *SIGPLAN Notices*, 28(3), 1993.

[CTV98] Jean-François Cordeau, Palo Toth und Daniele Vigo. A Survey of Optimization Models for Train Routing and Scheduling. *Transportation Science*, 32(4):380–404, 1998.

[CVS] *Concurrent Versions System.* `http://www.cvshome.org/`.

[Dal94] Mario Dal Cin. Grundprobleme und Basismechanismen. In Wedekind [Wed94].

[DB97] Romuald Debruyne und Christian Bessière. Some Practicable Filtering Techniques for the Constraint Satisfaction Problem. In *Proc. Fifteenth International Joint Conference on Artificial Intelligence, IJCAI-97*, 1997.

[Dec90] Rina Dechter. Enhancement Schemes for Constraint Processing: Backjumping, Learning, and Cutset Decomposition. *Artificial Intelligence*, 41, 1990.

[Dec92] Rina Dechter. Constraint Networks. In *Encyclopedia of Artificial Intelligence*. Wiley and Sons, 1992. `http://www.ics.uci.edu/~csp/r17-survey.ps`.

[Dec99] Gebhard Decknatel. Modelling Train Movement with Hybrid Petri Nets. In *Proc. FME Rail Workshop*, 1999.

[Dec00] Rina Dechter, Hrsg. *Proc. Principles and Practice of Constraint Programming – CP'00.* Springer, 2000.

[Deua] Deutsche Bahn AG. *Daten und Fakten 2002.* `http://www.bahn-net.de/presse/pdf/daten_und_fakten_2002.pdf`.

[Deub] Deutscher Wetterdienst. *Numerische Wettervorhersage.* `http://www.dwd.de/de/FundE/Analyse/Modellierung/Modellierung.htm`.

[DG92] David J. DeWitt und Jim Gray. Parallel Database Systems: The Future of High Performance Database Processing. *Communications of the ACM*, 36(6), 1992.

[DH99] Birgit Demuth und Heinrich Hußmann. Using UML/OCL Constraints for Relational Database Design. In *Proc. 2nd Int'l Conf. Unified Modeling Language*, 1999.

[Die94] Heinrich Dietsch. Physische Eigenschaften Verteilter Rechensysteme: Beiwerk oder Basis? In Wedekind [Wed94].

[dis] *distributed.net.* `http://www.distributed.net/`.

[dLA99] J. de Lara und M. Alfonseca. Simulating Partial Differential Equations in the World-Wide Web. In *Proceedings of EUROMEDIA '99*, 1999.

[DLC91] E.H. Durfee, V.R. Lesser und D.D. Corkill. Distributed Problem Solving. *The Encyclopedia of Artificial Intelligence, Second Edition*, January 1991.

[dO01] Elias Silva de Oliveira. *Solving Single-Track Railway Scheduling Problem Using Constraint Programming.* Dissertation, The University of Leeds, UK, 2001.

[dOS01] Elias Silva de Oliveira und Barbara M. Smith. A Job-Shop Scheduling for the Single-track Railway Scheduling Problem. In Pavel Brazdil und Alípio Jorge, Hrsg., *EPIA*, Band 2258 of *Lecture Notes in Computer Science*. Springer, 2001.

[DS83] Randall Davis und Reid G. Smith. Negotiation as a Metaphor for Distributed Problem Solving. *Artificial Intelligence*, 20, 1983.

[DS98] Gebhard Decknatel und Eckehard Schnieder. Modellbildung und Simulationssysteme für den Schienenverkehr. *Eisenbahntechnische Rundschau*, 47:535–539, 1998.

[EFW98] Peggy S. Eaton, Eugene C. Freuder und Richard J. Wallace. Constraints and Agents: Confronting Ignorance. *AI Magazine*, 1998.

[ESS00] Karl-Heinz Erhard, York Schmidtke und Hans Strahberger. FALKO: Fahrplan-Validierung und -Konstruktion für spurgebundene Verkehrssysteme. *Signal und Draht*, 92, 2000.

[EUR] *EUROGRID – Application Testbed for European GRID computing.* `http://www.eurogrid.org`.

[FA97] Thom Frühwirth und Slim Abdennadher. *Constraint-Programmierung*. Springer, 1997.

[Fab99] Marko Fabiunke. Parallel Distributed Constraint Satisfaction. In *Proc. International Conference on Parallel and Distributed Processing Techniques and Applications PDPTA'99*, 1999.

[FBB92] Bjørn N. Freeman-Benson und Alan Borning. Integrating Constraints with an Object-Oriented Language. In Ole Lehrmann Madsen, Hrsg., *ECOOP*, Band 615 of *Lecture Notes in Computer Science*. Springer, 1992.

[FBKG02] Cèsar Fernàndez, Ramón Béjar, Bhaskar Krishnamachari und Carla P. Gomes. Communication and Computation in Distributed CSP Algorithms. In Van Hentenryck [Van02].

[FBMB90] Bjorn N. Freeman-Benson, John Maloney und Alan Borning. An Incremental Constraint Solver. *Communications of the ACM*, 33(1), 1990.

[Fer96] Alois Ferscha. Parallel and Distributed Simulation of Discrete Event Systems. In Zomaya [Zom96].

[FH00] Antonio J. Fernandez und Patricia M. Hill. A Comparative Study of Eight Constraint Programming Languages Over the Boolean and Finite Domains. *Constraints*, 5, 2000.

[FHK+02] Alan M. Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel und Toby Walsh. Global Constraints for Lexicographic Orderings. In Van Hentenryck [Van02].

[Fik68] Richard Fikes. *A Heuristic Program for Solving Problems Stated as Non-deterministic Procedures.* Dissertation, Carnegie Mellon University, 1968.

[Fip] Foundation for Intelligent Physical Agents. *Agent Communication Language Specifications.* `http://www.fipa.org/repository/aclspecs.html`.

[Fis01] George S. Fishman. *Discrete-Event Simulation.* Springer, 2001.

[FKT02] Ian Foster, Carl Kesselmann und Steven Tuecke. Die Anatomie des Grid. In Schoder et al. [SFT02].

[Fok00] Wan Fokkink. *Introduction to Process Algebra.* Springer, 2000.

[Fra86] Reinhold Franck. *Rechnernetze und Datenkommunikation.* Springer, 1986.

[Fre82] Eugene C. Freuder. A Sufficient Condition for Backtrack-Free Search. *Journal of the ACM*, 29(1):24–32, 1982.

[Fri00] Stefan Fricke. *Werkzeuggestützte Entwicklung kooperativer Agenten im Dienstkontext.* Dissertation, Technische Universität Berlin, 2000.

[Frü94] Thom Frühwirth. Temporal Reasoning with Constraint Handling Rules. Technical report, ECRC Munich, 1994.

[Frü95] Thom Frühwirth. Constraint Handling Rules. In A. Podelski, Hrsg., *Constraint Programming: Basics and Trends.* Springer, 1995.

[Frü98] Thom Frühwirth. Theory and Practice of Constraint Handling Rules. *The Journal of Logic Programming*, 37, 1998.

[Fuj90] Richard M. Fujimoto. Parallel Discrete Event Simulation. *Communications of the ACM*, 33(10):30–52, 1990.

[Gas77] John Gaschnig. A General Backtrack Algorithm That Eliminates Most Redundant Tests. In *Proc. 5th International Joint Conference on Artificial Intelligence*, 1977.

[Gas78] John Gaschnig. Experimental case studies of backtrack vs. Waltz-type vs. new algorithms for satisficing assignment problems. In *Proc. Second Canadian Conference on Artificial Intelligence*, 1978.

[GB65] Solomon W. Golomb und Leonard D. Baumert. Backtrack programming. *Journal of the ACM*, 12(4), 1965.

[Ger97] Carmen Gervet. Interval Propagation to Reason about Sets: Definition and Implementation of a Practical Language. *Constraints*, 1(3), 1997.

[Ges93] Ulrich Geske. *Prolog.* Akademie-Verlag, 1993.

[GGJ97] Ulrich Geske, Hans-Joachim Goltz und Ulrich John. Industrielle Anwendungen constraintbasierter Planung und Konfiguration. *Industrie Management*, 13(6), 1997.

[GH89] Hans-Joachim Goltz und Heinrich Herre. *Grundlagen der Logischen Programmierung*. Akademie-Verlag, 1989.

[Gig89] Peter Gigel. Simulation of Railway Networks with RWS-1. Technical report, Eidgenössische Technische Hochschule Zürich, 1989.

[Gin93] Matthew L. Ginsberg. Dynamic Backtracking. *Journal of Artificial Intelligence Research*, 1993.

[GJ79] M. R. Garey und D. S. Johnson. *Computers and Intractability. A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[GJ96] Hans-Joachim Goltz und Ulrich John. Methods for Solving Practical Problems of Job-Shop Scheduling in CLP(FD). In *Proc. Practical Application of Constraint Technology PACT'96*, 1996.

[GK00] Gunnar Gohlisch und Klaus Kämpf. Verkehrstelematik und Umwelt. *Internationales Verkehrswesen*, 52, 2000.

[Glo90] F. Glover. Tabu Search: A Tutorial. *Interfaces*, 20, 1990.

[GM99] Hans-Joachim Goltz und Dirk Matzke. University Timetabling Using Constraint Logic Programming. In *Proc. PADL'99*, 1999. `ftp://www.first.gmd.de/pub/plan`.

[Gol89] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[Gol95] Hans-Joachim Goltz. Reducing Domains for Search in CLP(FD) and its Application to Job-Shop Scheduling. In *Proc. Principles and Practice of Constraint Programming – CP'95*, 1995.

[Got01] Achim Gottschalk. Operative Simulation – Leistungssteigerung automatisierter Ablaufanlagen. *Signal und Draht*, 93, 2001.

[GSW00] Ian Gent, Kostas Stergiou und Toby Walsh. Decomposable Constraints. *Artificial Intelligence*, 123, 2000.

[GW93] Ian Gent und Toby Walsh. Towards an understanding of hill-climbing procedures for SAT. In *Proc. 11th National Conference on Artificial Intelligence AAAI-93*, 1993.

[Gör95] Günther Görz, Hrsg. *Einführung in die Künstliche Intelligenz*. Addison-Wesley, 1995.

[HaC] *HaCon Ingenieurgesellschaft mbH*. `http://www.hacon.de/`.

[Han00] Markus Hannebauer. Their Problems are my Problems - The Transition between Internal and External Conflict. In Cathérine Tessier, Laurent Chaudron und Heinz-Jürgen Müller, Hrsg., *Conflicting Agents: Conflict Management in Multi-Agent Systems*. Kluwer, 2000.

[Han01] Markus Hannebauer. *Autonomous Dynamic Reconfiguration in Collaborative Problem Solving.* Dissertation, Technische Universität Berlin, 2001.

[Har01] Volker Harder. Ausbildungssystem in Leitzentralen für den Nah- und Fernverkehr. *Signal und Draht*, 93, 2001.

[Hau00] Dirk Hauptmann. *Automatische und diskriminierungsfreie Ermittlung von Fahrplantrassen in beliebig großen Netzen spurgeführter Verkehrssysteme.* Dissertation, Universität Hannover, 2000.

[HB85] Kai Hwang und Faye A. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill, 1985.

[HE80] R. M. Haralick und G. L. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14, 1980.

[Hen88] Matthew Hennessy. *Algebraic Theory of Processes.* The MIT Press, 1988.

[HFV00] Dirk Helbing, Illes Farkas und Tamas Vicsek. Simulating Dynamical Features of Escape Panic. *Nature*, 407, 2000.

[HGTZ02] Klaus Hempelmann, Arnold Groß-Thebing und Hans Zimmer. Analyse der Fahrzeug/Fahrweg-Interaktion zur Ableitung von Maßnahmen mit dem Simulationswerkzeug SFE AKUSRAIL. *Eisenbahningenieur*, 52, 2002.

[HH89] Ralf Guido Herrtwich und Günter Hommel. *Kooperation und Konkurrenz.* Springer, 1989.

[HH98] Dirk Helbing und Bernardo A. Huberman. Coherent Moving States in Highway Traffic. *Nature*, 396, 1998.

[Hin77] G. E. Hinton. *Relaxation and its Role in Vision.* Dissertation, University of Edinburgh, 1977.

[HMSW03] Matthias Hoche, Henry Müller, Hans Schlenker und Armin Wolf. firstcs – A Pure Java Constraint Programming Engine. In *Proc. 2nd International Workshop on Multiparadigm Constraint Programming Languages MultiCPL'03*, 2003.

[Hoa84] C. A. R. Hoare. Notes on Communicating Sequential Systems. In Broy [Bro84].

[Hol95] Christian Holzbaur. OFAI clp(q,r) Manual. Technical Report TR-95-09, Austrian Research Institute for Articial Intelligence, 1995. `http://www.ai.univie.ac.at/clpqr/`.

[Hos00] Hiroshi Hosobe. A Scalable Linear Constraint Solver for User Interface Construction. In Dechter [Dec00].

[How98] Walter Hower. Revisiting Global Constraint Satisfaction. *Information Processing Letters*, 66(1), 1998.

[HY97] Katsutoshi Hirayama und Makoto Yokoo. Distributed Partial Constraint Satisfaction Problem. In Smolka [Smo97].

[HY00] Katsutoshi Hirayama und Makoto Yokoo. The Effect of Nogood Learning in Distributed Constraint Satisfaction. In *Proc. 20th IEEE International Conference on Distributed Computing Systems*, 2000.

[Hyd98] Dan Hyde. Introduction to the Programming Language Occam. Technical report, Computer Science Department, Bucknell University, 1998.

[Hür] Daniel Hürlimann. *OpenTrack – Simulation of Railway Networks.* `http://www.opentrack.ch/`.

[Hür01] Daniel Hürlimann. *Objektorientierte Modellierung von Infrastrukturelementen und Betriebsvorgängen im Eisenbahnwesen.* Dissertation, Eidgenössische Technische Hochschule Zürich, 2001.

[ILO] ILOG. *ILOG Solver.* `http://www.ilog.com/products/solver/`.

[Inm93] William H. Inmon. *Client/Server-Anwendungen.* Springer, 1993.

[J2Ea] SUN Microsystems. *The J2EE 1.4 Tutorial.* `http://java.sun.com/j2ee/1.4/docs/tutorial/doc`.

[J2Eb] SUN Microsystems. *Java 2 Platform, Enterprise Edition (J2EE).* `http://java.sun.com/j2ee/`.

[Jav] SUN Microsystems. *Java 2 Platform.* `http://java.sun.com/`.

[JCL] EPFL Lausanne. *Java Constraint Library.* `http://liawww.epfl.ch/~akira/JCL/`.

[JDB] SUN Microsystems. *JDBC Technology.* `http://java.sun.com/products/jdbc`.

[Jef85] David R. Jefferson. Virtual Time. *ACM Transactions of Programming Languages and Systems*, 7(3), 1985.

[Jen00] Eberhard Jentsch. Energiebedarf für Zugfahrten. *Eisenbahningenieur*, 51, 2000.

[JKC] *Java Constraint Kit.* `http://www.pms.informatik.uni-muenchen.de/software/jack/index.html`.

[JMSY92] J. Jaffar, S. Michaylov, P. J. Stuckey und R. H. C. Yap. The CLP(R) System and Language. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, 1992.

[Joh02] Ulrich John. *Konfiguration und Rekonfiguration mittels Constraint-basierter Modellierung.* Dissertation, Technische Universität Berlin, 2002.

[Jov03] Stanislav Jovanovic. Kostenreduktion bei der Oberbauinstandhaltung durch ECOTRACK. *Eisenbahningenieur*, 54, 2003.

[Kan01] Gene Kan. Gnutella. In Oram [Ora01].

[KCSÅ97] Per Kreuger, Mats Carlsson, Thomas Sjöland und Emil Åström. The TUFF Train scheduler – Trip Scheduling on Single Track Networks. In *Proc. Workshop on Industrial Constraint-Directed Scheduling*, 1997.

[KCSÅ01] Per Kreuger, Mats Carlsson, Thomas Sjöland und Emil Åström. Sequence Dependent Task Extensions for Trip Scheduling. Technical Report SICS-T–2001-14, Swedish Institute of Computer Science, 2001.

[KK99] George Karypis und Vipin Kumar. A Fast and Highly Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1), 1999.

[KK00] Angelika Klein und Uwe Köhler. Güterverkehrszentren und Verkehrssicherheit. *Internationales Verkehrswesen*, 52, 2000.

[Kla94] Volker Klahn. *Die Simulation großer Eisenbahnnetze*. Dissertation, Universität Hannover, 1994.

[KMM03] Walter Kik, Rainer Menssen und Dirk Moelle. Kräfte und Verschleiß in der Wendeschleife und im Abzweig einer Weiche. *Eisenbahningenieur*, 54, 2003.

[Koa] Koalog. *Koalog Constraint Solver*. `http://www.koalog.com/php/jcs.php`.

[Kor00] Holger Koriath. Die Anwendung der LCC-Methode für die Fahrbahn der DB AG. *Eisenbahningenieur*, 51, 2000.

[Kow74] Robert A. Kowalski. Predicate Logic as Programming Language. In *Proceedings of IFIP*. North Holland Publishing, 1974.

[KP00] Brian W. Kernighan und Rob Pike. *Programmier-Praxis*. Addison-Wesley, 2000.

[KPMS01] Holger Kruse, Karl Popp, Hans-Georg Matuttis und Alexander Schinner. Behandlung des Schotters als Vielkörpersystem mit wechselnden Bindungen. *Eisenbahningenieur*, 52, 2001.

[Kri89] E. V. Krishnamurthy. *Parallel Processing*. Addison Wesley, International Computer Science Series, 1989.

[Krö87] Fred Kröger. *Temporal Logic of Programs*. Springer, 1987.

[Krü01] Matthias Krüger. Langzeiteigenschaften der Schotterfahrbahn. *Eisenbahningenieur*, 52, 2001.

[Kum92] Vipin Kumar. Algorithms for Constraint Satisfaction Problems: A Survey. *AI Magazine*, 13(1), 1992.

[KvB95] Grzegorz Kondrak und Peter van Beek. A Theoretical Evaluation of Selected Backtracking Algorithms. In Chris Mellish, Hrsg., *IJCAI'95: Proc. International Joint Conference on Artificial Intelligence*, Montreal, 1995.

[Lam94a] Leslie Lamport. Introduction to TLA. Technical report, digital Systems Research Center, 1994. SRC Technical Note 1994-001.

[Lam94b] Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[LeL79] G. LeLann. An Analysis of Different Approaches to Distributed Computing. In *Proc. 1st International Conference on Distributed Computing Systems*, 1979.

[LHB93] Q. Y. Luo, P. G. Handry und J. T. Buchanan. Heuristic Search for Distributed Constraint Satisfaction Problems. Technical Report KEG-6-93, University of Strathclyde, Glasgow, 1993.

[Lor01] Wilhelm Lorsbach. Testen von Stellwerken für den Nahverkehr. *Signal und Draht*, 93, 2001.

[LPRS02] Claude Le Pape, Laurent Perron, Jean-Charles Régin und Paul Shaw. Robust and Parallel Solving of a Network Design Problem. In Van Hentenryck [Van02].

[Mac77] Alan K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8(1), 1977.

[Mas93] Jean-Luc Massat. Using Local Consistency Techniques to Solve Booelan Constraints. In Benhamou und Colmerauer [BC93].

[Mat89] Friedemann Mattern. *Verteilte Basisalgorithmen*. Springer, 1989.

[MB03] Dirk Matzke und Maren Bolemant. Modellierung innovativer Systemtechniken der Zugbeeinflussung mit constraint-logischer Programmierung. In *ASIM – Symposium Simulationstechnik*. SCS Europe, 2003.

[MC98] Jeff Matocha und Tracy Camp. A Taxonomy of Distributed Termination Detection Algorithms. *Systems and Software*, 43(3), 1998.

[MD01] Nihar R. Mahapatra und Shantanu Dutt. An Efficient Delay-Optimal Distributed Termination Detection Algorithm. Technical Report 2001-16, University of Illinois at Chicago, 2001. `http://citeseer.nj.nec.com/296827.html`.

[MH01] Nelson Minar und Marc Hedlund. A Network of Peers. In Oram [Ora01].

[Mil80] Robin Milner. *A Calculus of Communicating Systems*. Springer LNCS, 1980.

[Mis86] J. Misra. Distributed Discrete-Event Simulation. *ACM Computing Surveys*, 18(1), 1986.

[MJPL92] Steven Minton, Mark D. Johnston, Andrew B. Philips und Philip Laird. Minimizing Conflicts: A Heuristic Repair Method for Constraint Satisfaction and Scheduling Problems. *Artificial Intelligence*, 58(1-3):161–205, 1992.

[ML98] Dorothy L. Mammen und Victor R. Lesser. Problem Structure and Subproblem Sharing in Multi-Agent Systems. In *Proc. 3rd International Conference on Multi-Agent Systems, ICMAS'98*, 1998.

[Mon94] Markus Montigel. *Modellierung und Gewährleistung von Abhängigkeiten in Eisenbahnsicherungsanlagen*. Dissertation, Eidgenössische Technische Hochschule Zürich, 1994.

[Mor93] Paul Morris. The Breakout Method for Escaping from Local Minima. In *Proc. National Conference on Artificial Intelligence, AAAI-93*, 1993.

[MR88] James L. McClelland und David E. Rumelhart. *Explorations in Parallel Distributed Processing*. The MIT Press, 1988.

[MRR90] B. C. Merrifield, S. B. Richardson und J. B. G. Roberts. Quantitative Studies of Discrete Event Simulation Modelling of Road Traffic. In *Proc. SCS Multiconference on Distributed Simulation*, 1990.

[MS98] Kim Marriot und Peter J. Stuckey. *Programming with Constraints*. The MIT Press, 1998.

[Muk04] Patrick Mukherjee. Constraint-basierte Simulation von Eisenbahnfahrten: Hierarchische Problemzerlegung und -Lösung. Diplomarbeit, Technische Universität Berlin, 2004.

[Myr] *Myrinet*. `http://www.myri.com/myrinet/`.

[MZ03] Amnon Meisels und Roie Zivan. Asynchronous Forward-Checking on DisCSPs. Technical report, Ben-Gurion University of the Negev, Israel, 2003. `http://www.cs.bgu.ac.il/~am/papers.html`.

[Nar01] Alexander Nareyek. *Constraint-Based Agents – An Architecture for Constraint-Based Modeling and Local-Search-Based Reasoning for Planning and Scheduling in Open and Dynamic Worlds*. Springer, 2001.

[ND95] Thang Nguyen und Yves Deville. A Distributed Arc-Consistency Algorithm. Technical report, Département d'Ingénerie Informatique, Université Catholique de Louvain, 1995.

[NET] Microsoft Corp. *.NET*. `http://www.microsoft.com/net/`.

[NR96] K. Nökel und A. Rehkopf. Betriebsführung von Nahverkehrssystemen mit dem Simulationswerkzeug TRANSIT. *Signal und Draht*, 3, 1996.

[NZ01] Udo Nackenhorst und Bernd Zastrau. Numerische Parameterstudien zur Beanspruchung von Rad und Schiene beim Rollkontakt. *Eisenbahningenieur*, 52, 2001.

[Ora01] Andy Oram, Hrsg. *Peer-to-Peer*. O'Reilly, 2001.

[Osb02] Jörg Osburg. Untersuchung des Leistungsverhaltens beliebiger Signaltechniken. *Signal und Draht*, 94, 2002.

[Pac00] Jörn Pachl. *Systemtechnik des Schienenverkehrs*. B. G. Teubner, 2000.

[Pac02] Jörn Pachl. *Railway Operation and Control*. VTD Rail Publishing, 2002.

[PD00] Larry S. Peterson und Bruce S. Davie. *Computernetze*. dpunkt-Verlag, 2000.

[Pel00] David Peleg. *Distributed Computing – A Locality-Sensitive Approach*. Society for Industrial and Applied Mathematics, 2000.

[Pes01] Gilles Pesant. A Filtering Algorithm for the Stretch Constraint. In Walsh [Wal01].

[Pet62] Carl Adam Petri. *Kommunikation mit Automaten*. Dissertation, Institut für instrumentelle Mathematik, 1962.

[Pro93] Patrick Prosser. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence*, 9(3), 1993.

[Pro95] Patrick Prosser. Forward Checking with Backmarking. In *Constraint Processing, Selected Papers*. Springer, 1995.

[Pug94] Jean-Francois Puget. A C++ implementation of CLP. In *Proc. 2nd Singapore International Conference on Intelligent Systems (SPICIS)*, 1994.

[PVM] *PVM (Parallel Virtual Machine)*. `http://www.csm.ornl.gov/pvm/pvm_home.html`.

[Pöp99] Christoph Pöppe. Fahrplan-Algebra. *Spektrum der Wissenschaft, Digest: Wissenschaftliches Rechnen*, 2:18–21, 1999.

[Qua] University of Oxford, England. *Centre for Quantum Computation*.

[Rai] RMCon GmbH. *RailSys – Fahrplan- und Infrastruktur-Management*. `http://www.rmcon.de/uber_railsys.html`.

[RAS] HaCon Ingenieurgesellschaft mbH. *RASIM: Analyse und Planung von Betriebskonzepten und -anlagen mittels Rangiersimulation*. `http://www.hacon.de/rasim/index.shtml`.

[Reh98] Andreas Rehkopf. Betriebliche Simulation für den spurgeführten Verkehr. *Signal und Draht*, 90:10–15, 1998.

[Rei98] Wolfgang Reisig. *Elements of Distributed Algorithms.* Springer, 1998.

[Rin03] Georg Ringwelski. *Asynchrones Constraintlösen.* Dissertation, Technische Universität Berlin, 2003.

[Rip00] Burchard Ripke. Ziele der LCC Berechnung für die Fahrbahn der DB AG. *Eisenbahningenieur*, 51, 2000.

[RJB99] James Rumbaugh, Ivar Jacobson und Grady Booch. *The Unified Modelling Language Reference Manual.* Addison-Wesley, 1999.

[RMI] *Java Remote Method Invocation (RMI).* `http://java.sun.com/products/jdk/rmi/`.

[RPD90] Francesca Rossi, Charles Petrie und Vasant Dhar. On the Equivalence of Constraint Satisfaction Problems. In *Proc. European Conference on Artificial Intelligence*, 1990.

[RR96] E. Thomas Richards und Barry Richards. Nogood Learning for Constraint Satisfaction. In *Proc. Workshop on Constraint Programming Applications*, Cambridge, Massachusetts, 1996.

[RR00a] Thomas Rauber und Gudula Rünger. *Parallele und verteilte Programmierung.* Springer, 2000.

[RR00b] Jean-Charles Régin und Michel Rueher. A Global Constraint Combining a Sum Constraint and Difference Constraints. In Dechter [Dec00].

[RS00a] Georg Ringwelski und Hans Schlenker. Type Inference in CHR Programs for the Composition of Constraint Systems. In *Proc. 15. Workshop Logische Programmierung*, Berlin, 2000.

[RS00b] Georg Ringwelski und Hans Schlenker. Using Typed Interfaces to Compose CHR Programs. In *Proc. Second Workshop on Rule-Based Constraint Reasoning and Programming*, Singapore, 2000.

[RS02] Georg Ringwelski und Hans Schlenker. Dynamic Distributed Constraint Satisfaction with Asynchronous Solvers. In *ERCIM/CologNet Workshop on Constraint Solving and Constraint Logic Programming*, Cork, 2002.

[Ruh04] Julia Ruhmane. Constraint-basierte Simulation von Eisenbahnfahrten: Analyse und Dezentralisierung des Kontroll-Algorithmus. Diplomarbeit, Technische Universität Berlin, 2004.

[RW91] Bernd Rosenstengel und Udo Winand. *Petri-Netze.* Vieweg, 1991.

[Rég94] Jean-Charles Régin. A Filtering Algorithm for Constraints of Difference in CSPs. In *Proc. 12th National Conference on Artificial Intelligence*, 1994.

[Rég98] Jean-Charles Régin. Minimization of the Number of Breaks in Sports Scheduling Problems using Constraint Programming. In *Proc. DIMACS Workshop on Constraint Programming and Large Scale Discrete Optimization*, 1998.

[San94] Michael Sanella. SkyBlue: A Multi-Way Local Propagation Constraint Solver for User Interface Construction. In *Proceedings of UIST'94*, 1994.

[Sar93] Vijay A. Saraswat, Hrsg. *Concurrent Constraint Programming.* The MIT Press, 1993.

[Sch78] Steven Schoemaker. On Simulation. In Steven Schoemaker, Hrsg., *Computer Networks and Simulation.* North-Holland Publishing, 1978.

[Sch99] Matthias Schmauss. An Implementation of CHR in Java. Diplomarbeit, Ludwig Maximilians Universität München, 1999.

[Sch00] Hans Schlenker. Reduce-To-The-Max: Ein schneller Algorithmus für Multi-Ressourcen-Probleme. In *Proc. 14. Workshop Logische Programmierung*, Würzburg, 2000.

[Sch02a] Hans Schlenker. Distributed Constraint-based Railway Simulation. In *Proc. Principles and Practice of Constraint Programming – CP'02*, Seite 762. Springer LNCS 2470, 2002.

[Sch02b] Christian Schulte. *Programming Constraint Services.* Springer Verlag, 2002. PhD Thesis.

[Sch03] Hans Schlenker. Distributed Constraint-based Railway Simulation. In *Proc. Principles and Practice of Constraint Programming – CP'03*, Seite 995. Springer LNCS 2833, 2003.

[SET] *SETI@home – The Search for Extraterrestrial Intelligence.* `http://setiathome.ssl.berkeley.edu/`.

[SF88] Alois Schütte und Detlef Feldmann. *Occam2 Softwaretools.* McGraw-Hill, 1988.

[SFT02] Detlef Schoder, Kai Fischbaum und Rene Teichmann, Hrsg. *Peer-to-Peer.* Springer, 2002.

[SG91] Rok Sosic und Jun Gu. 3,000,000 Queens in Less Than One Minute. *SIGART Bulletin*, 2(2), 1991.

[SGM96] Gadi Solotorevsky, Ehud Gudes und Amnon Meisels. Modeling and Solving Distributed Constraint Satisfaction Problems. In *Proc. Principles and Practice of Constraint Programming – CP'96.* Springer, 1996.

[SGO02] Hans Schlenker, Hans-Joachim Goltz und Joerg-Wilhelm Oestmann. TAME – Time Resourcing in Academic Medical Environments. In *Proc. Artificial Intelligence in Medicine, AIME01.* Springer LNCS 2101, 2002.

[Shi01] Clay Shirky. Listening to Napster. In Oram [Ora01].

[SIC] Swedish Institute of Computer Science (SICS). *SICStus Prolog*. `http://www.sics.se/sicstus/`.

[Sin95] Moninder Singh. Path Consistency Revisited. In *Proc. 7th IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, 1995.

[SKK00a] Kirk Schloegel, George Karypis und Vipin Kumar. Graph Partitioning for High Performance Scientific Simulations. In *CRPC Parallel Computing Handbook*. Morgan Kaufmann, 2000.

[SKK00b] Kirk Schloegel, George Karypis und Vipin Kumar. Parallel Multilevel Algorithms for Multi-constraint Graph Partitioning. In *Proc. European Conference on Parallel Processing Euro-Par*, 2000.

[SKK01] Kirk Schloegel, George Karypis und Vipin Kumar. Graph Partitioning for Dynamic, Adaptive and Multi-phase Scientific Simulations. In *Proc. IEEE International Conference on Cluster Computing*, 2001.

[SKÅD98] T. Sjöland, P. Kreuger, E. Åström und P. Danielsson. Coordination of scheduling and allocation agents. In *2nd International Workshop on Constraint Programming for Time Critical Applications COTIC*, 1998.

[Sma] SmartDraw.com. *Business + Charting – Gantt Charts + Timelines*. `http://www.smartdraw.com/`.

[SMFBB93] Michael Sannella, John Maloney, Bjorn Freeman-Benson und Alan Borning. Multi-Way versus One-way Constraints in User-interfaces: Experiences with the DeltaBlue Algorithm. *Software Practice and Experience*, 1993.

[Smo97] Gert Smolka, Hrsg. *Proc. Principles and Practice of Constraint Programming – CP'97*. Springer, 1997.

[SR00a] Hans Schlenker und Frank Rehberger. Generalising Asynchronous Weak Commitment Search: from Intensional to Extensional Constraint Representation. In *Proc. 15. Workshop Logische Programmierung*, Berlin, 2000.

[SR00b] Hans Schlenker und Georg Ringwelski. New Reasons for the Introduction of Concurrency in Prolog. In *Proc. Workshop on Concurrency Specification and Programming*, Berlin, 2000.

[SR02a] Hans Schlenker und Georg Ringwelski. Global Data Structures for CLP in SICStus. In *Proc. 17. Workshop Logische Programmierung*, Dresden, 2002.

[SR02b] Hans Schlenker und Georg Ringwelski. POOC - A Platform for Object-Oriented Constraint Programming. In *ERCIM/CologNet Workshop on Constraint Solving and Constraint Logic Programming*. Springer LNCS 2627, 2002.

[SRSF91] K. Sycara, S. Roth, N. Sadeh und M. Fox. Distributed Constrained Heuristic Search. *IEEE Transactions on Systems, Man and Cybernetics*, 21(6), 1991.

[SS94] Leon Sterling und Ehud Shapiro. *The Art of Prolog.* The MIT Press, 1994.

[SSW99] Joachim Schimpf, Kish Shen und Mark Wallace. *Tutorial on Search in Eclipse*, 1999.

[Ste99] Kathleen Steinhöfel. *Stochastic Algorithms in Scheduling Theory.* Dissertation, Technische Universität Berlin, 1999.

[SUNa] SUN Microsystems. *Java Servlet Technology.* `http://java.sun.com/products/servlet/`.

[Sunb] SUN Microsystems. *Sun Fire 15K Server.* `http://www.sun.com/servers/highend/sunfire15k/`.

[Sut63] Ivan E. Sutherland. *Sketchpad: A Man-Machine Graphical Communication System.* Dissertation, Lincoln Laboratory, Massachusetts Institute of Technology, 1963.

[SWSM96] Howard Jay Siegel, Lee Wang, John John E. So und Muthucumaru Maheswaran. Data Parallel Algorithms. In Zomaya [Zom96].

[Tan00] Andrew S. Tanenbaum. *Computernetzwerke.* Pearson Education, 2000.

[Tur96] Louis H. Turcotte. Cluster Computing. In Zomaya [Zom96].

[TvS03] Andrew S. Tanenbaum und Marten van Steen. *Verteilte Systeme.* Pearson Education, 2003.

[TWF97] Marc Torrens, Rainer Weigel, und Boi Faltings. Java Constraint Library: Bringing Constraint Technology on the Internet Using Java. In *Proc. CP-97 Workshop on Constraint Reasoning on the Internet*, 1997.

[Van89] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

[Van02] Pascal Van Hentenryck, Hrsg. *Proc. Principles and Practice of Constraint Programming – CP'02.* Springer, 2002.

[VD91] Pascal Van Hentenryck und Yves Deville. The Cardinality Operator: A New Logical Connective for Constraint Logic Programming. In *Proc. International Conference on Logic Programming*, 1991.

[VH99] Peter Van Roy und Seif Haridi. Mozart: A Programming System for Agent Applications. In *Proc. International Workshop on Distributed and Internet Programming with Logic and Constraint Languages*, 1999.

[VS94] Gérard Verfaillie und Thomas Schiex. Dynamic Backtracking for Dynamic Constraint Satisfaction Problems. In *Proceedings of the European Conference on Artificial Intelligence Workshop on "Constraint Satisfaction Issues Raised by Practical Applications"*, 1994.

[Wal72] David L. Waltz. *Generating Semantic Descriptions From Drawings of Scenes With Shadows.* Dissertation, Massachusetts Institute of Technology, 1972. MIT Technical Report AI271.

[Wal75] David L. Waltz. Understanding Line Drawings of Scenes with Shadows. *The Psychology of Computer Vision*, 1975. `http://www.neci.nj.nec.com/homepages/waltz/SFXC7.pdf`.

[Wal01] Toby Walsh, Hrsg. *Proc. Principles and Practice of Constraint Programming – CP'01.* Springer, 2001.

[Wal03] Toby Walsh. Constraint Patterns. In *Proc. Principles and Practice of Constraint Programming – CP'03*, Seiten 53–64. Springer LNCS 2833, 2003.

[War77] David H. D. Warren. *Implementing Prolog - Compiling Predicate Logic Programs.* Dissertation, University of Edinburgh, 1977.

[Wed94] Hartmut Wedekind, Hrsg. *Verteilte Systeme.* BI-Wissenschafts-Verlag, 1994.

[Wed02] Sebastian Wedeniwski. ZetaGrid. In Schoder et al. [SFT02].

[Wei] Eric W. Weisstein. *Four-Color Theorem.* Wolfram Research. `http://mathworld.wolfram.com/Four-ColorTheorem.html`.

[WM94] Mark Weigelt und Peter Mertens. Dezentrale Produktionssteuerung mit dem Agenten-System DEPRODEX-1. In Wedekind [Wed94].

[Wol98] Armin Wolf. Solving Hierarchies of Finite-Domain Constraints. *Journal of Experimental and Theoretical Artificial Intelligence*, 10(1), 1998.

[Wol99] Armin Wolf. *Konfiguration und Rekonfiguration mittels Constraint-basierter Modellierung.* Dissertation, Technische Universität Berlin, 1999.

[Wol01a] Armin Wolf. Adaptive Constraint Handling with CHR in Java. In Walsh [Wal01].

[Wol01b] Wilfried Wolter. Kollisionssicherheit von Schienenfahrzeugen. *Eisenbahningenieur*, 52, 2001.

[Wol03a] Armin Wolf. Pruning while Sweeping over Task Intervals. In *Proc. Principles and Practice of Constraint Programming – CP'03.* Springer, 2003.

[Wol03b] Armin Wolf. A Specialized Search Algorithm for Contiguous Task Scheduling Problems. In *Proc. Workshop of the ERCIM Working Group on Constraints*, 2003. `http://ws.ailab.sztaki.hu/Program.html`.

[YDIK92] Makoto Yokoo, Edmund H. Durfee, Toru Ishida und Kazuhiro Kuwabara. Distributed Constraint Satisfaction for Formalizing Distributed Problem Solving. In *12th IEEE International Conference on Distributed Computing Systems*, 1992.

[YDIK98] Makoto Yokoo, Edmund H. Durfee, Toru Ishida und Kazuhiro Kuwabara. The Distributed Constraint Satisfaction Problem: Formalization and Algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 10(5), September 1998.

[YH96] Makoto Yokoo und Katsutoshi Hirayama. Distributed Breakout Algorithm for Solving Distributed Constraint Satisfaction Problems. In *Proc. 2nd International Conference on Multiagent Systems ICMAS'96*, 1996.

[YH98] Makoto Yokoo und Katsutoshi Hirayama. Distributed Constraint Satisfaction Algorithm for Complex Local Problems. In *Proc. 3rd International Conference on Multi-Agent Systems, ICMAS'98*, 1998.

[YH00] Makoto Yokoo und Katsutoshi Hirayama. Algorithms for Distributed Constraint Satisfaction: A Review. *Autonomous Agents and Multi-Agent Systems*, 3(1), 2000.

[Yok94] Makoto Yokoo. Weak-Commitment Search for Solving Constraint Satisfaction Problems. In *Proc. National Conference on Artificial Intelligence, AAAI-94*, 1994.

[YSH02] Makoto Yokoo, Koutarou Suzuki und Katsutoshi Hirayama. Secure Distributed Constraint Satisfaction: Reaching Agreement without Revealing Private Information. In Van Hentenryck [Van02].

[ZM93] Ying Zhang und Alan K. Mackworth. Parallel and Distributed Finite Constraint Satisfaction: Complexity, Algorithms and Experiments. *Parallel Processing for Artificial Intelligence*, 1993.

[Zom96] Albert Y. Zomaya, Hrsg. *Parallel and Distributed Computing Handbook*. McGraw-Hill, 1996.

Index