Scotland Yard — PG 240
Informatik Lehrstuhl X
Uni Dortmund

Abschlußbericht der Projektgruppe 240 Scotland Yard — Evaluation und Entwurf von Werkzeugen für eine Entwicklungsumgebung zum Prototyping

Klaus Alfert Oliver Alsbach Jörn Bodemann
Markus Brameier Stefan Hedtfeld Peter Jodeleit
Marcus Kirsch Peter Neumann Dirk Niemann
Stefan Schüler Horst Sdun Knuth Waltenberg

Betreuer:
Prof. Dr. Ernst-Erich Doberkat
Dr. rer. nat. Wilhelm Hasselbring
Dipl.-Inf. Claus Pahl

29. März 1995

UNI DO

Universität Dortmund

Inhaltsverzeichnis

Abbildungsverzeichnis

Teil I

Einleitung und erste Seminarphase

1. Einleitung

Dieses Kapitel gibt einen kurzen Überblick in die Organisation und Aufgabenstellung der Projektgruppe (PG).

1.1 Die Organisation der Projektgruppe Scotland Yard

Die PG240 „Scotland Yard — Evalution und Entwurf von Werkzeugen für eine Entwicklungsumgebung zum Software Prototyping" wird im Sommersemester 1994 und im Wintersemester 1994/95 am Lehrstuhl für Software-Technologie der Universität Dortmund durchgeführt unter der Leitung von:

Prof. Dr. Ernst-Erich Doberkat
Dr. rer. nat. Wilhelm Hasselbring
Dipl.-Inf. Claus Pahl

Die Teilnehmer sind:

Klaus Alfert
Oliver Alsbach
Jörn Bodemann
Markus Brameier
Stefan Hedtfeld
Peter Jodeleit
Marcus Kirsch
Peter Neumann
Dirk Niemann
Stefan Schüler
Horst Sdun
Knuth Waltenberg

1.2 Aufgabe der Projektgruppe

Ziel der Projektgruppenarbeit ist es, Studenten sowohl selbständiges, strukturiertes Arbeiten als auch einen Praxisbezug zu Lehrinhalten zu vermitteln. Durch die Projektgruppe soll der Student vom Bearbeiten vorgegebener Aufgaben zum eigenverantwortlichen Erarbeiten und Lösen komplexer Aufgaben angeleitet werden unter Berücksichtigung der Kooperation mit anderen Studenten.

Die konkrete Aufgabenstellung der Projektgruppe 240 lautete wie folgt:

> „Die Projektgruppe soll zunächst ein komplexes Programm als Prototyp in der mengenorientierten Prototyping-Sprache ProSet entwickeln. Auf der Grundlage der hierbei gewonnenen Erfahrungen mit dem zugrundeliegenden Ansatz zum Software Prototyping sollen in einem zweiten Schritt Werkzeuge in einer Softwareentwicklungsumgebung, die exploratives Programmieren in ProSet unterstützt, spezifiziert werden."

Das zu entwickelnde Programm, eine Implementierung des von Ravensburger bekannten Strategiespiels *Scotland Yard* [Spi83], soll mit Hilfe des Prototyping-Ansatzes realisiert werden. Durch Prototyping entsteht ein ausführbares Modell, das wesentliche Eigenschaften des endgültigen Produktes hat. Prototyping bietet sich dann besonders gut an, wenn zur Problemlösung mit verschiedenen Algorithmen experimentiert werden muß, um beispielsweise verschiedene Planungsalgorithmen zu testen.

Im ersten Semester wurde also mit Hilfe des Prototyping-Ansatzes eine komplexe Planungsaufgabe bearbeitet, wobei sich die Verteilung der Planungsaufgabe auf mehrere Detektivprozesse anbot.

1.3 Überblick der Projektgrupppenarbeit

Die erste Projektgruppenphase bestand in einer detaillierten Einarbeitung in vorgegebene Themengebiete, die in einem Kompaktseminar in Kleve von den PG-Teilnehmern vorgestellt wurden. Eine inhaltliche Zusammenfassung der einzelnen Seminarvorträge ist in Kapitel 2 zu finden.

Nach der Seminarphase folgte eine Einarbeitungsphase in die Sprache ProSet und der Benutzungsoberfläche für den Compiler. Unterstützt wurde diese Phase durch Vorträge zur Sprachbeschreibung von ProSet. Dabei wurden fünf Gruppen gebildet, die die Themen *Sprachkern, Ausnahmebehandlung, Persistenz, Linda* und *Module* behandelten.

Die praktische Realisierung des Programms *Scotland Yard* wurde mit der Spezifikation eingeleitet. Zur Problemlösung wurden die drei Teilgruppen *Graphische Benutzeroberfläche* (Stefan H., Peter N., Dirk und Horst), *Treiber* (Jörn, Marcus, Peter J. und Knuth) sowie *Planung* (Klaus, Oliver, Markus und Stefan S.) gebildet (vgl. Abb. 1.1 auf Seite 14). Um einen guten Informationsfluß zu gewährleisten, gab es in jeder Teilgruppe

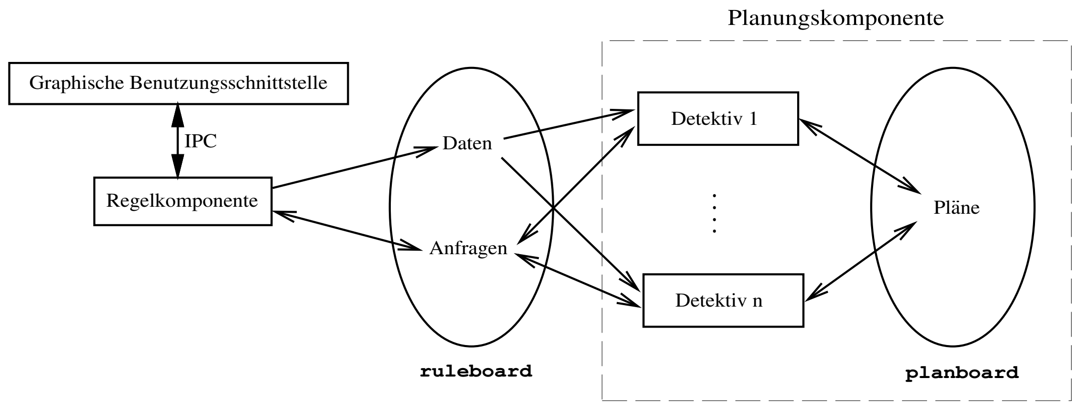


Abbildung 1.1: Architektur des Programms Scotland Yard

einen Kommunikator, der dafür Sorge zu tragen hatte, daß in seiner Teilgruppe jeder andere Teilnehmer über den Stand des Gesamtprojektes informiert war. Ziel dieser Phase war es, für den Prototypen folgende Aspekte zu entwerfen:

- Eine generelle Programmstruktur, die den Zusammenhang zwischen der grafischen Benutzungsoberfläche sowie der Treiber- und Planungskomponente verdeutlicht
- Eine Interprozeßkommunikation (IPC) zwischen der grafischen Benutzungsoberfläche und den in ProSet gehaltenen Programmteilen
- Datenstrukturen in ProSet zur Modellierung des Londoner Stadtplans sowie von Mister X und den Detektiven
- Planungsalgorithmen

Während der Implementierungsphase wurde entsprechend des Entwurfs der Prototyp in ProSet realisiert. In diese Phase fiel auch schon die Evaluation, weil konstruktive Kritik am ProSet-Compiler und der zugehörigen Benutzeroberfläche *xpst* immer wieder zu entsprechenden Modifizierungen führten. Unermüdlich lieferte die ProSet-Gruppe einen guten und schnellen Support bei allen anstehenden Problemen.

1.3.1 Termine

Für den zeitlichen Ablauf der PG im Sommersemester 1994 wurde folgender grober Rahmen gesteckt:

Einarbeitung:	4. April - 22. April
Entwurf:	25. April - 13. Mai
Implementierung:	16. Mai - 24. Juni
Evaluation:	27. Juni - 8. Juli

Für den zeitlichen Ablauf der PG im Wintersemester 1994/95 wurde folgender grober Rahmen gesteckt:

Seminarvorträge:	7. und 8. November
Spezifikation von Werkzeugen:	15. November bis 10. Februar
Exkursion an die Vrije Universiteit Amsterdam:	3. Februar - 5. Februar

Die PG-Teilnehmer trafen sich während der Vorlesungszeit zweimal wöchentlich, während der vorlesungsfreien Zeit höchstens einmal pro Woche; jedoch war die Teilnahme in der vorlesungsfreien Zeit freiwillig.

In den Sitzungen werden Aufgaben verteilt und der aktuelle Stand der Arbeit besprochen. Um die Ergebnisse festzuhalten, werden Protokolle angefertigt; diese werden der Reihe nach von jedem PG-Teilnehmer geschrieben und sowohl in die News-Gruppe *unido.informatik.lehre.pg240* gepostet als auch im PG-Ordner abgelegt.

2. Die erste Seminarphase

Die erste Aktivität bestand in der Durchführung eines Kompaktseminars, bei der jeder Teilnehmer ein Themengebiet erläuterte. Wie die Zusammenfassungen zeigen, wurde durch das Seminar ein grundlegender Einblick in das Arbeitsgebiet der Projektgruppe vermittelt.

Die zwölf Seminarvorträge umfaßten folgende Themenbereiche:

- Einführung in den Forschungsbereich *Prototyping* und ProSet
- Die Koordinationssprache Linda
- Kooperative Planung
- Software-Entwicklungsumgebungen und formale Spezifikation

2.1 Einführung in den Forschungsbereich *Prototyping* und ProSet

2.1.1 Der Software Life Cycle (Stefan Schüler)

Der Begriff *Software Engineering* entstand in den 60er Jahren, als sich die Entwicklung immer größerer Softwaresysteme zunehmend schwieriger gestaltete – das Schlagwort *Softwarekrise* schien immer treffender das Dilemma der Softwareentwicklung zu charakterisieren: Programmieren wurde als Kunst angesehen. Damit die Forschung nicht „durch einen Flaschenhals der Software stranguliert wurde“ [Bau93], sollte durch ingenieurmäßiges Vorgehen die Softwareproduktion zu qualitativ hochwertigen Programmen und Programmsystemen führen. Das Entstehen eines Softwareproduktes wurde von seiner Entstehung bis zu seiner Verwendung systematisch in verschiedene Stadien eingeteilt *(Software-Life-Cycle)*, und zwar in die drei Gruppen *Design, Production* und *Service* [SW93].

Diese drei Gruppen bilden auch die Grundlage für das Wasserfall-Modell, das in die Phasen *Analyse, Entwurf, Implementation, Installation* und *Wartung* eingeteilt ist. Jede dieser Phasen hat sowohl einen klar festgelegten Start- als auch Endpunkt. Im Wasserfall-Modell ist der Zyklus des Software-Life-Cycles unterbrochen und zu einer Sequenz wohldefinierter Tätigkeiten geformt worden. Es gibt allerdings Möglichkeiten, gewisse Rückkopplungen durchzuführen, um Änderungen oder Verbesserungen vorzunehmen [DF89].

Der Überblick zu den Phasen des Wasserfall-Modells zeigte, daß das Wasserfall-Modell idealisiert ist und die Realität nicht widerspiegelt, weil es in der Praxis nur annäherungsweise angewendet werden kann. Es funktioniert eigentlich nur dann, wenn alle Spezifikationen feststehen und diese auch nicht im Verlauf des Projektes geändert werden müssen. Wenn in späteren Phasen aufgrund von mangelhaften Entwürfen Probleme entstehen und diese lokal gelöst werden, führt dies unter Umständen zu Implementationen mit zerstörter Struktur. Ändert man hingegen nachträglich den Entwurf, so muß oft eine große Menge schriftlichen Materials geändert werden. Weitere Nachteile des Wasserfall-Modells sind:

- Kommunikation zwischen Entwickler und Benutzer sind von der Implementierungsphase an mangelhaft.
- Experimentelle oder evolutionäre Vorgehensweise wird nicht gefördert.
- Lauffähige Programme liegen erst in einer sehr späten Phase vor.

Um die genannten Nachteile aufzufangen, ist es nötig, das Modell zu manipulieren. Da die meisten Mängel der Analyse-Phase (Anforderungsdefinition, Spezifikation) zuzuordnen sind, ist es naheliegend in dieser Phase in den Software-Life-Cycle einzugreifen, indem man die Phase um mehrere Tätigkeiten ergänzt, und zwar um den Entwurf eines Prototypen sowie dessen Benutzung, Erprobung und Analyse [DF89]. Dadurch ist es sowohl gelungen, die strikte Trennung einzelner Phasen aufzuheben, als auch die Anforderungen an das zu erstellende System besser auszuloten, weil der Benutzer viel stärker in die Entwicklung einbezogen werden kann. Die Art der Vorgehensweise birgt allerdings auch die Gefahr für den Entwickler, „quick and dirty" zu programmieren, um schnell alternative Lösungen zu präsentieren. Ein weiterer Nachteil kann darin liegen, daß Anforderungen übersehen werden, weil Benutzer und Entwickler zu sehr auf das Modell fixiert sind.

Prototyping ist als Ergänzung zum Wasserfall-Modell zu sehen, bei dem die statische Trennung von Anforderungsdefinition, Systemdesign und Implementierung in eine Vorgehensweise konvergiert, durch die in der Analyse-Phase ein ablauffähiges Programm vorliegt, das schrittweise in Übereinstimmung mit dem Benutzer soweit verfeinert und entwickelt werden kann, bis das Anwendungsprogramm mit voller Funktionalität entstanden ist. Dabei kann man experimentell oder auch evolutionär vorgehen [BKKZ92].

Weitere Modelle zum Software Life Cycle, wie z.B. das Meta-Modell, sind in [GJM91] zu finden.

2.1.2 Prototyping-Zugänge (Knuth Waltenberg)

Als Grundlage diente [Flo84] und [HI88].

Einleitung

Prototyping ist eine Komponente der Software-Entwicklungsverfahren. Im Prototyping wird eine operative Version des Programmes schon zu einem frühen Zeitpunkt erstellt. Relevante Probleme werden dann experimentell gelöst. Weiterhin dient der *Prototyp* als Basis für die Diskussion zwischen Entwicklern und Benutzern.

Der Prozeß des Prototypings

Den Prozeß des Prototyping kann man in die vier Schritte *funktionale Auswahl*, *Konstruktion*, *Auswertung* und *weitere Benutzung* unterteilen. In der *funktionalen Auswahl (Functional Selection)* wird festgelegt, welche Funktionen des späteren Systems im Prototypen abgebildet werden sollen. Wenn man Software in einem Schichtenmodell betrachtet, kann man zwei Arten unterscheiden:

- Im *Horizontalen Prototyping* werden nur die Funktionen einer Schicht verwirklicht, z.B. eine Benutzungsschnittstelle.
- Im *Vertikalen Prototyping* werden dagegen komplette Teile des Zielsystems durch alle Schichten hindurch verwirklicht.

Ein weiterer Schritt ist die *Konstruktion* des Prototypen. In diesen Schritt sollte natürlich viel weniger Mühe investiert werden als in die Entwicklung des Endproduktes. Dies kann erreicht werden durch eine entsprechende Auswahl der Funktionen und durch Anwendung geeigneter Techniken und Werkzeuge. Es können qualitative Einschränkungen gemacht werden.

Die *Auswertung* des Prototypen ist der wichtigste Schritt, denn sie liefert das Feedback für den weiteren Entwicklungsprozeß. An ihr sollten alle relevanten Gruppen voraussichtlicher Benutzer teilnehmen.

Der letzte Schritt ist die *weitere Benutzung* des Prototypen. Hier stehen zwei Möglichkeiten offen:

- Der Prototyp ist nur als Lerngegenstand gedacht und wird am Ende weggeworfen oder
- er wird teilweise oder ganz im Zielsystem weiterbenutzt.

Die Methoden des Prototypings

Bei den Methoden des Prototypings können je nach dem Ziel, welches verfolgt wird, drei Arten unterschieden werden: *Erforschungs-* , *Experimentier-* und *Entwicklungs-Prototyping*.

Haben die Entwickler ein zu geringes Wissen über das spätere Anwendungsfeld des Systems und die Benutzer keine genaue Vorstellung was der Rechner überhaupt für sie tun soll, dann wird das *Entwicklungs-Prototyping* angewendet.

Hier liegt das Hauptaugenmerk auf der Klärung der Erfordernisse und wünschenswerten Funktionen des Zielsystems und der Diskussion alternativer Lösungen. Eine praktische Demonstration möglicher Systeme soll Ideen und eine bessere Zusammenarbeit zwischen allen Parteien fördern.

Beim *Experimentellen Prototyping* sollen geeignete Lösungen für Anwenderprobleme durch Experimentieren gefunden werden. Die Benutzer können durch die Versuche ihre Ideen genauer spezifizieren und die Entwickler die technische Durchführbarkeit abschätzen. Auch hier kann man verschiedene Arten unterscheiden:

- Die volle Funktionssimulation,
- die Anwender-Schnittstellen-Simulation,
- die Skelett-Programmierung,
- die Basismaschinen-Konstruktion und
- die partielle Funktionssimulation.

Das *Evolutionäre Prototyping* ist eine Entwicklung in *Versionen*. Es verläßt die linerare Stufenform der Softwareentwicklung und verändert sie zu einer*Entwicklung in Zyklen*. Je nach der Anzahl der Zyklen kann man zwischen *inkrementeller Systementwicklung* und *evolutionärer Systementwicklung* unterscheiden.

Die Techniken und Werkzeuge

Bei der Betrachtung der Techniken und Werkzeuge des Prototypings sollte man die Werkzeuggruppen für das Funktions- und Schnittstellenprototyping gesondert betrachten. Im Funktionsprototyping stehen folgende Arten zur Verfügung:

- Ausführbare Spezifikationen,
- Abstrakte Modelle,
- Very High Level Languages (VHLL),
- Anwendungsorientierte VHLLs (AHLL),
- Funktionale Sprachen,
- Tool-Set-Methode und
- wiederverwendbare Software.

Das Schnittstellenprototyping nutzt dagegen folgende Werkzeuge:

- Simulation,
- Formale Grammatiken,
- Status-Übergangsprotokolle und
- Bildschirmgeneratoren (GUI-Builder).

2.1.3 Evolutionäres Prototyping und weitere Aspekte (Peter Jodeleit)

Als Grundlage diente [BKKZ92].

Was ist mit dem Begriff ëvolutionär„ gemeint?

- Während der gesamten Laufdauer eines Projektes findet eine intensive Kommunikation zwischen Entwicklern und späteren Benutzern statt,
- der Entwicklungsprozeß ist auch gleichzeitig ein Lernprozeß für alle beteiligten Gruppen (im Gegensatz zu einem linearen Prozeß von der Spezifikation zum fertigen Zielsystem),
- die Entwicklung geht in kleinen Schritten von überschaubarer Größe voran, wodurch ein einfaches Korrigieren oder Neudefinieren der Entwicklungsrichtung ermöglicht wird und
- die gesamte Entwicklung wird, soweit möglich durch (möglichst operationale) Software ergänzt (Prototypen).

Ansätze für Evolutionäres Prototyping in der Praxis

Derzeit gibt es in der Praxis zwei Entwicklungsarten, in denen sich die Ziele des Evolutionären Prototypings wiederspiegeln.

- Alpha - Beta - Test
  Hier gibt es eine interne Alpha-Version und eine Beta-Version, die durch typische Benutzergruppen extern getestet wird. Deren Erfahrungen fließen in den weiteren Entwicklungsprozeß ein. Die Vorteile, die sich ergeben sind neben der gesteigerten Qualität zum einen der Wettbewerbsvorteil der Testnutzer, ein besserer Support und eine Reflektion ihrer eigenen Arbeit, für die Entwickler zum anderen nicht zuletzt ein Marktvorteil durch "Referenznutzer“.
- Versioning
  Ähnlich dem Alpha - Beta - Test. Hier fließt das Feedback von Käufern in die

jeweils neueste Version des Produktes ein. Hierbei gewinnt auch immer mehr das indirekte Feedback durch Spezialmagazine und Newsgroups an Bedeutung. Die Verwendung des Begriffes „Prototyps“ ist hier aber kritisch, da jede Version ein komplettes und lauffähiges Produkt sein muß, welches durch die nächste Version verbessert und erweitert wird.

Argumente für Prototyping

Welche Argumente sprechen nun für Prototyping bei der Softwareentwicklung? Zum einen sind die traditionellen Ansätze zu unflexibel, zum anderen wird die Erstellung von Prototypen durch immer günstigere und leistungsfähigere Hardware und immer leistungsfähigere Software (wie high performance Betriebssysteme, moderne Programmiersprachen und Programmiertools) überhaupt erst ermöglicht und in Zukunft auch immer einfacher werden. Weiterhin hat sich die Zielgruppe von Softwareprodukten und damit die Anforderungen an diese Produkte geändert. Es sind nicht mehr nur noch Spezialisten, die Software bedienen. Dementsprechend haben sich auch die Anforderungen der Benutzer geändert. Es sind nun leichte Bedienbarkeit, Intelligenz, Änderbarkeit und Kompatibilität der Produkte mehr denn je gefragt.

Techniken, um Prototypen zu Implementieren

Durch verschiedene Techniken kann die Erstellung von Prototypen vereinfacht werden. Dazu zählen

- Objekt-Orientiertes-Design
  Techniken wie message-passing, Kapselung in Klassen und Vererbung zwischen verschiedenen Klassen ermöglichen eine effiziente und verständliche Programmierung. Zudem ermöglicht OO-Design auch eine einfachere Kommunikation zwischen Benutzern und Entwicklern.

- Client-Supplier-Model
  Eine Software setzt sich hier aus verschiedenen Softwarekomponenten zusammen. Jede Komponente stellt anderen Komponenten Funktionen zur Verfügung und benutzt Funktionen anderer Komponenten. Nur die Schnittstellen sind gegenseitig bekannt. Werden die Schnittstellen frühzeitig spezifiziert, können Komponenten als Prototypen implementiert werden.

- Trennung von Interaktion und Funktionalität
  Hier werden Benutzerkommunikation und funktioneller Kern getrennt. Die Kommunikationskomponente kann so individuell angepaßt werden, ohne daß weitreichende Änderungen an der übrigen Software nötig wird. Die Oberflächen sollten dabei immer gleichen Richtlinien folgen, um leichtere Bedienbarkeit und Portierbarkeit zu erreichen.

- Model-View-Controller Paradigma
  Wie beim vorherigen Punkt, hier wird die interaktive Komponente jedoch noch weiter aufgeteilt (in model, view und controller). Durch die unabhängige Implementierung kann jede Komponente als unabhängiger Prototyp realisiert werden, dessen Änderung die Entwicklung der anderen Komponenten nicht beeinflußt. Die Entwicklungszeit sinkt und zusätzlich wird die Portierung vereinfacht.

Very High Level Languages

Prototyping wird durch hochentwickelte Porgrammiersprachen (very high level languages = VHLLs) mit komfortablen Entwicklungsumgebungen unterstützt und vereinfacht. Das Ziel ist es, möglichst einfach und schnell zu ausführbaren Prototypen zu kommen. Dies bedeutet z.B. daß auch nicht komplette Programme ausführbar sind, daß Interpreter oder inkrementelle Compiler zur Verfügung stehen oder daß es Möglichkeiten zu kleinen Compiliereinheiten gibt. Die Anforderungen an VHLLs sind die Generierung von kompakten Programmen, ein einfaches und gut überprüfbares semantisches Modell, minimaler expliziter Kontrollfluß, „mächtige“ Datentypen und einfache Schnittstellen zu anderen Sprachen. Für die effiziente Erstellung von Prototypen spielt auch die Programmierumgebung eine wichtige Rolle. VHLLs lassen sich in vier Hauptgruppen aufteilen:

- Logische Sprachen (z.B. Prolog)
- Funktionale Sprachen (meist basierend auf dem Lambda-Kalkül, z.B. Scheme, ML)
- Algebraische Sprachen (z.B. II an der Universität Dortmund)
- und andere.

Vielen VHLLs fehlen zum effizienten Protoyping noch die mächtigen und umfangreichen Bibliotheken, wie sie für andere Porgrammiersprachen existieren.

2.1.4 ProSet (Marcus Kirsch)

Einführung

Die Programmiersprache ProSet [DWH+94] (ProSet ist eine Abkürzung für *Prototyping with Sets*) ist der Kern einer Entwicklungsumgebung zur Unterstützung evolutionären Prototypings. ProSet erlaubt es, durch natürliche Programmierung Prototypen als ausführbare Modelle zu formulieren. Natürliche Programmierung bedeutet hier, daß ProSet auf einem gängigen mathematischen Kalkül, nämlich der endlichen Mengenlehre, beruht und sich an die in der Mathematik übliche Notation anlehnt. Diese Eigenschaften kommen dem für evolutionäres Prototyping typischen Rückkopplungsprozeß zwischen Entwickler und Benutzer entgegen, weil diese als ausführbare Modelle gegebenen Prototypen über eine hohe Lesbarkeit verfügen.

Beispiele für die im folgenden beschrieben Sprachelemente finden sich in [DFG+92], [FGH+93] und [DWH+94].

Datentypen

ProSet stellt Datentypen der endlichen Mengenlehre zur Verfügung. Zu den primitiven Datentypen gehören `integer`, `real`, `boolean` und `string` mit der üblichen Semantik sowie der Typ `atom`, der eindeutig identifizierbare Werte bietet.

Endliche Mengen und Tupel (d.h. endliche Vektoren) der Mathematik sind Vorlage für die zusammengesetzten Datentypen `set` und `tuple`. Sie werden wie in der Mathematik üblich durch Aufzählung oder durch Beschreibung von Eigenschaften definiert.

Datentypen höherer Ordnung sind Funktionen, Module und Instanzen.

Kontrollstrukturen

ProSet verfügt über die aus anderen imperativen Programmiersprachen bekannten Anweisungen wie `loop`, `while`, `repeat`, `for`, `if` und `case`. Diese Anweisungen sind im besonderen auch auf die zusammengesetzten Datentypen anwendbar. Speziell auf zusammengesetzte Typen abgestimmte Konstrukte sind mit der `whilefound`-Anweisung möglich. Auch `for` ermöglicht beispielsweise Iterationen über Tupel.

Prozeduren und Funktionen

Prozeduren in ProSet sind parametrisch polymorph und liefern einen Wert zurück. Die Parameter werden als Kopie (keine Referenzen) durch *call by value*, *call by result* oder *call by value/result* übergeben. Weiterhin ist es möglich, anonyme Prozeduren zu definieren. Aus Prozeduren können durch Anwendung des Operators `closure` Funktionen erzeugt werden, die einen eigenen Datentyp darstellen. Ein `closure` bewirkt das Einfrieren der zum Zeitpunkt der Anwendung gültigen Wertebelegungen nicht-lokaler Objekte, wie z.B. globaler Variablen.

Ausnahmebehandlung

Ausnahmebehandlung in ProSet wird in erster Linie als Mittel zur Strukturierung und Modellierung gesehen. Sie ermöglicht eine knappe und präzise Formulierung eines Algorithmus durch die Trennung der Ausnahmesituationen vom Kernalgorithmus. Diese Trennung ist ein Aspekt des Prototypings, da sie dem Modellierer ebenso wie dem Kunden hilft, die zu entwerfende Anwendung zu verstehen.

Die Behandlung einer Ausnahme läuft wie folgt ab:

1. Wird eine Ausnahmesituation erkannt, wird dieses Ereignis in Form einer Ausnahme dem rufenden Programm gemeldet.

2. Das rufende Programm reagiert darauf mit dem Aufruf einer mit der betreffenden Ausnahme assoziierten Behandlungsroutine (Handler).

3. Der Handler hat die Aufgabe, die Situation zu analysieren, entsprechend zu reagieren und schließlich die Art des weiteren Programmablaufs zu bestimmen. Bei letzterem kann dynamisch zwischen der Rückkehr zum ausnahmeerzeugenden Programm (Wiederaufnahmemodell) und dessen Beendigung (Terminierungsmodell) entschieden werden.

Module und Instanzen

Module sind Schablonen, die die Arbeitsweise von Operationen auf einer gemeinsamen Datenstruktur beschreiben. Bevor die Dienste eines Moduls in Anspruch genommen werden können, muß es instantiiert werden. Dadurch erhält man einen Wert des Typs `instance`. Wie Prozeduren sind Module und ihre Instantiierungen polymorph, d.h. der Typ der importierten und exportierten Werte muß nicht angegeben werden.

Persistenz

Im Rahmen des evolutionären Prototypings werden nicht nur allein Algorithmen entwickelt, sondern es werden auch Daten und Datenstrukturen modelliert. Dies erfordert eine Möglichkeit, Daten persistent zu machen, d.h. einmal berechnete Daten über eine Programmausführung hinweg zu erhalten und sie anderen Programmen zugänglich zu machen. ProSet stellt zu diesem Zweck einen abstrakten Datentyp *P-file* zur Verfügung. Jedes Objekt eines Datentyps erster Klasse (also ein primitiver, zusammengesetzter Datentyp oder ein Datentyp höherer Ordnung) kann persistent gemacht werden, indem es mit dem Schlüsselwort `persistent` unter Angabe des *P-files* und der Zugriffsart (konstant oder variabel) deklariert wird. Auf die gleiche Weise kann auf persistente Werte zugegriffen werden.

Da Funktionen, Module und Instanzen in ProSet Datentypen erster Klasse darstellen und damit persistent gemacht werden können, kann sowohl die getrennte Übersetzung als auch das Binden und Laden von Programmen innerhalb der Sprache formuliert werden.

Programmierung paralleler Anwendungen

Beim Software Prototyping wird ein Modell eines Systems entwickelt. Viele reale Systeme haben eine parallele Struktur. Um nun Modelle solcher paralleler Systeme einfach beschreiben zu können, stellt ProSet Konstrukte zur parallelen Programmierung bereit. So gibt es einfache und mächtige Konstrukte zur dynamischen Erzeugung von Prozessen und zur Koordination von parallelen Prozessen. Die Kommunikation und Synchronistation paralleler Prozesse erfolgt über einen virtuellen gemeinsamen Datenraum, in dem – basierend auf dem Linda-Konzept – einzelne Tupel eingefügt, entfernt, gelesen und geändert werden können. (Siehe dazu auch Abschnitt 2.2.3.)

2.2 Die Koordinationssprache Linda

2.2.1 Linda im Kontext (Klaus Alfert)

Das Konzept von Linda

Bei der Programmierung paralleler Systeme stellen sich Probleme dar, die bei sequentiellen Programmen so nicht auftreten:

- Es muß sowohl Programmfluß (Computation) als auch Kommunikation zwischen parallelen verlaufenden Programmteilen/Threads (Coordination) codiert werden.
- Auf welchem Abstraktionsniveau wird die parallele Hardware verborgen?
- Wie portabel ist der Quelltext?
- Wer entscheidet über den Paralleliesierungsgrad (Compiler versus Programmierer)?
- Lassen sich alte Software–Bibliotheken weiterverwenden, insbesondere wenn neue Programmiersprachen eingesetzt werden müssen?

Der Linda–Ansatz ([Gel85], [GC89], [GC92]) versucht für viele dieser Probleme eine Lösung zu finden. Dabei wird im einzelnen vollständig von Hardware abstrahiert, so daß Portabilität bezgl. der Hardware–Architektur erreicht wird. Desweiteren wird keine neue Sprache definiert, sondern existierende Sprachen um den Koordinationsaspekt erweitert. Damit lassen sich alte Bibliotheken weiterverwenden, weil keine neue Sprache benutzt werden muß. Linda beschränkt sich vollständig auf den Koordinationsapekt, so daß es vom Konzept her sprachunabhängig ist. Die Realisierung erfolgt durch die Einführung einer verteilten, assoziativ addressierbaren Datenstruktur *Tupelraum* sowie Operatoren auf dieser Datenstruktur.

Datenstruktur und Operatoren von Linda

Linda stellt eine Datenstruktur mit dem Namen Tupelraum zur Verfügung. Dieser Tupelraum ist ein Container von Tupeln, die beliebig komplex zusammengesetzt sein können. Er wird von allen Threads geteilt. Die Tupel in dem Tupelraum sind anonym, das heißt, sie gehören keinem der Threads. Der Zugriff auf die Threads erfolgt über einen assoziativen Zugriff über Anzahl, Typen und Werte der Elemente der Tupel. Es werden insbesondere keine Addressen zur Identifikation der Tupel benutzt, dies bewirkt eine starke Abstraktion von der Hardware.

Tupel können als reine Daten durch den Operator `out (t)` oder als aktive Tupel, deren Werte noch berechnet werden durch `eval (t)` in den Tupelraum eingebracht werden. Bei aktiven Tupeln wird die Berechnung der Daten parallel zum aktuellen Thread durchgeführt.

Die so in den Tupelraum eingetragenen Daten können mit vier verschiedenen Operatoren wieder gelesen werden, die sich durch die orthogonalen Eigenschaften Blockieren, bei nicht vorhandenen Daten sowie Entfernung des Tupels aus dem Tupelraum durch das Einlesen, aufgeteilt werden können.

	blockierend	nicht blockierend
entfernt Tupel	`in (t)`	`inp (t)`
beläßt Tupel	`rd (t)`	`rdp (t)`

Kritik des Linda–Ansatzes

Die Eleganz von Linda wird durch die bestechende Einfachheit der Konzepte und Operatoren bewirkt. Dieses ist aber gerade das Problem ([Bal90], [Bal91]), weil die Operatoren von Linda nur eine Art *Tupelraum–Assembler* darstellen. Höhere Kommunikationskonzepte wie etwa Mailboxen, Message Sending oder Remote Procedure Calls sind nicht vorhanden und müssen von Hand implementiert werden. Teilweise ist dies kaum möglich (RPC). Da die Hardware durch Linda vollständig abstrahiert wird, ist die Performance entscheidend von der Realisierung des Linda–Laufzeitsystems und –Compilers abhängig.

Die Vorteile von Linda liegen im Bereich der Portablität bzgl. der Hardware–Architekturen, die Migration sequentieller Software durch schrittweise Implementierung der parallelen Erweiterungen sowie die zur Verfügungstellung eines einzigen Koordinations–Paradigma für eine Vielzahl von Sprachen mit den unterschiedlichsten Berechnungs–Paradigmen (C–Linda, ProSet–Linda, Scheme–Linda, Prolog–Linda, Postscript–Linda, C++Linda). Insgesamt stellt Linda eine elegante Interprozesskommunikation zwischen eng gekoppelten Prozessen dar.

2.2.2 Linda in der Anwendung (Peter Neumann)

Zu diesem Abschnitt siehe auch [RB91] und [MK89].

Linda unterstützt die Programmierung in verteilten oder parallelen Systemen. Dabei ist es egal, ob man sich in einem Netzwerk befindet oder mit gemeinsam benutztem Speicher arbeitet.

Damit das möglich ist, arbeitet Linda mit einer verteilten Datenstruktur, dem Tupelraum. Über diesen Tupelraum kommunizieren die parallelen Prozesse, indem sie dort ihre Daten ablegen oder herausholen.

Linda stellt die dafür in Frage kommenden Operationen zur Verfügung. Diese werden in eine Programmiersprache eingebettet, z.B. C oder Modula-2.

Replicated Worker Style (Master-Worker Modell)

Jeder Prozeß entspricht einem Worker, wobei einer davon als Master zu sehen ist, der die Worker kontrolliert.

Im Tupelraum stehen die Aufgaben, welche zu erledigen sind.
Der Master stößt die anderen Worker an, die Aufgaben im Tupelraum zu erledigen, bis alle Aufgaben abgearbeitet sind.

Linda als High-Level Koordinationssprache

Linda muß als High-Level Sprache eine große Portabilität auf verschiedene Systeme unterstützen und dabei gleichzeitig garantieren, daß auch wirklich parallele Programme parallele Prozesse erzeugen.

Koordinationssprache soll ausdrücken, daß sich die Prozeßerstellung und die Kommunikation zwischen den Prozessen einfach und ausdrucksstark abbilden läßt.

Message Passing

Da sich in der heutigen Zeit effektive und leistungsfähige Netzwerke mehr und mehr durchgesetzt haben ist das Message Passing durch die akkumulierte Kapazität der Prozessoren zu einer der wichtigsten Umgebungen für parallele Programmierung geworden.

Vergleich :
Linda vs Message Passing

Anhand von verschiedenen Beispielen läßt sich erkennen, daß man mit Linda nicht immer eine Verbesserung gegenüber Message Passing erreichen kann.

Bei einer Multiplikation von 240 x 240 Matrizen oder bei der Lösung eines Gleichungssystems mit 400 Unbekannten sind Linda und Message Passing auf dem gleichen Leistungsniveau [Bjo91].

Eine 30-fache Verbesserung erreicht man, wenn Linda beim Ray-Tracing mit einem Standard Master-Worker Modell eingesetzt wird. Ein solches Programm, Rayshade, hat der Student Craig Kolb an der Universität Yale entwickelt [RB91].

Unter Umständen kann es z.B. beim 'All Pairs Shortest Path Problem (ASP)' zu einer Verschlechterung kommen, da Worker im Linda Programm blockiert werden können, wenn andere Worker ihre Aufgabe noch nicht beendet haben.

Das Piranha Modell

Um in einem leistungsfähigen Netzwerk freie Kapazitäten der Prozessoren ausnutzen zu können, verwendet man das Piranha Modell in einem Linda Programm.

Diese Methode bezieht jeden angemeldeten und freien Prozessor (Piranha) mit in den Berechnungsvorgang des Linda Programms mit ein, so daß eine möglichst große Kapazitätsauslastung im Netzwerk erreicht wird.

Sollten die Prozessoren vom Benutzer der jeweiligen Arbeitsstationen benötigt werden, sorgt das Piranha Modell dafür das der Linda Prozeß solange unterbrochen wird, bis der Benutzer dem Prozessor eine Pause gönnt.
Nun wird der unterbrochene Linda Prozeß wieder aufgenommen. Falls er schon beendet worden war, kann nun ein neuer Prozeß aus dem Tupelraum bearbeitet werden.

2.2.3 ProSet-Linda (Jörn Bodemann)

Motivation

Parallele Programmierung wird als eine der Schlüsseltechnologien gehandelt, wenn es darum geht, die Performance von Computerprogrammen signifikant zu steigern. Das liegt zum einen in der Tatsache begründet, daß die Steigerungsraten der Prozessorleistungen einen gewissen Faktor nicht übersteigen und zum anderen an den stetig sinkenden Kosten für modere Prozessoren. Durch den Einsatz paralleler Algorithmen soll eine Performancesteigerung somit schnell und kostengünstig erreichbar sein. Ein Grund dafür, daß sowenige parallele Algorithmen in der Praxis Anwendung finden, resultiert aus der Schwierigkeit, diese Algorithmen zu entwickeln. Die Struktur menschlichen Denkens ist nicht auf Parallelität ausgelegt, was das Nachvollziehen des Ablaufs solcher Programme und Programmteile den Autor vor extrem schwierig zu lösende Probleme stellt.

Ziel der Prototypingprogrammiersprache ProSet-Linda [Has93], ist es, dem Entwickler eine einfache Möglichkeit an die Hand zu geben, parallele Algorithmen zu entwickeln.

Problemlösung

Die beiden zentralen Probleme bei der Entwicklung paralleler Algorithmen sind:

1. Prozeßkommunikation
2. Prozeßsynchronisation

ProSet wurde, um der Zielsetzung gerecht zu werden, durch Linda ergänzt. Dardurch reduzieren sich die beiden oben genannten Probleme auf den Zugriff auf einen gemeinsamen Speicherbreich. Dieses Konzept hat alle Vorteile der 'shared-memory-architecture' als da wären:

1. anonyme Kommunikation
2. einfache Lastverteilung

Dieser gemeinsame Speicherbereich hat unter Linda den Namen 'Tupelraum'. Durch das Konzept der anonymen Kommunikation kann mit ProSet-Linda jeder Prozeß relativ unabhängig von den anderen erstellt werden. Dadurch werden die Unterschiede zum sequentiellen Programmieren sehr gering.

Die Programmiersprache ProSet wird in einem separaten Vortrag vorgestellt (vgl. Kap. 2.1.4 auf Seite 22).

Die Tupeloperationen

Der Tupelraum ist in ProSet-Linda durch drei Operationen manipulierbar.

- deposit
- fetch
- meet

Ein Tuple wird durch eine Anweisung, wie zum Beispiel

```
deposit [ 123, "mystring", 3.14 ] at TS end deposit;
```

in den Tupelraum geschreiben.

Gespeicherte Tupel werden duch das Kommando `fetch` ausgelesen, wie das folgende Beispiel zeigt:

```
fetch ( "name", ? x | (type $(2) = integer) ) at TS end fetch;
```

Diese Anweisung bewirkt, daß ein Tupel gesucht wird, dessen erste Komponente den Wert `"name"` hat, und dessen zweite Komponente vom Typ `integer` ist. Das `"$"`-Zeichen wird als Platzhalter für den entsprechenden Wert des Tupels im Tupelraum verwendet. Der Ausdruck `"$(2)"` selektiert dann das zweite Element des Tupels. Ein mit der `fetch`-Operation ausgelesenes Tupel wird im Tupelraum gelöscht.

Die folgende Operation:

```
meet ( "name", ? into $(2)+1 ) at TS end meet;
```

ist in der Lage, ein Tupel auszulesen, ohne es jedoch aus dem Tupelraum zu löschen. Die Anweisung `"into"` bewirkt hier, daß die zweite Komponente um den Wert `1` vergrößert wird. Die obige Anweisung verändert den Tupelraum also zusätzlich, wobei anzumerken ist, daß diese Operation atomar verläuft.

2.3 Kooperative Planung

2.3.1 Verteilte KI und kooperatives Arbeiten (Markus Brameier)

Einleitung

Die *Verteilte KI (VKI)* hat sich in zwei Hauptrichtungen entwickelt. Das *Verteilte–Probleme–Lösen* betrachtet, wie die Lösung eines konkreten Problems auf mehrere kooperierende wissensteilende Module verteilt werden kann. In *Multi–Agenten–Systemen* hingegen liegt der Schwerpunkt auf der Koordination des Verhaltens von autonomen intelligenten Agenten.

Sozialisation künstlicher Agenten

Was sind die Charakteristiken eines künstlichen Agenten, um ein für die Gemeinschaft optimales Verhalten zu zeigen, das vielleicht nicht ganz optimal für ihn selbst ist? Die Agenten sollten natürlich die Fähigkeit haben zu kooperieren, d.h. sie sollten u.a. mit Mitteln zur Kommunikation und zur Darstellung von Bedürfnissen anderer Agenten ausgestattet sein. Mindestens genauso wichtig wie die Fähigkeit selbst ist die Darstellung eines Willens, diese Fähigkeit zugunsten der Gesellschaft einzusetzen. Jeder Agent sollte stets seine eigenen internen Ziele, nämlich seine Aufgaben zu erfüllen, auf die er spezialisiert ist, den Zielen der Gemeinschaft unterordnen, wenn die globale Aufgabe dies erfordert. Die optimale Sozialisation eines Agenten, d.h. seine Integration in eine Gesellschaft von Agenten, erfordert also geeignete Mittel zur Aufrechterhaltung eines optimalen Kompromisses zwischen Selbstsüchtigkeit bzgl. seiner eigenen Ziele und Selbstlosigkeit gegenüber anderen Agenten bzw. dem gemeinsamen Ziel. Ohne diesen Kompromiß wäre ein Agent in beiderlei Hinsicht für die Gesellschaft nutzlos. Zusammengefaßt ergeben sich 5 Eigenschaften, *Kooperationsaxiome*, die jeder kooperative Agent einer Gesellschaft besitzen sollte [MS91] :

1. Die Mittel seine Umwelt darzustellen, deren Veränderungen aufzunehmen und Schlüsse daraus zu ziehen.

2. Die Fähigkeit zur Kommunikation.

3. Den allgemeinen Willen zur gegenseitigen Unterstützung.

4. Das Wissen über andere Agenten.

5. Das Wissen über seine eigenen Ziele und das globale Ziel der Gemeinschaft.

Ziel ist es, nicht eine zentrale Planungsagentur, sondern eine verteilt planende Gemeinschaft zu schaffen, die ihre Stärke aus der kooperativen Interaktion von autonomen Agenten zieht.

Struktur der Agenten

Die Darstellung künstlicher Agenten ist eng verbunden mit Objekten in objekt-orientierten Sprachen und Modellen. In Ansätzen des Verteilte–Probleme–Lösens führen die Agenten nicht einfach (zentral) vorgegebene Pläne aus, sondern sind auch aktiv an deren Konstruktion beteiligt. Das Verhalten von Objekten unterliegt ausschließlich fest definierten Gesetzen. Alle Aktionen sind an bestimmte Vorbedingungen geknüpft, die wieder Ergebnisse anderer Aktionen sein können. Bis zu einem gewissen Grad verhalten sich Agenten wie Objekte. Zusätzlich, enthalten sie eine Wissensstruktur, die ihr Verhalten bis zum Erreichen eines Ziel bestimmt und sich aus Zielstruktur, angestellten Vermutungen und Bekanntschaften mit anderen Agenten zusammensetzt. Ein Agent ist damit in der Lage, selbständig über die Ausführung einer Aktion zu entscheiden.

Kooperationsstrategien

Wichtige Kooperationsstrategien [HD91] aus dem Gebiet der VKI sind:

1. *Master–Slave:* Ein Agent hat die vollständige Kontrolle über einen anderen und kann ihm befehlen, Aufgaben auszuführen.

2. *Vertragsnetz (contract net):* Ein Lieferantenagent (*contractor agent*) unterbreitet seine Lösung eines Ziels einem oder mehreren Agenten, indem er nach Angeboten für die Ausführung des Ziels fragt. Der beste Anbieter, d.h. der Agent mit den besten Eigenschaften, schießt mit dem Lieferantenagent den Vertrag ab.

3. *Schwarzes Brett (blackboard):* Agenten teilen Ziele und ihre Lösungen, indem sie diese an einem Schwarzen Brett notieren, das von allen jederzeit gelesen werden kann. Während einer Planungsphase ist ein Agent in der Lage ein angeschriebenes Ziel zu analysieren und eine geeignete Strategie vorzuschlagen, d.h. seine Lösung am Schwarzen Brett anzubringen, um sie für andere nutzbar zu machen.

Ist jeder Agent in der Lage, die zum Erreichen eines Ziels geeignetste Strategie auszuwählen, kann sich eine hoch dynamische Struktur entwickeln, in der mehrere unterschiedliche Strategien gleichzeitig umgesetzt werden und die auf keinem vorbestimmten Organisationsschema beruht. Eine *Kooperationswelt (c–world)* ist definiert als die Gesamtheit der Agenten, ihrer Kommunikationsverbindungen und den Kooperationsstrukturen, um das übergeordnete globale Ziel umzusetzen.

2.3.2 Analyse eines Scotland-Yard-Programms (Oliver Alsbach)

Das analysierte Programm stammt von Becker und Zell an der Universität Stuttgart. Als Grundlage für die Analyse diente hauptsächlich der Quelltext des Programms. Weitere Informationen sind der Ausarbeitung von Becker und Zell [BZ90] entnommen. Eine Analyse des Programms zur Laufzeit konnte nicht vorgenommen werden, da es für ein Mehrprozessorsystem implementiert wurde (siehe Hardware-/ Softwareanforderungen).

Programmidee

Das Programm soll das Spiel *Scotland Yard* der Firma Ravensburger realisieren, wobei die Detektive vom Computer gespielt werden. Der Benutzer spielt mit Hilfe einer graphischen Oberfläche die Figur des Mister X. Zur Vereinfachung des Programmalgorithmus hat Mister X keine 2x-Tickets zur Verfügung. Der Computer realisiert durch parallele Prozesse bis zu 10 Detektiven (im Originalspiel sind nur 5 erlaubt), die jeweils zwei Züge voraus berechnen. Im Gesamtverhalten aller Detektive soll der Computer die Spielstärke von menschlichen Spielern haben. Dem Verhalten der Detektive liegt keine zentrale Strategie zugrunde, bei der Teilaufgaben auf die einzelnen Detektiven verteilt werden. Vielmehr sind die einzelnen Detektive autonom und versuchen ihren eigenen selbstentworfenen Plan zu verwirklichen. Die globale Zusammenarbeit kommt durch Abstimmung und Nachrichtenaustausch zustande. Dabei veröffentlichen alle Detektive ihre augenblicklichen Pläne in einem Blackboard, auf das auch alle anderen zugreifen können. Durch wiederholtes Durchführen der Planung und den Austausch von Information kommt es zur kooperativen Planung voneinander unabhängiger Agenten.

Hardware- / Softwareanforderungen

Das Programm ist in der Programmiersprache C implementiert. Es ist entwickelt worden für ein Mehrprozessorsystem vom Typ Sequence Symmetry. Diese Rechnertypen erlauben es, daß mehrere Prozesse zur Laufzeit auf gemeinsamen Speicher zugreifen können. Dadurch ist das Programm auf anderen Rechnern nicht lauffähig. Zur Nutzung des gemeinsamen Speichers wurden folgende C-Spracherweiterungen genutzt:

1. Variablen vom Typ `shared`
2. die Befehle `shmalloc()` und `shfree`, um shared Variablen zu allocieren und freizugeben.
3. die Befehle `s_init_lock()`, `s_lock()`, und `s_unlock()` um für die `shared` Variablen das Semaphorkonzept zu verwirklichen. Dabei darf immer nur ein Prozess auf eine Variable zugreifen.

Kommunikation

Die grundlegende Anforderung für die kooperative Planung ist, daß die einzelnen Detektivprozesse untereinander Informationen austauschen können. Dieser Informationsaustausch funktioniert über ein `Blackboard`, das durch `shared` Variablen realisiert ist. Durch das Semaphorkonzept wird sichergestellt, daß immer nur ein Prozess auf eine Variable zugreifen kann. Im Blackboard stehen alle Informationen, die ein Detektivprozess braucht, um einen Zug zu planen und seine Planung mit den anderen Detektivprozessen abzustimmen. Auch das Hauptprogramm entnimmt aus dem Blackboard die Information, ob ein Detektivprozess eine Zugplanung beendet hat und falls dies geschehen ist, welchen Zug er berechnet hat.

Programmablauf

Nach dem Start des Programms wird zuerst das Spielbrett bzw. das Wegenetz aus einer Datei geladen. Dadurch ist es möglich, mit unterschiedlichen Spielplänen zu spielen. Danach wird die graphische Oberfläche gestartet und der Benutzer kann die Startpositionen der Detektive bestimmen. Dann werden die Prozesse für die Detektive abgespalten. Der Hauptprozeß wartet auf Benutzereingaben und schreibt die Zugdaten von Mister X in das Blackboard, damit die Detektive wissen, wann ein Spielzug abgeschlossen ist. Durch das Blackboard erfährt der Hauptprozess dann, ob die Detektivprozesse ihre Planung beendet haben und welchen Zug sie berechnet haben.

Detektivprozesse

Die Detektivprozesse planen ihren nächsten Zug dadurch, daß allen möglichen Zügen ein Wert zugewiesen wird. Dieser Wert richtet sich danach, wie groß der Abstand des Detektives vom Zielort zu allen möglichen Orten von Mister X entfernt ist. Als Abstand wird hierbei immer die minimale Anzahl von Zügen betrachtet, die ein Detektiv mit seinem Ticketrest benötigt, um zu dem entsprechendem Zielort zu gelangen.

Die Detektivprozesse verfahren nach folgendem Algorithmus:

1. Lesen der aktuellen Daten aus dem Blackboard.
2. (P) Alle möglichen Aufenthaltsorte von Mister X berechnen.
3. (P) Aufenthaltswahrscheinlichkeit dieser Orte berechnen, welche sich nach den alten Wahrscheinlichkeiten und den Abständen der Detektive zu diesen Orten richtet.
4. (P) Alle möglichen eigenen Zielorte berechnen.
5. (P) Zielorte bewerten: Summe der Abstände zu allen möglichen Mister X Orten.
6. Alle (P) - Schritte mit den Zielorten aus den ersten vier Schritten, um den Folgezug zu berechnen.
7. Bestes Folgeziel auswählen.
8. Bestes Ziel auswählen.
9. Konfliktlösung: Dazu werden die eigenen Pläne mit denen der anderen Detektive verglichen. Bei gleichen Zielen muss ein Detektiv ausweichen. Dazu werden die Werte der Pläne verglichen. Durch den gemeinsamen Speicher kann ein Detektiv den Plan eines anderen aus dem Blackboard löschen. Dies darf er allerdings nur, wenn dieser seine Planung noch nicht abgeschlossen hat.

Strategie

Die grundlegende Strategie ist, daß die Wahrscheinlichkeit der Orte, an denen Mister X sich aufhalten kann, anhand der Abstände zu den möglichen Detektivorten ausgerechnet wird. Im Planungsalgorithmus treten aber Wertminderungen auf, die das Programm anfällig macht für das, was der Mensch oft genug bei solch einem Spiel macht, das Bluffen. Das Programm betrachtet es als nahezu unwahrscheinlich, daß Mister X ein `Black-Ticket` an einem Ort benutzt, von dem aus er nur mit einem Verkehrmittel weiterfahren kann, bzw. mit einem `Black-Ticket` einen solchen Ort aufsucht. Genauso unwahrscheinlich wird die Möglichkeit angesehen, daß Mister X einen Ort wählt, den eine Detektiv im nächsten Zug erreichen kann. Die Berechnung der Aufenthaltswahrscheinlichkeit von Mister X wird nur in Abhängigkeit von den Abständen zu den Detektiven berechnet. Es wird nicht berücksichtigt, daß manche Orte für Mister X wertvoller sind als andere, zum Beispiel Orte mit U-Bahn-Stationen (größere Auswahl für den nächsten Zug) oder Orte wie 157.

2.3.3 Analyse der graphischen Oberfläche eines Scotland-Yard-Programms (Horst Sdun)

Dieser Text gibt eine kurze Zusammenfassung des Seminarvortrags über die *Analyse der graphischen Oberfläche eines Scotland Yard Programms*, weiterhin sollten die Möglichkeiten einer Wiederverwendung dieser Quellen für das Projekt Scotland Yard diskutiert und der dazu nötige Arbeitsaufwand abgeschätzt werden. Die Quellen stammen aus der Arbeit "Kooperatives Planen unabhängiger Agenten" [BZ90] von Wolfgang Becker und Andreas Zell und sind an der Universität Stuttgart am Institut für parallele und verteilte Höchstleistungsrechner erstellt worden. Weiterhin sollen auch mögliche Erweiterungswünsche wie z.B. Farbdarstellung nicht außer Acht gelassen werden.

Das Programm Das Programm ist in einem Superset von K&R-C implementiert. Es ist schlecht und teilweise gar nicht oder nur unverständlich dokumentiert. Eine Gliederung innerhalb der Funktionen ist nicht erkennbar. Der Quelltext wirkt extrem gedrängt, kompakt und unübersichtlich. Die Funktionen sind nach ihrer Funktionalität wie z.B. Callbacks, Prozesse oder Widgets in verschiedene Dateien aufgeteilt und nicht nach Zugehörigkeit zu bestimmten Programmkomponenten.

- widgets.c
  Mister-X, Spielplan, Strukturplan,Detektiv
  Widgets erzeugen

- callbacks.c
  Mister-X, Spielplan, Strukturplan, Detektiv
  Callbacks

- grafik.c
  Spielplan, Strukturplan Grafikfunktionen
  wie Linien,Kreise,Ellipsen...

- prozesse.c
  Spiel beenden, Detektiv Spielzug, Mister-X bedienen,

- wege.c
  Ortsliste, Abstandsliste, Abstände berechnen

- blackboard.c
  Shared Variablen, Blackboard Operationen

- init.c
  Blackboard initialisieren , Mister-X Startorte ermitteln,
  Zugdaten lesen/schreiben

- main.c
  Prozesse erzeugen, Grafik initialisieren

- strategie.c
  Zielorte berechenen Wahrscheinlickeiten berechenen

- hilf.c
  Alles was übrig ist, z.B. Abstandslisten berechnen,
  Zufallsorte bestimmen, Strings ausgeben...

Eine Ausnahme bildet die Blackboard Komponente des Programms.

Die Graphische Benutzerschnittstelle Das Programm setzt laut Autor auf X11R4 auf, benutzt allerdings HP-Widgets (Xw) für X11R3. Bei den HP-Widgets handelt es sich um ein X Toolkit aus dem Buch [You]. Ursprünglich wurden die Sourcen von der Hewlett-Packard Company als "unsupported, user-contributed code." verbreitet. Die Sourcen sind via ftp frei verfügbar und unterliegen dem Xaw-Copyright. Die HP Widgets werden allerdings nicht für das Spielplan und das Struktur Fenster benutzt. Diese Fenster sind direkt über das X Toolkit Programmiert, da die HP Widgets keine Bit/-Pixmaps unterstützen. Dies ist wohl auch der Grund für den fehlenden Redraw des Strukturfensters. Diese direkte X Toolkit Programmierung wirkt stilistisch eher wie ein verzweifelter Versuch einen Spielplan brachial ohne X11 Kenntnisse auf den Bildschirm zu bringen. Die Xw's sind für X11R5 bereits verfügbar und natürlich zu der alten Version kompatibel. Eventuell kann bei der neuen Version sogar Xaw3d anstelle Xaw verwendet werden.

Das Spielplan Fenster Eine kurze Beschreibung der Elemente der Benutzeroberfläche befindet sich in der Arbeit [BZ90]. Die vorliegende Benutzeroberfläche, insbesondere der Spielplan ist in monochrom. Hinzu kommt eine, wahrscheinlich aufgrund nachträglichem Ditherns, sehr schlechte Bildqualität. Einige Teile des Spielplans sind nur sehr schwer erkennbar und der Spielplan wirkt daher unübersichtlich. Ferner ist der Spielplan nicht in seiner Darstellungsgröße nicht skalierbar. Dies ist unpraktisch, da eine Bildgröße von 800x604 Pixeln auch bei einer Auflösung von 1280x1024 recht viel Platz in Anspuruch nimmt und gerade noch Platz für einige Detektiv Fenster und das Mister X Fenster läßt. Für Fenster eines Debuggers o.Ä. bleibt kein ausreichender Platz mehr. Die Bitmapdatei hat kein Standardformat (Xloadimage(1) erkennt es zumindest nicht). Eine nachträgliche Bearbeitung mit gängigen Tools wie XV, ImageMagic, oder den PBMTools war nicht möglich. Das Programm bietet auch eine Strukturanzeige, diese ist zwar skalierbar, bietet aber keinen automatischen Redraw. Wird das Fenster überdeckt, so muß der Button "Frisches Bild" angeklickt werden, um die Anzeige zu restaurieren. Werden die Quellen für einen automatischen Redraw geändert, so werden im Betrieb *vollständige* Fenster Redraws für jeden Teilbereich angestoßen. In der Praxis kann man mit einer solchen "Redraw Flut" kaum noch arbeiten.

Die Architektur Das GUI wird durch die Funktion *Grafik_Prozess()* initiert (Source: prozesse.c).

Die Funktion erzeugt zunächst ein Widget für das Mister X Fenster und ein Widget für das Spielplan Fenster. Dann wird das Programm bis zu seinem Ende nur noch durch X Events gesteuert. Der XToolkit bildet die Events auf die entsprechenden Callbacks ab (vgl. Abbildung 2.1). Aus diesen Callbacks heraus werden dann je nach Spielzustand (Aufbau- oder Spielphase) die entsprechenden Aktionen des Programms gesteuert, bzw der innere Zustand des Programms geändert. Das was unter X der Normalfall ist um eine "strenge" Menueführung zu vermeiden, bedeutet aber auch eine Steuerung des Programmablaufs durch die Benutzeroberfläche. Dies hat zur Folge, daß die eigendliche Funktionalität des Programms in die Benutzeroberfläche eingebettet wird, oder sogar Bestandteil dieser ist. Das Programm verwendet zwar im Gegensatz dazu für seine "Funktionalität" separate Prozesse. Aber letztendlich werden diese Prozesse auch via IPC aus der Benutzeroberfläche heraus gesteuert. Betrachtet man also die IPC als transparent, so macht dies keinen wesentlichen Unterschied aus. Letzteres ist der Fall, es liegt also keine Einbettung in die Benutzeroberfläche mittels einer Kapselung der Funktionalität vor, sondern die direkte Verwendung der IPC innerhalb der Widgets respektive der Callbacks. Wobei an dieser Stelle schon einmal erwähnt sei, daß die IPC selbst wiederum auch nicht gekapselt ist (Siehe dazu auch Abschnitt 2.3.3). Die Tatsache, daß die "Funktionalität" des Programms in die Benutzeroberfläche eingebettet ist, macht es natürlich unmöglich an dieser Stelle eine Funktionsbibliothek oder zumindest eine Schnittstelle zu einer solchen zu präsentieren.

Die Interprozesskommunikation Da die graphische Benutzeroberfläche des Programms mit der verwendeten IPC eng verwoben ist, soll in diesem Abschnitt ein rudimentärer Überblick über die im Programm verwendete IPC gegeben werden. Der Autor des Pro-

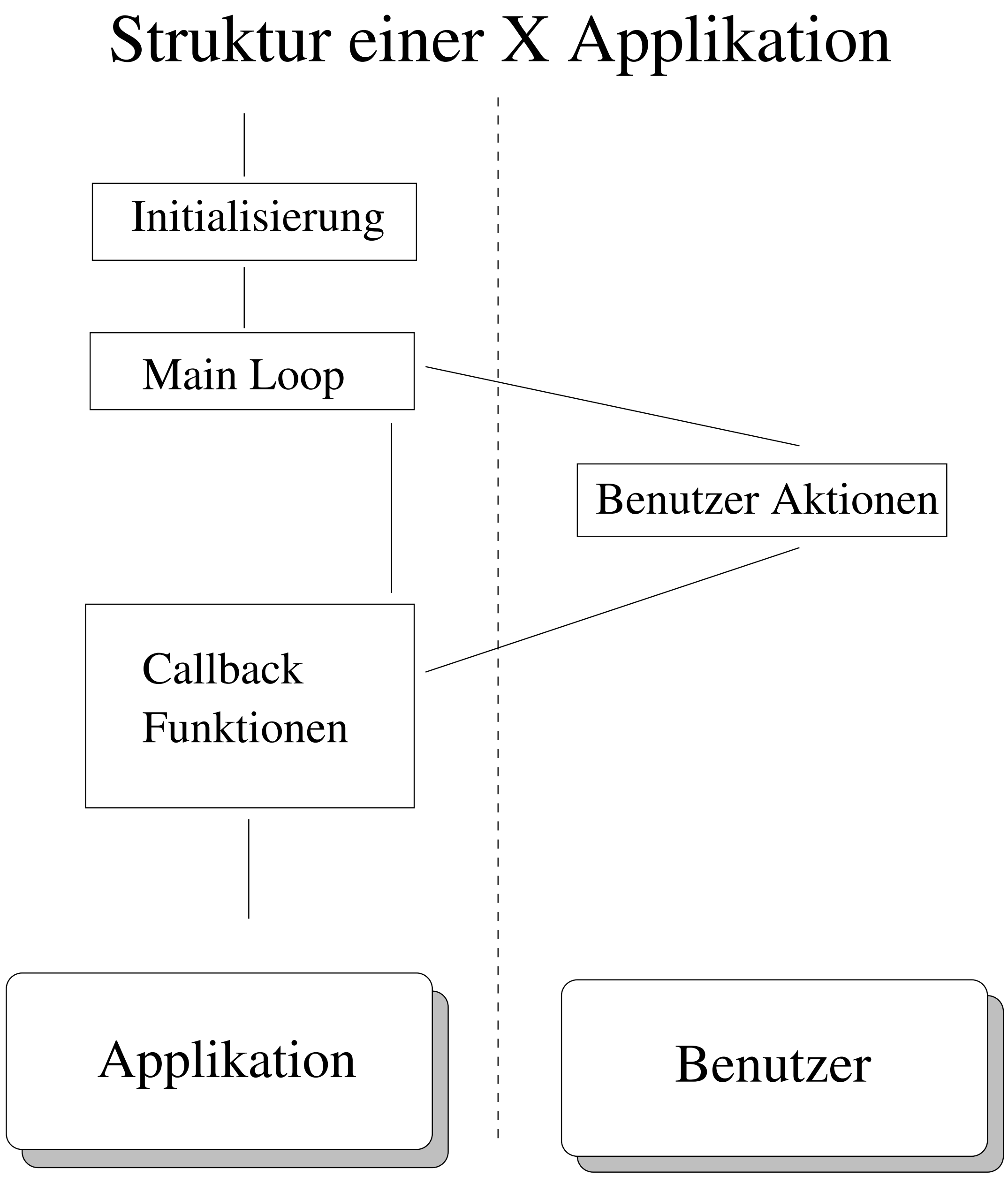


Abbildung 2.1: Die X GUI Struktur

gramms schreibt in dem von ihm verfasstem Text [BZ90] zum Thema Kommunikationsstrukturen:

"Für die Anwendung wurde eine sehr einfache Kommunikationsstruktur gewählt: ... Als Übertragungsmedium dient ein gemeinsamer Informationspool (Blackboard), der keinerlei Steuerungs- oder Synchronisationsaufgaben übernimmt..."

In der Architektur des Programms wurde allerdings genau dies nicht realisiert! Vielmehr sind aus der Sicht der Funktionen, die der Benutzeroberfläche zuzuordnen sind, drei Teile der IPC sichtbar.

- Dynamische Strukturen im Shared Memory
- Globale Variablen der Speicherklasse "shared"
- Blackboard

Es wurde also keine klare IPC Schnittstelle definiert, die nur das Blackboard verwendet.

Das Blackboard Die Kommunikation wird im vorliegendem Source über ein sog. Blackboard abgewickelt. Hierbei wird ein von allen Prozessen gemeinsamen genutzter Datenraum definiert (Abbildung 2.2,2.3,2.4), das Blackboard. Jeder Prozess kann Nachrichten von dem Blackboard lesen oder Nachrichten hinterlassen. Das hier verwendete Blackboard übernimmt ferner weder Steuerungs- noch Synchronisationsaufgaben, obwohl in der Blackboard Struktur Semaphore verankert sind. Es wird lediglich vom Blackboard vermerkt welche Informationen für welchen Agenten neu sind ([BZ90, Seite 10ff]). Die Kommunikation wird von allen Programmstellen direkt über das Blackboard unter Zuhilfenahme von Shared Memory abgewickelt, d.h. es ist keine Kapselung der IPC vorhanden. Eine Kapselung des Blackboards hätte ein einfaches Austauschen der IPC ermöglicht.

Das Shared memory Neben dem Blackboard wird Shared Memory verwendet um dynamische Listen zwischen den Prozessen auszutauschen. Es werden Abstandlisten (Source: hilf.c) und "Textstringlisten" (Source: grafik.c) ausgetauscht. Bei den Abstandlisten handelt es sich um Listen mit Ortsdistanzen. Der genaue Zweck der "Textstringlisten" ist mir bis dato noch nicht klar geworden.

Die Speicherklasse *shared* Es wurde eine Speicherklasse "shared" verwendet um die globalen Variablen des Programms auch "Systemglobal" verwenden zu können. Die Adressen dieser Variablen werden dann zur Laufzeit via Blackboard Operationen im Blackboard eingetragen (vgl. Abbildung 2.5). Diese Speicherklasse entspricht keinen C-Standard. Sie entstammt dem Compiler eines Sequent Systems, auf dem diese Scotland Yard Implementation ursprünglich endstanden ist. Mir ist kein anderer C Compiler bekannt, der eine solche Speicherklasse unterstützt. Die Realisierung einer solchen

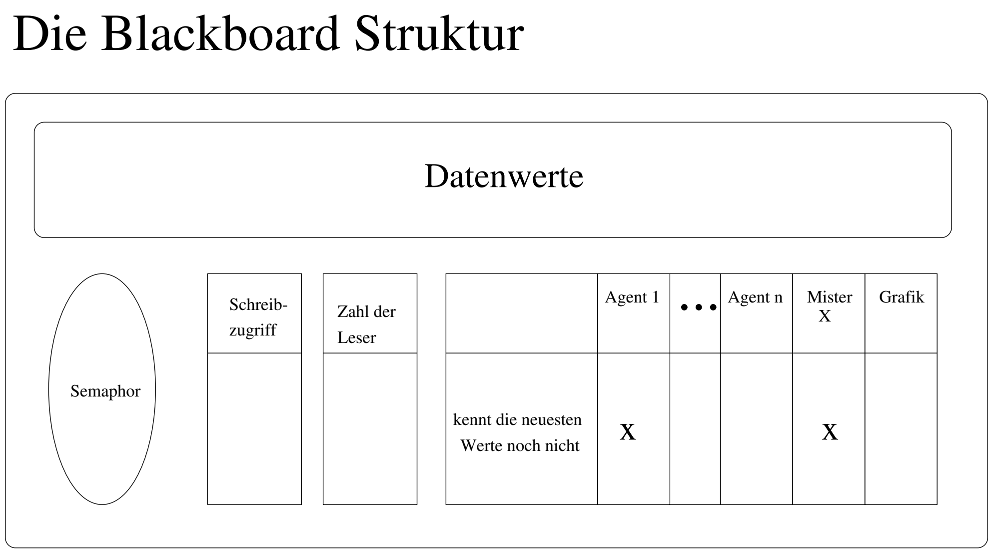


Abbildung 2.2: Die Blackboard Struktur des Autors

```
/*
 * Datentyp fuer das Attributpaket, welches fuer jeden Datensatz im
 * Blackboard notwendig ist. Entspricht einem Filepointer (FILE).
 * Der Mister-X-Prozess und der Grafik-Prozess greifen mit den
 * naechsthoeheren Nummern auf die Daten zu, daher MAX_DETEKTIVZAHL+3.
 */

typedef struct
{
  slock_t lock;
  Boolean Write;
  Boolean Change [MAX_DETEKTIVZAHL+3];
  int     Readers;
}
BB_Attribut;
```

Abbildung 2.3: Implementierte Blackboard Attribute

```
/*Das Blackboard*/
/*Informationen von den Detektiven*/
shared int_array              BB_SpielzugNr,
                              BB_Ringzaehler,
                              BB_Startorte;
shared BB_Attribut_array      BA_SpielzugNr,
                              BA_Ringzaehler,
                              BA_Startorte;
shared Boolean_array          BB_ist_gezogen;
shared BB_Attribut_array      BA_ist_gezogen;
shared Planliste2_array       BB_Plaene;
shared BB_Attribut_array      BA_Plaene;
shared int_array              BB_Geduld;
shared BB_Attribut_array      BA_Geduld;
shared X_Ortliste2_array      BB_X_Zielorte1;
shared BB_Attribut_array      BA_X_Zielorte1;
shared Abstandliste_array     BB_Abstaende;
shared BB_Attribut_array      BA_Abstaende;
/*Informationen von Mister-X*/
shared int                    BB_X_SpielzugNr;
shared BB_Attribut            BA_X_SpielzugNr;
shared int                    BB_X_Mittel;
shared BB_Attribut            BA_X_Mittel;
shared int                    BB_X_Auftauch;
shared BB_Attribut            BA_X_Auftauch;
shared int                    BB_Spielzustand; /*damit die Callbacks wissen,*/
shared BB_Attribut            BA_Spielzustand; /*was der Benutzer gerade will*/
shared int                    BB_Trace_Modus;
shared BB_Attribut            BA_Trace_Modus;
/* Informationen von Allen*/
shared BB_Attribut            BA_Textspeicher;  /*Speicher fuer*/
shared Textringliste          BB_Textspeicher;  /*auszudruckende Texte*/
```

Abbildung 2.4: Das Implementierte Blackboard

```
int              ich;             /*eigene DetektivNr*/
shared int       Detektivzahl;    /*Gesamtzahl der Detektive*/
shared Boolean   Startphase= True; /*bis die Detektive erzeugt sind*/
Boolean          Spielende= False;
Boolean          vielleicht_Spielende= False;
shared int       X_Prozess_Nr;    /*Prozessnummer des Mister-X Prozesses*/
shared int_array Prozess_Nr;      /*Prozessnummern der Detektive*/
Grafiktext       Text;
```

Abbildung 2.5: Einige Shared Deklarationen

Speicherklasse ist in C jedoch nachträglich nicht möglich. Die als shared definierten Variablen sind leider auch nicht gekapselt, so das eine transparente IPC Einbindung nur möglich ist, wenn alle Variablen im Source nachräglich gekapselt werden.

Weiterverwendung Bei einer Weiterverwendung der Sourcen der graphischen Benutzeroberfläche müssen alle die Oberfläche betreffenden Sourcen bezüglich der folgenden Punkte angepasst werden.

Der C Source Da das gesamte System in K&R C implementiert ist ist es sicherlich ratsam die Sourcen für eine Weiterverwendung in ANSI-C zu überführen. K&R C Prototypen, wenn man sie überhaupt so nennen darf, sind unter heutigen Gesichtspunkten nicht mehr diskutabel.

Ein Farbiger Spielplan Ein simples “Neuscannen” des Spielplans reicht keinesfalls aus um einen farbigen Spielplan in die Benutzeroberfläche einzubringen. X ermöglicht zwar ein sehr einfaches Handling von 1-Plane Bitmaps, die Handhabung von “bunten” Bildern ist allerdings aufgrund der verschiedenen Farbmodelle (Abbildung2.6) von X etwas komplexer (vgl. [Con, Kapitel 7]). Die X11R3 Xw unterstützt allerdings auch keine Images oder Pixmaps. X11R5 Xw unterstützt zumindest Images. Sollten also in Zukunft auch farbige Spielpläne oder zumindest Spielpläne mit Graustufen möglich sein, so wird man ein Reprogramieren der grafischen Benutzeroberfläche nicht umgehen können. Eine Anpassung dürfte sich als schwierig erweisen, da die vorhandene Version der HP-Widgets weder Bitmaps noch Pixmaps unterstützt. Das Programm selbst greift hier auf den XToolkit zurück. Die HP Widgets können also nicht konsequent genutzt werden.

Generelles zur Weiterverwendung der Oberfläche Da, wie bereits in 2.3.3 beschrieben, die IPC in die Oberfläche integriert ist, kann keine Funktionsbibliothek für

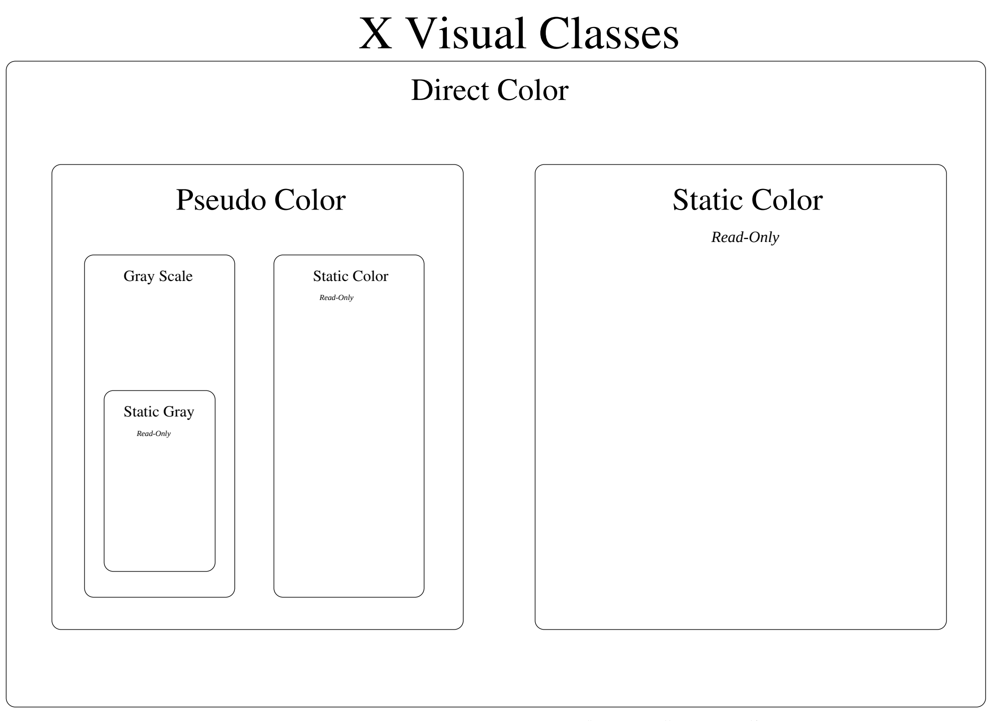


Abbildung 2.6: Die X Farbklassen

die Wiederverwertung der Graphischen Oberfläche direkt aus den Sourcen extrahiert werden. *Vielmehr müssen dazu die IPC Mechanismen (Abschnitt 2.3.3) und sonstige Rückstände komplett aus den Callbacks entfernt und durch eine saubere Schnittstelle ersetzt werden. Dazu reicht eine Einarbeitung in den Benutzeroberflächenteil keinesfalls aus, sondern vielmehr wird man sich in das gesammte Programm inclusive der verwendeten IPC einarbeiten müssen.* Die Tatsache, daß die Blackboardstruktur nicht konsequent eingehalten wurde, verkompliziert die Situation noch erheblich, da eine genaue Analyse der "Systemglobalen" shared Variablen gemacht werden muß.

Die Kommunikation Da, wie in 2.3.3 beschrieben, die Pointer von *shared* Variablen über das Blackboard an andere Prozesse weitergegeben werden, ist ein kompletter Ersatz der IPC unumgänglich. Selbst wenn die *shared* Variablen gekapselt und auf Shared Memory abgebildet werden, so kann keinesfalls die Adresse in das Blackboard eingetragen werden. Für einen anderen Prozess kann der Shared Memory Bereich in einem anderem Adressbereich liegen. Shared Memory Adressen werden im Normalfall über systemeindeutige Schlüssel durch die jeweiligen Prozesse ermittelt.

Überblick Für den Fall einer Wiederverwendung sollten zunächst die folgenden Maßnahmen ergriffen werden:

- Überführung in ANSI - C
- Neuaufteilung der Funktionen
- Einarbeitung in die IPC
- Einarbeitung in die GUI
- Kommentierung der Sourcen
- Extrahieren der GUI Bestandteile
- Einbettung einer neuen IPC in die GUI

Interface Builder Für eine eventuelle Neukonstruktion wären Toolkits wie etwa:

- Xlib
- XToolkit
- HP Widgets
- Athena Widgets
- OSF Motif

- ISA Dialog Manager
- TCL/TK

zu diskutieren.

Resümee Das Programm wirkt letzendlich sehr monolitisch, eine Weiterverwendung der Quellen kann ich nicht empfehlen. Die *ungeschickte Aufteilung der Funktionen in die verschiedenen Dateien* zerstören jegliche Übersicht über die Quellen. Die *Umgehung des Blackboards* durch Verwendung von *Systemglobalen "shared" Variablen* lassen das Programm so undurchsichtig wie eine Teller Spaghetti erscheinen. Die *kompakte Schreibweise* und letztendlich die *dünne Kommentierung* machen die Analyse der Quellen zu einer Gedultsprobe.

2.4 Software-Entwicklungsumgebungen und formale Spezifikation

2.4.1 Software-Entwicklungsumgebungen (Stefan Hedtfeld)

Als Grundlage für diesen Vortrag dienten die beiden Artikel von M. Nagl [Nag93] und U. Kelter [Kel93].

Begriffsabgrenzung

In der Literatur wird der Begriff *Software-Entwicklungsumgebung* nicht einheitlich verwendet. Meist wird er in folgenden Zusammenhängen verwendet:

- Ansammlung vorgegebener Programmbausteine;
- Plattform für Software-Entwicklungsumgebung, die auch für andere verteilte Systeme verwendet werden kann;
- abgestimmte Softwaretechnik-Arbeitsumgebung zur Erstellung beliebiger Softwaresysteme, auf eine oder mehrere Programmiersprachen abgestimmt;
- Auswahl von Werkzeugen und Bausteinen für einen Anwendungsbereich, die dort vorgefundene Sprachen und Methoden unterstützten;
- eine auf einen Anwendungsbereich abgestimmte, abgeschlossene Modellierungs- oder Arbeitsumgebung, in der der Anwender direkt seine Gedankenwelt vorfindet und nicht mehr im üblichen Sinne programmiert;
- eine Meta-Umgebung zum Bau von Software-Entwicklungsumgebungen.

In diesem Zusammenhang ist von Bedeutung, daß das Produkt *Software* aus Entwickler- und Managersicht nicht allein das abgelieferte, lauffähige System ist, sondern es besteht zudem aus unterschiedlichen Dokumenten:

Anforderungsdefinition, Architektur, Modulimplementation, Aufgabennetz, ... Diese Dokumente können aus Teildokumenten bestehen, die wiederum aus vielen Bestandteilen (Inkrementen) zusammengesetzt sind, die auf die Syntax der jeweiligen Sprache abgestimmt sind. Zwischen all diesen Dokumenten bestehen Querbezüge. Diese komplexen Konfigurationen müssen bei der Softwareerstellung, -wartung oder -wiederverwendung verwaltet werden.

Technische und administrative Ebene

Auf der *technischen* Ebene wird beschrieben, „wie" Dokumente und Prozesse zu strukturieren und durchzuführen sind; auf der *administrativen* Ebene wird beschrieben, „welche" Konfigurationen, Prozesse vorhanden und in welchem Zustand sind.

Integrationsrahmen

Am liebsten hätten es Anwender von SEU, daß sie benötigte Werkzeuge in einen „Software-Bus" stecken können, eben genau so, wie man die Hardware eines Rechner konfigurieren kann, indem man eine Karte in einen Bus steckt. Einen solchen Software-Bus nennt man auch Integrationsrahmen. Bisher ist eine solche SEU noch nicht gebaut.

Die Probleme zu beschreiben, die die Konstruktion von Integrationsrahmen erschweren, ist Inhalt des nun folgenden Teils.

Es hat schon Versuche gegeben, Integrationsrahmen zu definieren und auf dem Markt einzuführen. Dabei war der Funktionsumfang dieser Systeme oft sehr groß, oftmals auch schwer vergleichbar. Ebenso ist es nicht leicht, Funktionalitäten von mehreren Integrationsrahmen zu vereinigen.

Um Vergleiche zu vereinfachen haben die ECMA (European Computer Manufacturer Association) und der NIST gemeinsam ein Referenzmodell entwickelt, das *Reference Model for Frameworks of Software Engineering Environments (Version 3)*. Es ist als Klassifikationsraster gedacht, in dem die Leistungen und Dienste eines Integrationsrahmens beschrieben werden können, ohne konkret eine SEU-Architektur zu implizieren.

Als Funktionsbereiche haben sich herausgebildet:

1. Objektverwaltung
2. Steuerung des Software-Entwicklungsprozesses
3. Kommunikation zwischen Werkzeugen
4. Betriebssystemdienste
5. Konstruktion von Benutzungsschnittstellen
6. Zugriffskontrollen

7. Administration des Integrationsrahmens

Bestandteile eines Integrationsrahmens Die Architektur einer SEU ist grob wie folgt:

Kommandoverarbeitung Kontrolle des Entwicklungsprozesses
einzelne Werkzeuge
dedizierte Basisdienste für SEU

Die Kommandoverarbeitung regelt den Zugang zu dem System und erlaubt es den Entwicklern Werkzeuge aufzurufen. Diese Werkzeuge benutzen ihrerseits Dienste der Basissysteme. Ein Integrationsrahmen besteht üblicherweise aus folgenden Teilen.

1. Einem oder mehreren Basissystemen. Die Dienste der Basissystem können von den Werkzeugen über eine Programmschnittstelle benutzt werden. Die meisten Basissysteme haben außerdem eigene Administrationswerkzeuge, z.B. für Backup oder Benutzerverwaltung; diese sollten ebenfalls in die Umgebung integriert sein, auch wenn sie nur vom Administrator benutzt werden.

2. Manchmal einem speziellen Kommandointerpreter oder Werkzeugen zur Steuerung des Software-Entwicklungsprozesses.

3. Werkzeugen zum Verwalten und Konfigurieren der SEU.

4. Allgemein einsetzbaren, methodenunabhängigen Standardwerkzeugen wie Texteditoren oder elektronischer Post.

Integration der Basissysteme Leider ist es nicht möglich, mehrere unabhängige Basissysteme zu integrieren, die einen oder mehrere der oben aufgeführten Funktionsbereich abdecken.

- Die Objektverwaltung und die Zugriffskontrolle können offensichtlich nur zusammen realisiert werden.

- Transaktionen können nicht einem einzigen Bereich zugewiesen werden: wenn Werkzeuge in Prozeßhierarchien ausgeführt werden können oder auf Dateien arbeiten, müssen Concurrency Control und Recovery auch in Prozeßhierarchien und auf Dateien wirksam sein.

- Versionsverwaltung und lange Transaktionen betreffen sowohl die Objektverwaltung als auch die Steuerung des Entwicklungsprozesses.

Funktionsbereiche von Integrationsrahmen Die oben vorgestellten Funktionsbereiche werden nun beschrieben. Dabei werden nur diejenigen Funktionsbereiche beschrieben, für die speziell auf SEU zugeschnittene Basissysteme existieren.

Objektverwaltung Herkömmliche DBMS eignen sich nicht für SEU, da ihre Datenmodelle nicht ausreichen und die Transaktions-, Zugriffsschutz-, Verteilungs- und Administrationskonzepte den Ansprüchen nicht genügen. Daher ist ein sehr wichtiger Teil eines Integrationsrahmens ein nichtkonventionelles DBMS.

Datenmodelle Die wichtigsten formalen Standards eines Datenmodells für SEU sind IRDS (Information Resource Dictionary System) und PCTE (Portable Common Tool Environment). Anforderungen an Datenmodelle ergeben sich aus den Problemen:

- bei relationalen DBMS treten häufig Performance-Probleme auf, dazu später mehr;
- in großen, verteilten SEU-Installationen müssen Objekte gleichen Typs unter Umständen auf verschiedenen, nicht miteinander verbundenen Rechnern existieren können. Das bedeutet, daß i.a. nur ein Teil aller Instanzen eines Objekttyps lokal erreichbar ist. Ähnlich liegt der Fall, wenn Objekte extern gelagert werden. Das relationale Modell erlaubt aber partiell zugreifbare Relationen nicht.

Granularitätsproblem Von großer Bedeutung ist das Problem der Performance des DBMS. Am günstigsten wäre es, wenn man Daten möglichst feingranular in einer Datenbank ablegen würde. Doch sind konventionelle DBMS nicht schnell genug, um online-Zugriffe mit erträglichem Geschwindigkeitsverlust zu ermöglichen. Auch der Ausweg, Dateien komplett einzulesen und dann in einer internen Datenstruktur zu bearbeiten, hilft nicht viel weiter, denn

1. kommen auch hier wieder die zu groben Einheiten des Sperrens, Recovers und der Zugriffskontrolle zum Tragen, aber
2. kommt hinzu, daß wenig Unterstützung beim Datenaustausch paralleler Prozesse geboten wird.

Data Dictionaries Um Entwicklungsdaten verwalten zu können, muß ein spezielles DBMS entwickelt werden. Um aber nicht extra spezielle DBMS entwickeln zu müssen (Lizenzkosten), gibt es sogenannte Data Dictionary-Systeme (DDS), die versuchen, ein nichtkonventionelles Datenmodell auf Basis eines konventionellen DBMS zu realisieren.

Das ganze ist natürlich nur dann sinnvoll, wenn ein DDS wesentlich einfacher zu implementieren ist als ein DBMS, was wiederum bedeutet, daß ein DDS nicht grundsätzlich verschieden von allen (!) konventionellen Datenmodellen sein darf.

Standardschemata Um die Datenintegrität von Werkzeugen zu realisieren, müßte man sich auf gemeinsame Schemata einigen. In offenen Systemen müßten die Schemata praktisch standarisiert sein. Da diese Schemata jedoch Bestandteil des Entwurfs von Werkzeugen sind, werden hier völlig verschiedene Methoden und Entwurfskriterien eingesetzt. Da außerdem die feinkörnige Datenmodellierung nur bei sehr leistungsfähigen DBMS möglich ist, konnte sich bis jetzt kein Schema durchsetzten.

Kommunikation zwischen Werkzeugen Kommunikation zwischen Werkzeugen ist dann notwendig, wenn ihre Prozesse gleichzeitig ablaufen. Dies ist bei SEU oft der Fall, zum Beispiel wenn in mehreren Fenstern auf dem Bildschirm parallel mehrere Dokumente editiert werden.

Steuerung des Software-Entwicklungsprozesses Um den Anwender einer SEU im Ablauf des Software-Entwicklungsprozesses zu unterstützen, kann ein Prozeßmodell oder Prozeßprogramm realisiert werden.

Dieses Program sollte dem Benutzer bei interaktiven Arbeitseinheiten die nächstliegenden Arbeitseinheiten anzeigen und gegebenenfalls die Auswahl mehr oder weniger einengen und bei automatisch durchführbaren Arbeitseinheiten die Durchführung ohne Zutun des Anwenders anstoßen und kontrollieren, sobald die Voraussetzungen vorliegen.

Die Benachrichtigung dieses sogenannten *Laufzeitsystems* kann wie oben bei der Kommunikation zwischen Werkzeugen beschrieben ablaufen.

Schlußbetrachtung

Nagel beschreibt als zukünfitge Entwicklungslinien

integrierte Administration: wie anfangs beschrieben, sind administrative Aspekte nicht softwarespezifisch sondern allgemein für beliebige technische Projekte über komplexen, strukturierten Sachverhalten entwickelt und können dann jeder SEU hinzugefügt werden. Dies setzt allerdings standardisierte Schemata zugrundeliegender Modelle voraus. Dabei ist zu beachten, daß einerseits die administrativen Teilbereiche zu einer integrierten administrativen Umgebung zusammengefügt werden müssen, andererseits muß es eine Integration zwischen der administrativen und der technischen Seite geben.

vollständige Konfigurierbarkeit: alle zur Zeit existierenden SEU decken nur einen Teil der möglichen Unterstützung ab. Konfigurierbarkeit in a-posteriori-Ansätzen setzt voraus, daß die einzelnen getrennt entwickelten Werkzeuge zusammenpassen. Sollen sie nach dem a-priori-Ansatz neu entwickelt werden, muß der Erzeugungsaufwand entsprechend klein sein. Zielsetzung zukünftiger SEU muß es jedenfalls sein, im Laufe der Zeit alle Aufgaben, die gegenwärtig durch manuelle Konsistenzhaltung mit entsprechenden Absprachen und Regeln gelöst werden, durch Werkzeuge zu unterstützen.

Intelligenz, Integration, Flexibilität: Intelligenz heißt, daß das Arbeiten auf komplexen Konfigurationen mit einer komplizierten Innenstruktur unterstützt wird. Integrierte SEU müssen deshalb strukturbezogene Werkzeuge anbieten. Strukturbezogenheit darf sich nicht allein auf Dokumente beziehen, sondern muß die vielen Querbezüge zu anderen Dokumenten verwalten. Zukünftig muß es auch möglich sein, verschiedene Projekte, die miteinander zusammenhängen, zu handhaben, gemeinsame Teilprojekte herauszuziehen usw.

Die Handhabung von Konfigurationen muß jedoch eine entsprechende Flexibilität von Prozessen zulassen.

spezifische Entwicklungsumgebungen: Unter spezifischen SEU versteht man SEU, die für einen Anwendungsbereich passende Hilfsmittel anbieten und darüberhinaus passende Werkzeuge für andere Software. Voraussetzung dafür ist, daß bei der Entwicklung von Software Anwendungstechnik (Wissen über den Anwendungsbereich und entsprechende Hilfsmittel) und/oder Strukturklassentechnik (Kenntnis über die Struktur einer Softwareklasse und entsprechende Hilfsmittel) vorliegt.

Ein weitläufigerer Begriff von SEU hat mit Software nicht mehr viel zu tun. Sie beziehen sich auf spezielle Sprachen, Methoden, Modelle eines Anwendungsbereichs, den speziellen Anwendungsbereich und die Klasse von Systemen, die entwickelt werden soll. Hierbei kann das Ergebnis auch eine andere Entwicklungs- oder Modellierungsumgebung sein (z.B. CIM, VLSI-Umgebung, ...). Diese haben aber die gleiche interne Struktur wie eine SEU, es entstehen die gleichen Probleme bei der Realisierung. Insoweit sollten SEU-Entwickler mit diesen Bereichen zusammenarbeiten, um Erfahrungen auszutauschen.

2.4.2 Formale Spezifikation mit Π (Dirk Niemann)

Als Grundlage für dieses Seminar diente [SG94].

Der Begriff der Software-Komponente in Π

Da Π eine komponenten-orientierte Softwareentwicklung unterstützt, sollte zunächst der Begriff der Software-Komponente spezifiziert werden.
Als grundlegende Definition läßt sich eine Software-Komponente charakterisieren als eine Einheit aus

- zur Verfügung gestellter Funktionalität,
- zugehöriger Aufruf-Schnittstelle und
- lokalen Strukturen zur Implementierung der Funktionalität, evtl. mit Aufrufen weiterer Komponenten.

Im Zusammenhang mit Π lassen sich weiterhin folgende Eigenschaften festlegen:

- Jede Komponente kapselt einen oder mehrere Datentyp(en).
- Die Benutztbeziehungen zwischen Komponenten sind azyklisch, d.h. man erhält eine **hierarchische Strukturierung** des Systems.
- Mehrere Komponenten können zu einer Komponente zusammengefaßt werden (**Komposition von Komponenten**).
- Jede Komponente enthält ein explizit anzugebenes Import-Interface. Bei Π Konzept des **formellen Imports**:
  Unabhängige Entwicklung einzelner Komponenten, aber später explizite Angabe von Beziehungen zwischen den Komponenten (passendes Import- und Export-Interface).
- Importierte Datentypen können ohne Änderung exportiert werden (**Common Parameters**).

Basierend auf den genannten Definitionen kann eine Software-Komponente als 5-Tupel SC=(SCN, EXP, IMP, CP, BODY) aufgefaßt werden:

- **SCN**: Name der Komponente
- **EXP**: nicht leere Menge von Datentypen (*Export Section*)
- **IMP**: evtl. Leere Menge von Datentypen (*Import Section*)
- **CP**: evtl. Leere Menge von Datentypen (*Common Parameters section*)
- **BODY** $\supseteq$ EXP: nicht leere Menge von Datentypen (*Body section*)

Weiterhin werden folgende Begriffe definiert:

- **EXP_INT** := EXP $\cup$ CP (*Export Interface*)
- **IMP_INT** := IMP $\cup$ CP (*Import Interface*)

Die **Body Section** einer Komponente enthält die Operationen der eingekapselten Datentypen. Man unterscheidet:

- **ps-construction** (programming-in-the-small):
  Realisierung einzelner Datentypen in einer Komponente (*Atomic Component*)
- **pl-construction** (programming-in-the-large):
  Verbindung mehrerer Komponenten zu einer neuen, komplexeren Komponente (*Composite Component*)
- **basic-construction**:
  Keine Informationen bzgl. der verwendeten Operationen vorhanden (*Basic Component*)

Spezifikation von Software-Komponenten in Π

Die Idee hierbei ist die Spezifikation der verschiedenen Operationen mit Hilfe unterschiedlicher Sichtweisen (*Views*). Views sind dabei Spezifikationen, die einen bestimmten Abstraktionsgrad eines Datentyps darstellen, d.h. man berücksichtigt nur bestimmte Eigenschaften eines Datentyps, während andere zunächst ignoriert werden. Im Prinzip repräsentiert jedes View-Konzept eine eigene Sprache, d.h. Π ist also eine Vereinigung mehrerer Sprachen, die von präzisen, formellen Sprachen bis zu natürlichen Sprachen reichen können.

Spezifikation des Interfaces
Zur Spezifikation des Interfaces existieren momentan drei verschiedene Views:

- **Type View** definiert von der Ausführung unabhängige Eigenschaften eines Datentyps mit Hilfe algebraischer Spezifikationen.
- **Imperative View** definiert auch 'Seiten-Effekte' von Operationen auf Objekten durch präzise Angabe aller Operationen.
- **Concurrency View** definiert mögliche Reihenfolgen bei der Ausführung von Operartionen.

Type View
Für viele Datentypen wie Listen, Mengen, Warteschlangen usw. charakterisieren die invarianten Eigenschaften des Datentyps gleichzeitig dessen Funktionalität. Ausnahmen:

- Häufig muß auf die interne Struktur des Datentyps zurückgegriffen werden.
- Datentypen, deren Operationen Sequenzen von Objektzuständen erzeugen und nicht nur einen eindeutigen Endzustand.

Imperative View
Durch die Klassifizierung der Parameter in in-, inout- und return-Parameter können auch Seiten-Effekte berücksichtigt werden. Da jede Operation in Π entweder einen inout- oder einen return-Parameter hat, kann man zwei Arten von Operationen unterscheiden:

- Objekt modifizierende Operationen
- Objekt liefernde Operationen

Concurrency View
Angaben darüber, zu welcher Zeit welche Operationen zulässig sind und welche nicht, werden mit Hilfe der Concurrency View spezifiziert. Folgende Eigenschaften bzgl. der Ausführung einzelner Operationen können berücksichtigt werden:

- **Reihenfolgen**
  Die Festlegung der Reihenfolge geschieht durch reguläre Ausdrücke (*modular path expression*). Zulässige Operatoren sind hierbei:

Operator	Symbol	Bedeutung
sequence	a ; b	b wird nach a ausgeführt
alternation	a | b	entweder a oder b wird ausgeführt
concurrency	a + b	keine Reihenfolge festgelegt
repetition	[a]	wiederholte Ausführung von a
simultanity	{a}	simultane Ausführung von a
option	(* a *)	Ausführung von a kann übersprungen werden

- **Vorbedingungen** Hierdurch ist es möglich Operationen zu 'kontrollieren' die nur bei bestimmten Zuständen eines Objekts definiert sind, z.B. ist die Operation *top* des Datentyps Stack nur definiert, wenn der Stack nicht leer ist.

Konstruktion von Komponenten
In der Body Section einer Komponente werden die in der Export Section deklarierten Datentypen realisiert. Hierzu können neben den im Import-Interface angegebenen Datentypen beliebige weitere Datentypen und Operationen benutzt werden. Es werden zwei verschiedene Typen von Komponenten unterschieden:

- **CEM**: In dessen Body wird ein einziger Datentyp realisiert (vgl. *atomic component*)

- **Konfigurationen**: Hierbei handelt es sich um die Komposition mehrerer bereits existierender Komponenten (vgl. *composite component*)

Body Section bei CEMs
Zur Spezifikation der Body Section existieren momentan zwei verschiedene Views:

- **Type View Body Specification** gibt die Realisierung eines Datentyps durch algebraische Gleichungen an. Dies geschieht mit Hilfe eines sogenannten Konstruktions-Datentyp (constructing data type) und evtl. weiteren importierten Typen. Zur Realisierung können natürlich auch nicht sichtbare Operationen verwendet werden. Der Vorteil dieser Spezifikationsmethode ist die Analogie zur Type View Export Specification, d.h. es läßt sich leicht nachvollziehen, ob der Body eine korrekte Realisierung des exportierten Datentyps wiedergibt. Der Nachteil ist allerdings, daß eine Ausführung nur eingeschränkt möglich ist. Daher wird diese Methode hauptsächlich zum Prototyping verwendet.

- **Imperative View Body Specification** beschreibt die Implementierung mittels imperativen Algorithmen. Aufbau und grundlegende Elemente wie Schleifen, Verzweigungen ähneln stark anderen Programmiersprachen wie Pascal, Modula-2. Zur Unterstützung 'konkurrierender' Ausführungen sind zusätzlich cobegin-coend-Blöcke sowie zum Datenaustausch Shared Objects verwendbar.

Spezifikation von Konfigurationen
Die Spezifikation besteht aus

- der Aufzählung aller verwendeten Komponenten,
- der Bestimmung der Verbindungen zwischen den einzelnen Komponenten und
- dem Anpassen der Datentypen des Bodys mit denen des Interfaces.

Component Incarnation
Komponenten können in Π isoliert entwickelt werden. Dies hat allerdings zur Folge, daß die Parameter einer Komponente erst im Kontext anderer Komponenten eindeutig spezifiziert werden können. So kann z.B. der Datentyp eines Items eines Stacks sowohl ein Integer-Wert als auch eine komplette Operation sein. In diesem Abschnitt der Spezifikation werden die den Datentypen zugrundeliegenden Komponenten zugewiesen.
Component Connection
In dem Component Connection-Block werden die einzelnen Komponenten miteinander verbunden. Prinzipiell: Jedem exportierten Datentyp wird ein importierter Datentyp zugeordnet. Weiterhin werden in diesem Abschnitt die im Body genannten Datentypen denen des Interfaces zugeordnet. Hierzu werden die Begriffe Export, Import und Common Parameters als eine Art 'Komponenten' verwendet.

Teil II

Spezifikation

3. Überblick zur Spezifikation

Bevor wir mit der Spezifikation beginnen, möchten wir unsere Vorgehensweise motivieren. Die Software-Erstellung größerer Projekte kann nicht mehr in einem großen Block erfolgen. In den siebziger Jahren war das große Schlagwort *prozedurales Programmieren*, also die Top-down-Aufteilung von Problemen. Aus diesem hervorgegangen ist das *modulorientierte Programmieren*, also die Aufteilung eines Problems in relativ große, unabhängige Teilprobleme. In den neunziger Jahren ist der Begriff *objektorientiertes Programmieren* zum Schlagwort geworden.

Unsere Rahmenbedingung ist es, die am Lehrstuhl entwickelte Programmiersprache ProSet zu verwenden. Da diese das objektorientierte Programmieren nicht unterstützt, haben wir das Problem modulorientiert in drei große, unabhängige Teilbereiche unterteilt.

ProSet bietet nicht die Möglichkeit, eine grafische Benutzungsschnittstelle zu programmieren, da ProSet dem algorithmischen Prototyping dient. Das Spiel Scotland Yard soll jedoch eine ansprechende Oberfläche bieten. Daher wird die grafische Benutzungsoberfläche mit einem anderen Werkzeug erstellt werden. Wir werden die Begriffe grafische Benutzungsoberfläche und die in der Informatik gebräuchliche englische Abkürzung GUI – für *graphical user interface* – im folgenden synonym nebeneinander verwenden.

Daraus ergibt sich ein weiteres Modul, denn ProSet bietet zur Zeit noch keine akzeptable Schnittstelle zu anderen Sprachen. Deshalb muß außerdem eine Schnittstelle zwischen den ProSet-Moduln und der grafischen Benutzungsoberfläche erstellt werden.

3.1 Unterteilung des Problems

Die gestellte Aufgabe kann aufgrund der uns vorgegebenen Restriktionen[1] in drei etwa gleichwertige Teilaufgaben unterteilt werden. Neben der oben erwähnten grafischen Benutzungsoberfläche scheint es sinnvoll, die Regeln zusammen mit dem Spielbrett in einem zweiten Modul zu realisieren und die Strategien in einem dritten Modul zu implementieren. Damit ist gewährleistet, daß jedes Modul unabhängig von den anderen

[1] Damit meinen wir die uns vorweggenommene Wahl der Programmiersprache.

Moduln ist. Es ist zum Beispiel denkbar, zu Testzwecken einen kleineren Stadtplan zu verwenden, ohne daß die Planung geändert werden muß.

Die einzelnen Teilaufgaben werden in den folgenden Kapiteln spezifiziert. Die grafische Benutzungsoberfläche wird in Kapitel 4, die Interprozeßkommunikation in Kapitel 5 und das verwendete Nachrichtenprotokoll in Kapitel 6 spezifiziert. Die Treiber- und Regelkomponente wird in Kapitel 8 und Kapitel 7 und die Planungskomponente in Kapitel 9 spezifiziert.

4. Aufbau und Funktionalität der grafischen Benutzungsoberfläche

4.1 Einleitung

In diesem Kapitel wird die grafische Benutzungsoberfläche spezifiziert. Zunächst gehen wir auf die verwendete Sprache ein und warum diese gewählt wurde (Abschnitt 4.2). Danach werden die verwendeten Begriffe erklärt (Abschnitt 4.3), anschließend der grobe Aufbau der grafischen Benutzungsoberfläche beschrieben (Abschnitt 4.4). Schließlich werden die einzelnen Fenster beschrieben:

- das Hauptfenster in Abschnitt 4.5,
- das Konfigurationsfenster in Abschnitt 4.6.

4.2 Auswahl der zu verwendenden Sprache

Als mögliche Werkzeuge zur Realisierung der GUI kommen in Betracht:

- ISA-Dialogmanager
- Tcl/Tk
- C-Bibliotheken wie z.B. Xlib

Da der Schwerpunkt der GUI in der Darstellung des Spielfelds liegt, kommt es darauf an, daß das Werkzeug entsprechende Funktionalitäten zur Verfügung stellt. Da dies mit dem ISA-Dialogmanager nicht und mit C-Bibliotheken nur sehr umständlich zu realisieren ist, fällt die Wahl auf Tcl/Tk.

4.3 Erläuterung der verwendeten Begriffe

In den folgenden Abschnitten werden Elemente der grafischen Benutzungsoberfläche erwähnt, die an dieser Stelle definiert werden.

Button: Ein Button ist ein Schaltfeld, das mit der Maus gedrückt werden kann, wodurch eine bestimmte Aktion ausgelöst wird.

Canvas: Ein Canvas ist ein Objekt, in dem benutzerdefinierte Objekte wie Linien, Kreise o.ä. eingebunden werden können.

Frame: Ein Frame dient zur Gruppierung verschiedener Objekte.

Menü: Menüs sind grafische Objekte, über die der Benutzer verschiedene Befehle aktivieren oder Optionen setzen und löschen kann. Ein Menü besteht aus einem Menübalken, in dem mehrere Einträge nebeneinander stehen, die der Benutzer mit der linken Maustaste aktivieren kann. Daraufhin öffnet sich ein Pulldown-Menü, in dem die Einträge untereinander stehen. Diese kann der Benutzer wiederum mit der linken Maustaste aktivieren.

Separator: Ein Separator ist eine waagerechte Linie, die in Pulldown- und Popup-Menüs eingefügt werden kann, um ein längeres Menü für den Benutzer übersichtlicher zu gliedern.

Scalebar: Ein Scalebar ist ein grafisches Objekt, das wie ein Regler an einem Mischpult aussieht. Der Knopf des Reglers kann mit der Maus verschoben werden. Durch das Verschieben des Reglers werden Werte eingestellt.

Scrollbar: Ein Scrollbar ist ein Rollbalken, d.h. ein grafisches Objekt, mit dessen Hilfe Bereiche zur Darstellung von Objekten auf dem Bildschirm verschoben werden können, wenn diese Objekte nicht ganz in dem gewählten Bildschirmbereich dargestellt werden können. Sie kommen in der Regel bei Listboxen oder größeren Strukturen wie Canvases vor.

4.4 Aufbau der Benutzungsoberfläche

Die grafische Benutzeroberfläche wird durch drei Fenster realisiert.

- Das Hauptfenster wird immer angezeigt und enthält als wichtigstes Element das Spielbrett.
- Das Konfigurationsfenster dient zum Verändern bestimmter Spielparameter. Es wird vor jedem Start eines neuen Spiels angezeigt.
- Ein weiteres Fenster zeigt die benutzten Tickets von Mister X an.

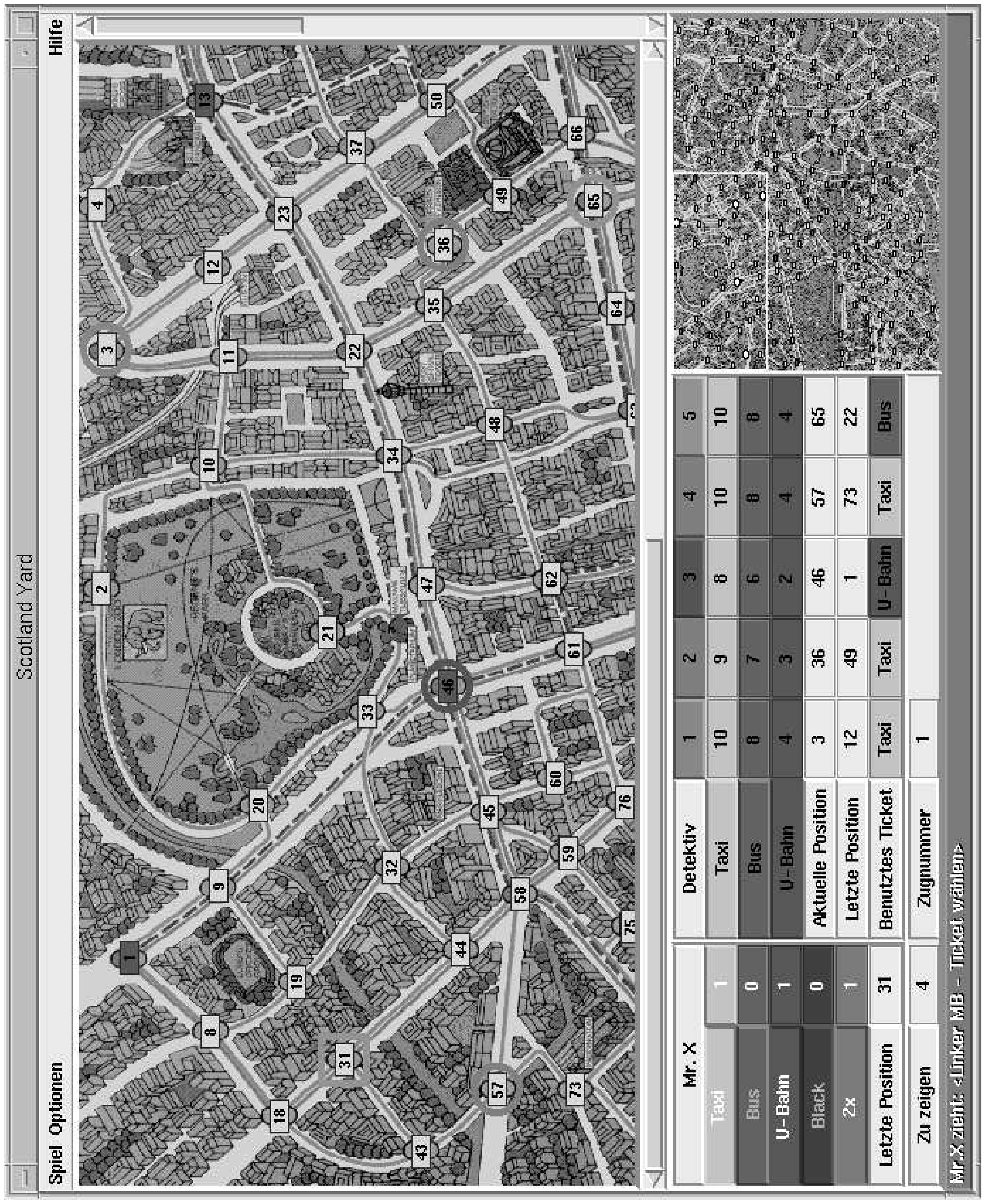


Abbildung 4.1: Das Hauptfenster mit Spielfeld (oben) und Informationen zu Mister X und den Detektiven (unten).

4.5 Das Hauptfenster

Das Hauptfenster (Abb. 4.1) ist in zwei Bereiche eingeteilt. Der obere Teil als größter Fensterbereich enthält das Spielbrett und die *Scrollbars*, mit denen der sichtbare Ausschnitt verschoben werden kann. Im unteren Teil werden aktuelle Informationen zu Mister X und zu den Detektiven angezeigt.

Außerdem enthält das Hauptfenster ein Menü (ganz oben) und eine Statuszeile (ganz unten).

4.5.1 Das Menü

Es werden drei Pulldown-Menüs angeboten, das DATEI-Menü, das OPTIONEN-Menü und das HILFE-Menü.

Datei-Menü: Das DATEI-Menü enthält fünf Einträge:

- Mit *Neues Spiel* wird ein neues Spiel gestartet. Dieser Eintrag wird deaktiviert[1], sobald der Spieler ein Spiel begonnen hat. Hat er ein Spiel beendet, ist der Eintrag wieder aktiviert.
- Mit *Spiel laden* kann ein zuvor gespeicherter Spielstand geladen werden. Dieser Eintrag wird deaktiviert, sobald der Spieler ein Spiel begonnen hat. Hat er ein Spiel beendet, ist der Eintrag wieder aktiviert.
- Mit *Spiel speichern* kann der aktuelle Spielstand gespeichert werden. Dieser Eintrag ist nur dann aktiviert, wenn Mister X am Zug ist.
- Mit *Spiel abbrechen* kann ein laufendes Spiel abgebrochen werden. Es erfolgt eine Sicherheitsabfrage, um die Aktion zu bestätigen. Dieser Eintrag ist deaktiviert, wenn das Programm gestartet wurde. Er wird aktiviert, sobald der Spieler ein neues Spiel beginnt. Hat der Spieler ein laufendes Spiel abgebrochen, dann wird der Eintrag wieder deaktiviert.
- Von den oben beschriebenen Menüeinträgen des DATEI-Menüs durch einen *Separator* optisch abgesetzt erscheint der immer aktive Eintrag *Beenden*, mit dem gegebenenfalls ein laufendes Spiel abgebrochen und das Programm beendet wird. Es erfolgt eine Sicherheitsabfrage, um die Aktion zu bestätigen.

Optionen-Menü: Das OPTIONEN-Menü enthält sieben Einträge.

- Mit *Sound* wird die Sound-Unterstützung ein- bzw. ausgeschaltet. Um zu garantieren, daß die Klang-Ausgabe stets auf dem richtigen Rechner geschieht, ist eine Aktivierung nur möglich, wenn das Spiel auf der Console gestartet wurde.

[1] Ein deaktivierter Menüeintrag ist daran zu erkennen, daß er mit dunkelgrauer Schrift angezeigt wird im Gegensatz zur schwarzen Schrift eines aktivierten Eintrags.

- Durch Aktivierung des Menüeintrags *Benutzte Tickets* kann ein Fenster geöffnet werden, in dem die von Mister X benutzten Tickets aufgelistet werden.
- Durch Aktivierung des Menüeintrags *Detektiv-Züge zeigen* werden alle bisherigen Züge der Detektive im Spielfeld eingezeichnet.
- Wählt man den Menüeintrag *Detektiv-Züge löschen* , so werden alle eingezeichneten bisherigen Züge der Detektive gelöscht.
- Mit *Mr.X Positionen zeigen* werden alle Felder markiert, die von den Detektiven als mögliche Mister X-Positionen angesehen werden.
- Durch Aktivieren des Menüeintrags *Detektive bewegen* werden die Figuren der Detektive auf dem Spielfeld bewegt.
- Mit *Alle Marker löschen* werden alle erzeugten Marker gelöscht[2].

Hilfe-Menü: Das HILFE-Menü enthält drei Einträge:

- Mit *Spielregeln* wird ein neues Fenster geöffnet, in dem die Spielregeln als ASCII-Text angezeigt werden. Außerdem enthält das Fenster einen *Scrollbar*, mit dem der sichtbare Ausschnitt des Hilfetextes verschoben werden kann, und einen *Button*, der mit SCHLIESSEN beschriftet ist und mit dem das Fenster wieder geschlossen werden kann.
- Mit *Bedienung* wird wie bei dem Eintrag *Spielregeln* ein Fenster geöffnet, in dem die Bedienungshinweise als ASCII-Text angezeigt werden. Das Fenster enthält außerdem einen *Scrollbar*, mit dem der sichtbare Ausschnitt der Bedienungshinweise verschoben werden kann, und einen *Button*, der mit SCHLIESSEN beschriftet ist und mit dem das Fenster wieder geschlossen werden kann.
- Von den oben beschriebenen Einträgen des HILFE-Menüs durch einen *Separator* optisch abgesetzt erscheint der Eintrag *Info*, mit dem ein Fenster mit Informationen über das Programm — also Version, Programmierer usw. — angezeigt wird. Über einen mit OK beschrifteten *Button* kann das Fenster wieder geschlossen werden.

4.5.2 Das Spielbrett

Das Spielbrett wird mit Hilfe einer *Canvas*-Struktur realisiert. In ihr wird u.a. die Grafik als `.gif`-Datei geladen. Mit *Scrollbars* kann der dargestellte Ausschnitt über das Brett bewegt werden. Weiterhin werden hier die Positionen der einzelnen Spielfiguren angezeigt. Die Detektive werden durch entsprechend gefärbte Kreise dargestellt, Mister X durch ein Quadrat.

[2] Marker können durch Doppelklicken mit der linken Maustaste auf dem Spielfeld erzeugt werden.

4.5.3 Informationen zu Mister X und den Detektiven

Im unteren Bereich des Hauptfensters werden aktuelle Informationen zu Mister X und den Detektiven angezeigt. Links sind die Informationen zu Mister X.

Informationen zu Mister X

Der *Frame* mit den Informationen zu Mister X ist oben mit *Mr. X* beschriftet. Darunter werden fünf *Buttons* mit je einem rechts daneben positionierten Textfeld angezeigt:

1. Taxi, in gelb,
2. Bus, in grün,
3. U-Bahn, in rot,
4. Black, in schwarz,
5. 2x, in orange.

In den Textfeldern steht die jeweilige Anzahl von Tickets der entsprechenden Art. Steht dem Spieler ein Ticket einer Art nicht mehr zur Verfügung, wird der entsprechende *Button* deaktiviert und vertieft dargestellt[3]. Hat der Spieler einen Spielzug mit einem 2x-Ticket gezogen, dann wird für diesen Zug der *Button* 2x deaktiviert (aber nicht vertieft) dargestellt.

Unter den fünf *Buttons* wird die letzte Position angezeigt, auf der Mister X sich gezeigt hat, die Anzahl der Züge, nach denen sich Mister X wieder zeigen muß und die Zugnummer des aktuellen Zuges.

Informationen zu den Detektiven

Die Informationen zu den Detektiven werden links neben den Informationen zu Mister X angezeigt und sehen auch ähnlich aus. Die Anzeige ist tabellarisch aufgebaut. In der linken Spalte steht als Überschrift Detektiv, darunter steht Taxi, Bus, U-Bahn, Aktuelle Position, Letzte Position und Benutztes Ticket. Daneben befinden sich entsprechend so viele Spalten wie Detektive am Spiel teilnehmen. Die Köpfe dieser Spalten enthalten die Nummer des Detektivs und haben eine Hintergrundfarbe entsprechend den Farben der Kreise, mit dem der jeweilige Detektiv auf dem Spielbrett angezeigt wird. Folgendes bedeutet der Inhalt der Zeilen:

Taxi: Anzahl der verbleibenden Taxi-Tickets des Detektivs.

Bus: Anzahl der verbleibenden Bus-Tickets des Detektivs.

U-Bahn: Anzahl der verbleibenden U-Bahn-Tickets des Detektivs.

[3] Ein deaktivierter *Button* wird, analog zu Menüeinträgen, mit dunkelgrauer Schrift dargestellt.

Aktuelle Position: Feldnummer des Feldes, auf dem der Detektiv zur Zeit steht.

Letzte Position: Feldnummer des Feldes, auf dem der Detektiv vor dem letzten Spielzug stand.

Benutztes Ticket: Ticketart des zuletzt verwendeten Tickets.

4.5.4 Die Statuszeile

In der Statuszeile werden weitere aktuelle Informationen angezeigt, zum Beispiel kann dort stehen, was der Spieler als nächstes machen soll oder darf.

4.6 Das Konfigurationsfenster

Wenn der Spieler ein neues Spiel startet, wird das Konfigurationsfenster (Abb. 4.2) angezeigt. Zu Beginn sind dort die Defaultwerte eingestellt, die vom Treiber in der Initialisierungsphase gesendet werden. Diese Einstellungen lehnen sich an die Spielregeln des Originalspiels an.

Das Konfigurationsfenster enthält im wesentlichen zwei Bereiche, in denen

- die Vorgaben zu Mister X verändert
- die Vorgaben zu den Detektiven verändert

werden können. Außerdem enthält das Fenster eine Menüzeile (ganz oben) und zwei *Buttons* (ganz unten).

4.6.1 Das Menü

Es werden zwei Pulldown-Menüs angeboten: OPTIONEN und HILFE.

Optionen: Das OPTIONEN-Menü enthält drei Einträge:

- Mit *Spielplan* wird ein weiteres Pulldown-Menü geöffnet, in dem aus den angezeigten Spielplänen einer ausgewählt werden kann.
- Mit *Spielstärke* wird ein weiteres Pulldown-Menü geöffnet, in dem aus den angezeigten Spielstärken eine ausgewählt werden kann.
- Mit *Voreinstellung* können die Defaultwerte eingestellt werden.

Hilfe: Das HILFE-Menü enthält zwei Einträge:

- Mit *Konfiguration* wird ein Fenster mit einer kurzen Hilfe zum Konfigurationenfenster, das mit einem OK-Button wieder geschlossen werden kann.
- Mit einem *Separator* von dem ersten Eintrag optisch abgesetzt ist der zweite Eintrag. *Info*, mit dem dasselbe Infofenster wie bei dem entsprechenden Menüeintrag des HILFE-Menüs geöffnet wird.

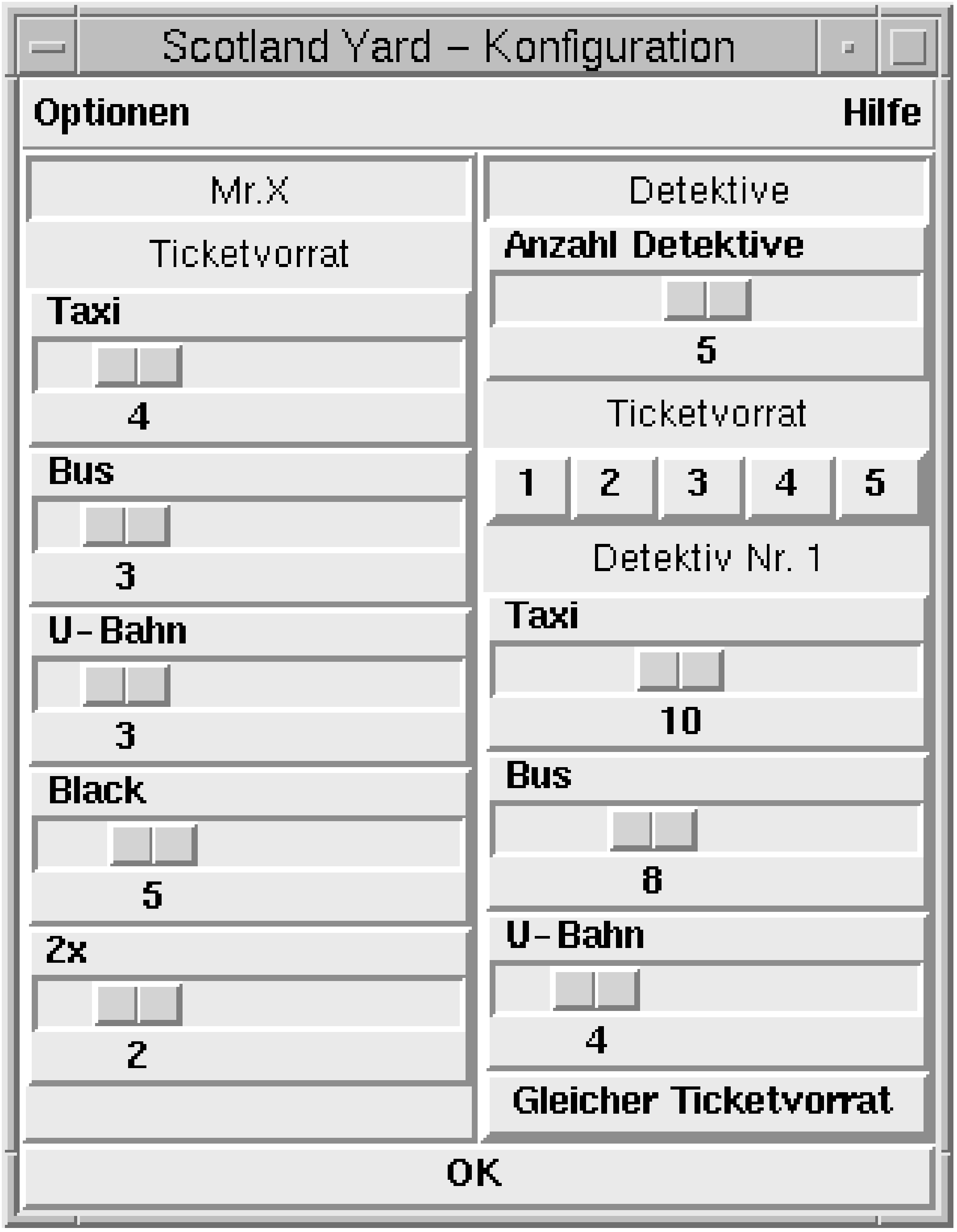


Abbildung 4.2: Das Konfigurationenfenster mit den Werten für Mister X (links) und und den Werten zu den Detektiven (rechts).

4.6.2 Einstellungen zu Mister X

In der linken Hälfte des Konfigurationenfensters können die Einstellungen zu Mister X verändert werden. Dieser Bereich ist mit MR. X überschrieben. Darunter steht TICKETVORRAT. Die Einstellungen können mittels *Scalebars* variiert werden, die untereinander angeordnet sind und mit der entsprechenden Ticketart beschriftet sind.

4.6.3 Einstellungen zu den Detektiven

In der rechten Hälfte des Konfigurationenfensters können die Einstellungen zu den Detektiven verändert werden. Dieser Bereich ist mit DETEKTIVE überschrieben. Darunter befindet sich ein mit ANZAHL DETEKTIVE beschrifteter *Scalebar*, mit dem die Anzahl der Detektive variiert werden kann. Darunter befinden sich in Abhängigkeit von der gewählten Anzahl der Detektive ein bis neun *Buttons*, die von links nach rechts durchnummeriert sind. Über den *Buttons* steht TICKETVORRAT. Mit diesen *Buttons* kann der Detektiv ausgewählt werden, für den die Einstellungen verändert werden sollen. Der aktuell ausgewählte Detektiv wird unter den *Buttons* angezeigt.

Darunter befinden sich drei *Scalebars*, beschriftet mit der jeweiligen Ticketart. Hiermit können die einzelnen Ticketvorräte variiert werden.

Wünscht der Spieler für alle Detektive die selben Werte, kann er, *nachdem* er alle Einstellungen wunschgemäß vorgenommen hat, auf den *Button* GLEICHER TICKETVORRAT drücken.

4.6.4 Die Buttons

Ganz unten im Fenster befinden sich nebeneinander zwei *Buttons*, beschriftet mit OK und ABBRUCH. Mit ersterem können die Einstellungen für das begonnene Spiel übernommen werden, der zweite bewirkt, daß die Einstellungen verworfen werden. Beide Buttons bewirken ein Schließen des Konfigurationsfensters.

5. Entwurf und Spezifikation der IPC

5.1 Motivation der IPC

Das in ProSet zu implementierende Scotland Yard soll sich dem Benutzer mit einer grafischen Benutzungsoberfläche präsentieren. ProSet selbst verfügt allerdings über keinerlei Unterstützung zur Entwicklung oder Einbindung von grafischen Bedienungselementen. Es muß also eine einfache Schnittstelle zwischen ProSet und einer GUI unterstützenden Sprache geschaffen werden. Eine ProSet/C Schnittstelle ist vorhanden, sie kann allerdings nur einige atomare Datentypen sowie C-Strings handhaben. Ferner ermöglicht sie nur, aus ProSet heraus C-Funktionen zu benutzen, nicht jedoch aus C heraus ProSet Prozeduren zu verwenden.

Die Einbindung von C-Libraries zur Implementation einer grafischen Benutzungsoberfläche verbietet sich zum einen aufgrund der zu ProSet inkompatiblen C-Datenstrukturen solcher Libraries, und zum anderen wegen der fehlenden Möglichkeit, Callbacks von C heraus auf ProSet abzubilden. Daher wird die Erstellung eines separaten Programmes vorgeschlagen, welches via Interprozeßkommunikation mit dem in ProSet implementierten Kern des Scotland Yard Spiels kommuniziert.

So können aus ProSet heraus Ereignisse von der Benutzungsoberfläche entgegengenommen werden. Weiterhin ist durch diesen Ansatz auch sichergestellt, daß die grafische Benutzungsoberfläche unabhängig von dem ProSet Kern genügend Rechenzeit zur Verfügung hat, um auf Benutzeraktionen und auf Ereignisse des Fenstersystems (zum Beispiel Neuzeichenen von Fensterbereichen) in einem angemessenem Zeitrahmen reagieren zu können, ohne dabei in Konkurrenz zu ProSet zu stehen. Die Kommunikation soll in Form einer zum ProSet Programm linkbaren C-Library organisiert werden.

5.2 Alternativen und Anforderungen

In diesem Abschnitt werden die Anforderungen an die IPC beschrieben und die verschiedenen Alternativen, die uns bekannt sind, und aus denen unsere IPC entwickelt wurde.

5.2.1 Grundsätzliche Anforderungen

Da sich Scotland-Yard dem Benutzer mit einer graphischen Benutzerschnittstelle präsentieren soll, aber ProSet nicht über die Möglichkeit verfügt, eine solche zu integrieren, wurde zur Anbindung eines GUIs ein separater GUI Prozeß auserkoren. Dieser Prozeß stellt dem Benutzer nur das GUI zur Verfügung und reicht Benutzereingaben via IPC an ProSet weiter. Entsprechend werden ProSet Ausgabedaten via IPC an den GUI Prozeß übergeben. Es ist also völlig ausreichend die IPC darauf zu beschränken, einfach strukturierte Strings zu versenden. Unter einer einfachen Struktur soll verstanden werden, daß die Strings keine komplexe innere Struktur besitzen wie z.B. eine Listenrepräsentation oder ähnliches. Wird auf solche Strukturen verzichtet, so können folgende Vorteile verzeichnet werden:

- Einfache Stringstruktur
  - Festlegung einer maximalen Stringlänge für Nachrichten.
  - Kein Parser notwendig, simples Scannen reicht aus.
- Festlegung einer maximalen Stringlänge
  - Verwendung von Strings statischer Größe und damit die Vermeidung aller Probleme einer dynamischen Speicherverwaltung.
  - Die Prüfung von Stringüberläufen wird vereinfacht, da die maximale Stringlänge immer gleich ist.
  - Keine Aufteilung von Nachrichten in mehrere Teilnachrichten notwendig, wenn die zugrundeliegende IPC eine Maximalgröße für Nachrichten vorsieht.

Diese Anforderungen an die IPC sind dadurch vergleichsweise gering. Es liegt daher nahe auch eine möglichst einfache IPC zu verwenden, die den Anforderungen unter minimalem Implementationsaufwand gerecht wird.

5.2.2 IPC-Alternativen unter UNIX

Es bieten sich generell drei verschiedene Vorgehensweisen für die Implementation einer IPC an. Zum einen die intuitive Methode, zum anderen die Nutzung solcher Methoden, die explizit durch das Betriebssystem angeboten werden, und nicht zuletzt die Nutzung von externer Software. Da für die Projektgruppe das Betriebsystem UNIX als Basis genutzt wird, beschränken sich die im folgenden vorgestellten IPC Mechanismen auf das System UNIX. Aufgrund der UNIX Entwicklungsgeschichte existieren verschiedene, voneinander unabhängige UNIX Systemvarianten.

Intuitive Methoden

Betrachtet man die Vielfalt und Komplexität der von UNIX angebotenen IPC Mechanismen, erscheint es dem IPC Neuling einfacher, die IPC mit den ihm bereits vertrauten Mechanismen zu realisieren, anstatt sich in die reichlich vorhandenen Strukturen einzuarbeiten. Auf diese bereits "vertrauten Mechanismen" soll nun eingegangen werden.

IPC mittels Dateien Die wahrscheinlich älteste Methode der IPC unter UNIX ist wohl die Verwendung von zwei synchronisierten Dateien. Die Synchronisation kann entweder über UNIX Semaphore oder über exclusive Dateierzeugung stattfinden. Die Probleme hierbei bestehen in der explizit durch das Programm sicherzustellenden Synchronisation der Dateien einerseits und dem Handling von unerwarteten Kommunikationsabbrüchen andererseits (Systembedingt, Programmabsturz oder Fehler). Diese Methode ist aufgrund der explizit zu implementierenden Synchronisations- und Abbruchmechanismen als relativ aufwendig einzustufen.

IPC mittels FiFo's Eine andere Methode besteht in der Verwendung zweier UNIX FiFo's, (FiFo's werden auch als "anonymous pipes" bezeichnet) wobei je ein FiFo für eine Kommunikationsrichtung benutzt wird. Da unter UNIX ein Nodetyp FiFo existiert, übernimmt hier das System die Synchronisation zwischen den Lese- und Schreibaktionen, und versendet automatisch das Signal "broken pipe" an die entsprechenden Prozesse, falls die Kommunikation aus irgendeinem Grunde zusammenbricht.

Ein mögliches Kommunikationsmodell für die Verwendung von FiFo's zur Kommunikation zwischen dem ProSet Prozeß und dem GUI Prozeß wird in Abb. 5.1 dargestellt.

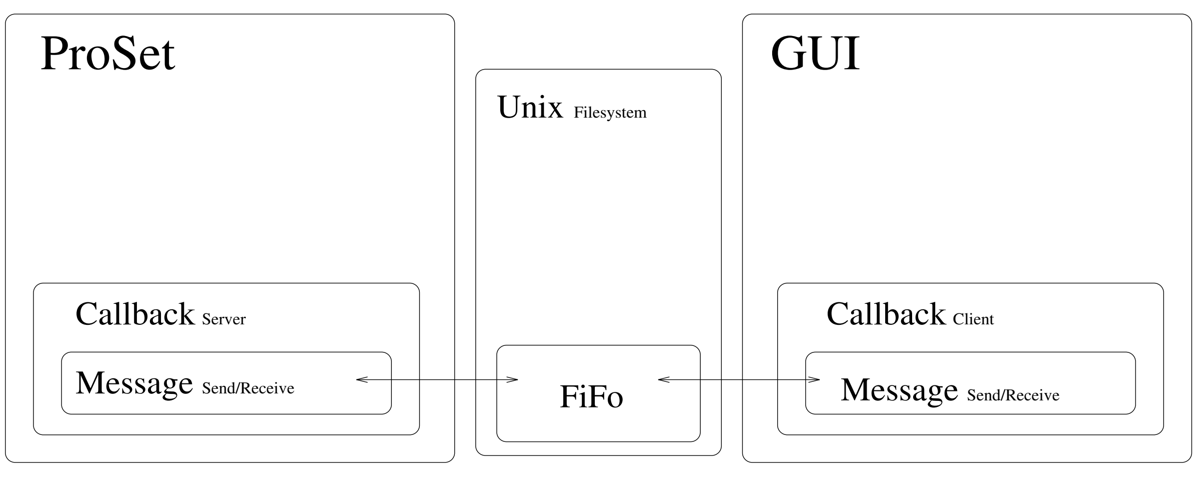


Abbildung 5.1: FiFo-IPC Kommunikationsmodell

Diese Methode wurde auch aufgrund ihres Leistungsumfanges in die engere Wahl gezogen.

SystemV-IPC

Das UNIX SystemV bietet Message Queues, Shared Memory, und Semaphore als IPC Mechanismus an. Die meisten dieser Mechanismen finden sich auch unter anderen UNIX Varianten wieder.

Message Queues Die SystemV Variante der UNIX Systeme bietet einen IPC Mechanismus an, der “SystemV Message Queues” genannt wird. Gemeinhin wird er schlicht als “SystemV-IPC” bezeichnet, obwohl der Begriff “SystemV-IPC” genaugenommen auch andere Mechanismen einschließt.

Die SystemV Message Queues sind ein Nachrichtensystem, das nach dem FiFo-Prinzip arbeitet. Zusätzlich kann jede Nachricht noch mit einer Zahl versehen werden, so daß der Empfänger nur Nachrichten mit einem bestimmten Zahlenwert der Queue entnehmen kann.

Ein mögliches Kommunikationsmodell für die Verwendung von SystemV-IPC zur Kommunikation zwischen dem ProSet Prozeß und dem GUI Prozeß wird in Abb. 5.2 dargestellt.

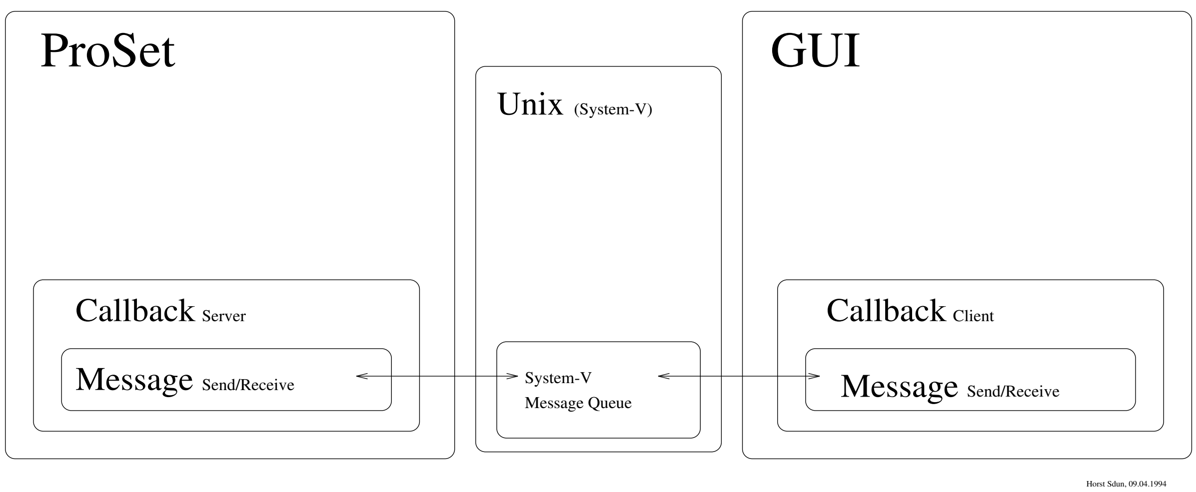


Abbildung 5.2: SystemV-IPC Kommunikationsmodell

Shared Memory Fast alle UNIX Varianten bieten SystemV Shared Memory als IPC Mechanismus an. Dieser Mechanismus ist nicht netzwerktauglich, bietet keine automatische Synchronisation, sowie keine Abbruchkontrolle. Als Gegenleistung erhält man maximale Performance. Shared Memory kann daher als eine “low-level” IPC angesehen werden.

Ein mögliches Kommunikationsmodell für die Verwendung von Shared Memory zur Kommunikation zwischen dem ProSet Prozeß und dem GUI Prozeß wird in Abb. 5.3 dargestellt.

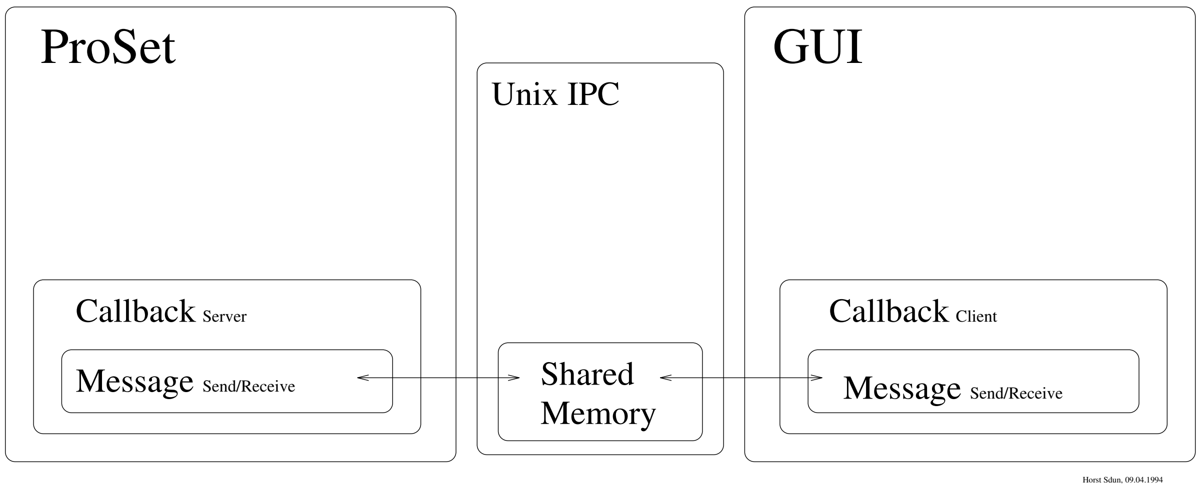


Abbildung 5.3: Shared Memory Kommunikationsmodell

Die Implementierung ist zu aufwendig. Die Performance für einen Prototypen nicht erforderlich.

NSL Zu der relativ jungen SystemV-NSL lagen mir leider keinerlei Unterlagen vor.

Berkeley UNIX

Die BSD-UNIX Variante bietet Sockets als IPC Mechanismus an. Dieser Mechanisus findet sich auch unter den meisten anderen UNIX Varianten wieder.

Sockets Auch viele nicht BSD-UNIX Varianten bieten sowohl UDP- als auch TCP-Sockets als Kommunikationsmechanismus an. Die Stärken der Sockets liegen vor allen Dingen in der Netzwerktauglichkeit mit der Einschränkung von nicht einheitlicher Datenrepräsentation. Die Zugriffe werden automatisch kontrolliert und Verbindungsabbrüche dem Prozeß gemeldet. Die Komplexität ist allerdings als hoch anzusehen.

Ein mögliches Kommunikationsmodell für die Verwendung von Sockets zur Kommunikation zwischen dem ProSet Prozeß und dem GUI Prozeß wird in Abb. 5.4 dargestellt.

Die Implementierung ist zu aufwendig. Netzwerktauglichkeit für einen Prototypen nur bedingt erforderlich.

Sonstige IPC (via RPC)

Einige UNIX Varianten bieten eigene IPC-Mechanismen an. Die meisten kann man wohl unter dem Begriff *Remote Procedure Call* einordnen. Hierunter fallen z.B. Xerox Courier und Sun RPC's.

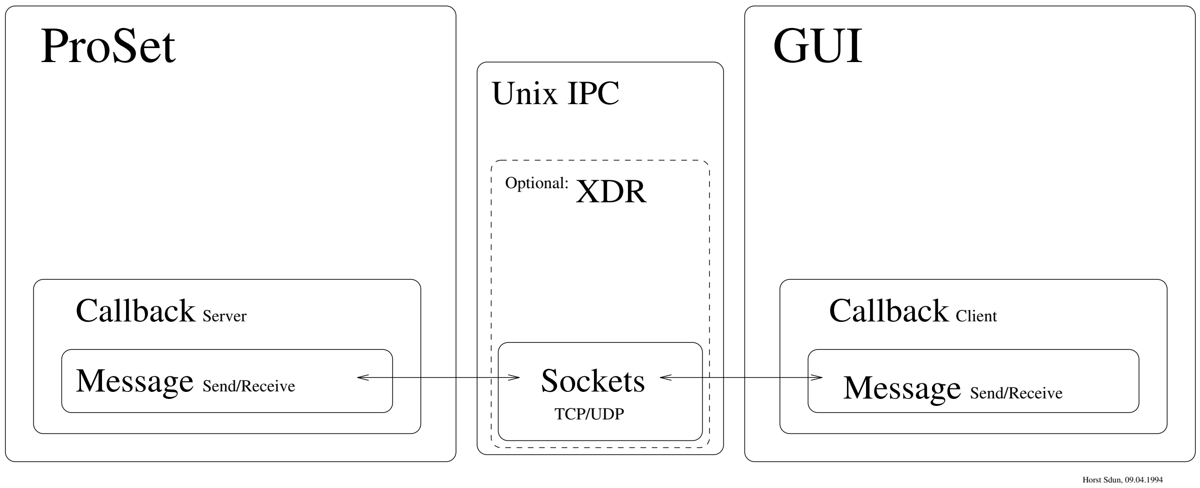


Abbildung 5.4: BSD-Socket Kommunikationsmodell

Sun RPC's Die von Sun Microsystems entwickelten und mittlerweile nicht zuletzt durch das Sun-NFS als Quasistandard etablierten RPC's stellen die wohl eleganteste Methode der IPC dar. Sie bieten Synchronisations- und Verbindungsabbruchkontrolle sowie eine unabhänige Datenrepräsentation (XDR). Die Komplexität ist für den Programmierer eher gering.

Es drängt sich allerdings hier die Frage auf, ob ein derartig gewaltiges Instrumentarium notwendig ist, um simple Strings zu versenden. Mit Sicherheit nicht.

Ein mögliches Kommunikationsmodell für die Verwendung von RPC's zur Kommunikation zwischen dem ProSet Prozeß und dem GUI Prozeß wird in Abb. 5.5 dargestellt.

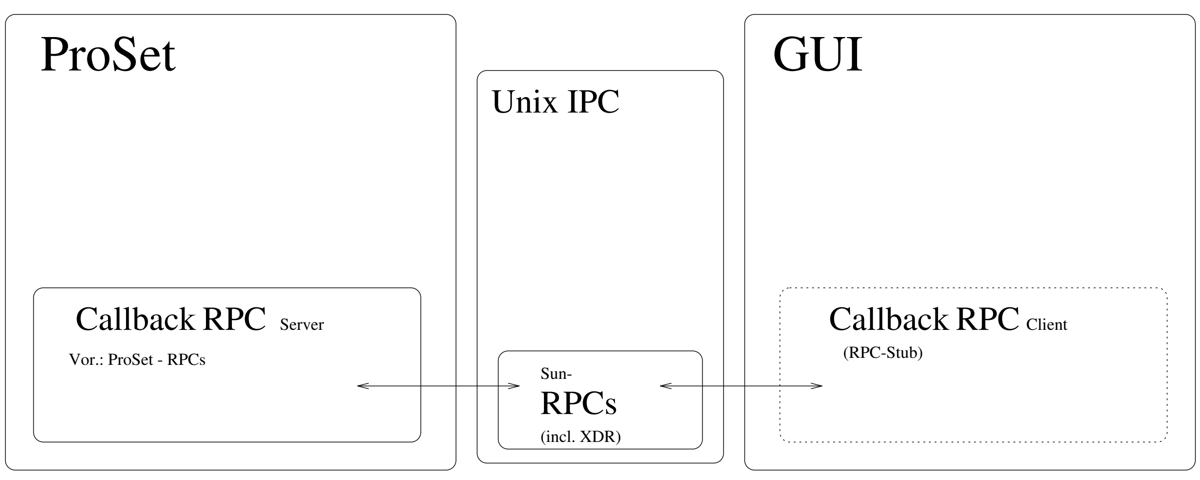


Abbildung 5.5: Sun-RPC Kommunikationsmodell

Externe Software - Linda

Da ProSet als IPC eine Linda Variante anbietet, liegt es nahe eine entsprechende C Version für die GUI Anbindung zu benutzten. Es existiert allerdings keine zu ProSet-Linda kompatible C-Linda Version.

Weitere Produkte lagen nicht zur Auswahl vor, oder sind nicht in diese einbezogen worden.

5.2.3 Weitere Anforderungen

Programmiersprache

Ein weiterer Punkt wurde bisher in den Anforderungen verschwiegen: Die Anbindungsart der IPC an ProSet und der noch auszuwählenden GUI. Als Implementierungsprache bietet sich von der UNIX Seite nur C an. ProSet produziert C-Code und bietet selbst auch eine C-Schnittstelle an, die es ermöglicht aus ProSet heraus C-Primitiven zu benutzen. Die meisten Interface Builder unter UNIX bieten eine C-Schnittstelle, da sie sich auf MIT's X11 abstützen. Die für Scotland-Yard in Frage kommenden GUIs bieten alle ausschließlich eine C-Schnittstelle. Die Wahl von C für die Implementation der GUI-IPC erscheint daher nur allzu logisch.

Client-Server Archiektur

Da in die Kommunikation stets nur zwei Parteien involviert sind und eine der Parteien einen Nachrichtenkanal öffnen (erzeugen) muß, ist es zweckmäßig eine Client/Server Architektur zu verwenden. Dies stellt auch die Verantwortlichkeit der Prozesse bezüglich des Erzeugens und Schließens eines Kanals klar heraus.

5.2.4 Entwurf

Die IPC Schnittstelle soll nur eine Art von Nachrichten übertragen: Strings. Dies vereinfacht sowohl das Protokoll als auch das Debugging mit entsprechenden Werkzeugen, mit denen einzugebende Strings versendet und empfangen werden können.

Nachrichten

Die zu verschickenden Nachrichten sollen in ihrer Länge auf eine noch zu definierenden Anzahl von Zeichen begrenzt werden und eine einfache Struktur aufweisen. (vgl. Abschnitt 5.2.1). Die Struktur sollte nur aus einem Nachrichtenschlüssel und den dazu gehörenden Parametern bestehen. Variable Anzahlen von Parametern sollten pinzipiell aufgrund der Längenbegrenzung von Nachrichten nicht unterstützt werden.

Die Schnittstelle

Die Schnittstelle benötigt nur wenige Funktionen. Diese können wie folgt beschreiben werden:

- Erzeugung eines Kanals
- Verbindungsaufbau
- Einstellen von Verbindungsmodi
- Senden von Nachrichten
- Empfangen von Nachrichten
- Verbindungsabbau

Die Schnittstelle wird hier außerdem noch gegenüber den Anforderungen erweitert. Sie soll nicht nur Strings, sondern beliebige binäre Daten versenden. Auf die Motivation hierzu wird in Abschnitt 12.2.1 eingegangen.

Namensfindung

Der zu implementierende Nachrichtenkanal wird mit dem Namen "Port" versehen, um einen Begriff zu benutzen, der nicht bereits durch die UNIX-IPC geprägt ist. Die dazugehörige Funktionensammlung wird "PortLibrary" genannt.

Die C-Schnittstelle

Für die C-Schnittstelle wurden unabhängig von der zugrundeliegenden IPC folgende Funktionenprototypen entworfen:

- Öffnen eines Ports

  ```
  int port_server_connect( const char *channel )
  ```

  Erzeugen und Öffnen eines Ports mit dem Namen `channel`. Der Name ist, mit einigen Einschränkungen, frei wählbar und muß entsprechend auf der Clientseite für die Verbindung angegeben werden. Bei erfolgreicher Öffnung des Ports liefert die Funktion den Wert 1 andernfalls den Wert 0. Die Einschränkungen beziehen sich auf die in Abschnitt 12.2.2 erläuterte IPC Schlüsselbildung.

- Verbinden mit einem existierenden Port

```
int port_client_connect( const char *channel )
```

Öffnen eines Kanals mit dem Namen `channel`. Der Name muß identisch mit dem Namen des durch den Server geöffnten Ports sein. Bei erfolgreicher Öffnung des Ports liefert die Funktion den Wert 1 andernfalls den Wert 0.

- Schließen eines Ports

```
int port_disconnect( void )
```

Schließen einer Verbindung auf Server oder Client Seite. Bei erfolgreichem Schließen des Ports liefert die Funktion den Wert 0 andernfalls einen Wert ungleich 0.

- Senden einer Nachricht

```
int /* length */ port_send( void *message, int length )
```

Einen Datensatz der Länge "length" an die andere Partei verschicken.

- Empfangen einer Nachricht

```
int /* length */ port_receive( void *message, int maxlen )
```

Einen Datensatz der anderen Partei erwarten. Es werden maximal `maxlen` Bytes der Nachricht in den Speicher `message` geschreiben, alle darüberhinausgehenden Bytes werden verworfen. Die Funktion liefert als "Returnwert" die Anzahl der gelesenen Bytes zurück.

- Zwischen blockierendem und nicht-blockierendem Empfang umschalten

```
void port_toggle_receive( int do_block )
```

Hat `do_block` den Wert 0, so wird nicht-blockierend empfangen (*liegt keine Nachricht vor, so liefert die Funktion die Nachrichtenlänge 0 zurück*), andernfalls wird blockierend empfangen (*liegt keine Nachricht vor, so wartet die Funktion solange bis eine Nachricht eintrifft und liefert diese zurück*).

- Die Ausgabe von Zusatzinformationen an und abschalten

```
void port_verbose_mode( short onoff )
```

Schaltet die Ausgabe von Debugging-Informationen, entsprechend dem Wert 1 bzw. 0 von `onoff`, auf dem Standardausgabekanal ein bzw. aus.

- Applikationsname der “PortLibrary” mitteilen

```
void port_application_name( const char *name )
```

Teilt der PortLibrary mit, von welcher Applikation sie benutzt wird. Der Name wird bei Fehlermeldungen und Debugging-Informationen mitausgegeben.

6. Nachrichtenaustausch zwischen dem GUI und der Treiberkomponente

6.1 Allgemeine Hinweise

Um eine Verständigung zwischen GUI und Treiber zu ermöglichen, werden Nachrichten über eine IPC versendet. Jede Nachricht ist ein String bestehend aus einem Identifikator und einer bestimmten Zahl an Parametern. Identifikator und Parameter sind jeweils durch ein Leerzeichen getrennte, zusammenhängende Strings.
ziffer ::= 0|1|2|3|4|5|6|7|8|9
groß ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
klein ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
sonder ::= '-'|'_'
zeichen ::= [*ziffer*|*groß*|*klein*|*sonder*]
zeichenkette ::= *zeichen*[*zeichen*]*

6.2 Nachrichten vom Treiber an das GUI

Nachrichten können

1. in der Initialisierungsphase,
2. in der Spielphase oder
3. in beiden Phasen

zulässig sein.

6.2.1 Nachrichten in der Initialisierungsphase

Eine Übersicht über die verschiedenen Nachrichten der Initialisierungsphase geben die Abbildungen 6.1 und 6.2.

- '**board_name**' + '␣' + *Name*
  Fügt den Spielplan *Name* in die Auswahlliste ein. Der erste Spielplan, der gesetzt

wird, wird als Voreinstellung benutzt.
Name ::= *zeichenkette*

- '**level**' + '␣' + *Stärke*
  Fügt die Spielstärke *Stärke* in die Auswahlliste ein. Die erste Spielstärke, die gesetzt wird, wird als Voreinstellung benutzt.
  Stärke ::= *zeichenkette*

- '**count**' + '␣' + *Anzahl* + '␣' + *Max_Anzahl*
  Setzt die voreingestellte Anzahl an Detektiven auf *Anzahl*. Gleichzeitig wird festgelegt, wieviele Detektive maximal mitspielen können.
  Anzahl ::= *ziffer*
  Max_Anzahl ::= *ziffer*

- '**D_startposition**' + '␣' + *Nr* + '␣' + *Feld-Nr*
  Setzt die voreingestellte Startposition von Detektiv *Nr* auf Feld *Feld-Nr*.
  Nr ::= *ziffer*
  Feld-Nr ::= 0|1 *ziffer ziffer*

- '**X_position**' + '␣' + *Feld-Nr*'
  Zeigt Mister X auf der Startposition *Feld-Nr*.
  Feld-Nr ::= 0|1 *ziffer ziffer*

- '**vertex**' + '␣' + *Feld-Nr*
  Zeigt das Feld *Feld-Nr* als ein zum Spielplan gehöriges Feld an.
  Feld-Nr ::= 0|1 *ziffer ziffer*

- '**block_end**'
  Zeigt der GUI an, daß alle Nachrichten einer Initialisierungsphase gesendet wurden.

6.2.2 Nachrichten in der Spielphase

Einen Überblick über die Nachrichten der Spielphase gibt die Abbildung 6.3.

- '**game_over**' + '␣' + *Gewinner*
  Das Spiel wurde von *Gewinner* gewonnen. *Gewinner* enthält 'X', falls Mister X der Sieger ist, andernfalls die Nummer des Detektivs.
  Gewinner ::= 'X'| *ziffer*

- '**X_move**' + '␣' + *Modus*
  Zeigt der GUI an, daß Mister X ziehen darf. Wird als Parameter *on* übergeben, so kann Mister X ein 2x-Ticket verwenden. Der Parameter *off* wird vom Treiber gesendet, wenn Mister X gerade ein 2x-Ticket verwendet hat und während dieses Zuges – den Regeln folgend – nicht noch einmal ein 2x-Ticket verwenden darf.
  Modus ::= 'on'|'off'

- '**X_show**' + '␣' + *Anzahl*
  Gibt an, daß sich Mister X in *Anzahl* Zügen zeigen muß. Ist *Anzahl* gleich '0', fragt die GUI nach der aktuellen Position von Mister X.
  Anzahl ::= *ziffer*

6.2.3 Nachrichten in der Initialisierungs- und in der Spielphase

- '**D_move**' + '␣' + *Nr* + '␣' + *Feld-Nr* + '␣' + *Ticket*
  Zeigt Detektiv *Nr* auf Feld *Feld-Nr* und das verwendete Ticket *Ticket*.
  Feld-Nr ::= 0|1 *ziffer ziffer*
  Ticket ::= 'taxi'|'bus'|'underground'

- '**D_tickets**' + '␣' + *Nr* + '␣' + *Taxi* + '␣' + *Bus* + '␣' + *U-Bahn*
  Aktualisiert die Anzahl der verschiedenen Tickets für Detektiv *Nr*. Während der Initialisierung bedeutet dies, daß Detektiv *Nr* die angegebene Anzahl an Tickets als Voreinstellung bekommt.
  Nr ::= *ziffer*
  Taxi ::= *ziffer ziffer*
  Bus ::= *ziffer ziffer*
  U-Bahn ::= *ziffer ziffer*

- '**X_tickets**' + '␣' + *Taxi* + '␣' + *Bus* + '␣' + *U-Bahn* + '␣' + *Black* + '␣' + *2x*
  Aktualisiert die Anzahl der verschiedenen Tickets für Mister X. Während der Initialisierung bedeutet dies, daß Mister X die angegebene Anzahl an Tickets als Voreinstellung bekommt.
  Taxi ::= *ziffer ziffer*
  Bus ::= *ziffer ziffer*
  U-Bahn ::= *ziffer ziffer*
  Black ::= *ziffer ziffer*
  2x ::= *ziffer ziffer*

6.3 Nachrichten vom GUI an den Treiber

6.3.1 Nachrichten in der Initialisierungsphase

Einen Überblick über die Nachrichten in der Initialisierungsphase geben die Abbildungen 6.1 und 6.2.

- '**new_game**'
  Benachrichtigt den Treiber darüber, daß der Spieler ein neues Spiel beginnen möchte.

- '**board_name**' + '␣' + *Name*
  Teilt dem Treiber mit, daß der Spieler den Spielplan *Name* ausgewählt hat.
  Name ::= *zeichenkette*

- '**level**' + '␣' + *Stärke*
  Teilt dem Treiber mit, daß der Spieler die Spielstärke *Stärke* ausgewählt hat.
  Stärke ::= *zeichenkette*

- '**count**' + '␣' + *Anzahl* + '␣' + *Max_Anzahl*
  Teilt dem Treiber mit, daß der Spieler *Anzahl* Detektive ausgewählt hat.
  Anzahl ::= *ziffer*
  Max_Anzahl ::= *ziffer*

- '**D_tickets**' + '␣' + *Nr* + '␣' + *Taxi* + '␣' + *Bus* + '␣' + *U-Bahn*
  Teilt dem Treiber mit, daß der Spieler für Detektiv *Nr* als Ticketvorrat *Taxi* Taxitickets, *Bus* Bustickets und *U-Bahn* U-Bahntickets gewählt hat.
  Nr ::= *ziffer*
  Taxi ::= *ziffer ziffer*
  Bus ::= *ziffer ziffer*
  U-Bahn ::= *ziffer ziffer*

- '**X_tickets**' + '␣' + *Taxi* + '␣' + *Bus* + '␣' + *U-Bahn* + '␣' + *Black* + '␣' + *2x*
  Teilt dem Treiber mit, daß der Spieler für Mister X als Ticketvorrat *Taxi* Taxitickets, *Bus* Bustickets, *U-Bahn* U-Bahntickets, *Black* Blacktickets und *2x* 2x-Tickets gewählt hat.
  Taxi ::= *ziffer ziffer*
  Bus ::= *ziffer ziffer*
  U-Bahn ::= *ziffer ziffer*
  Black ::= *ziffer ziffer*
  2x ::= *ziffer ziffer*

- '**D_startposition**' + '␣' + *Nr* + '␣' + *Feld-Nr*
  Setzt die voreingestellte Startposition von Detektiv *Nr* auf Feld *Feld-Nr*.
  Nr ::= *ziffer*
  Feld-Nr ::= 0|1 *ziffer ziffer*

- '**block_end**'
  Zeigt dem Treiber an, daß alle Nachrichten einer Initialisierungsphase gesendet wurden.

- '**init_end**'
  Zeigt dem Treiber an, daß die Initialisierung beendet ist.

6.3.2 Nachrichten in der Spielphase

Einen Überblick über die Nachrichten der Spielphase gibt die Abbildung 6.3.

- '**X_move** *Ticket*'
  Teilt dem Treiber mit, daß Mister X seinen Zug mit dem Ticket *Ticket* gemacht hat.
  Ticket ::= 'taxi'|'bus'|'underground'|'black'|'twice'

- '**game_over**'
  Teilt dem Treiber mit, daß der Spieler das Spiel abgebrochen hat. Diese Nachricht ist nur während der Spielphase zulässig.

6.3.3 Nachrichten in der Initialisierungs- und in der Spielphase

- '**X_position**' + '␣' + *Feld-Nr*
  Teilt dem Treiber mit, daß sich Mister X auf Feld *Feld-Nr* befindet. In der Initialisierungsphase wird hiermit die Startposition gesetzt.
  Feld-Nr ::= 0|1 *ziffer ziffer*

- '**exit**'
  Teilt dem Treiber mit, daß der Spieler das Programm beendet hat.

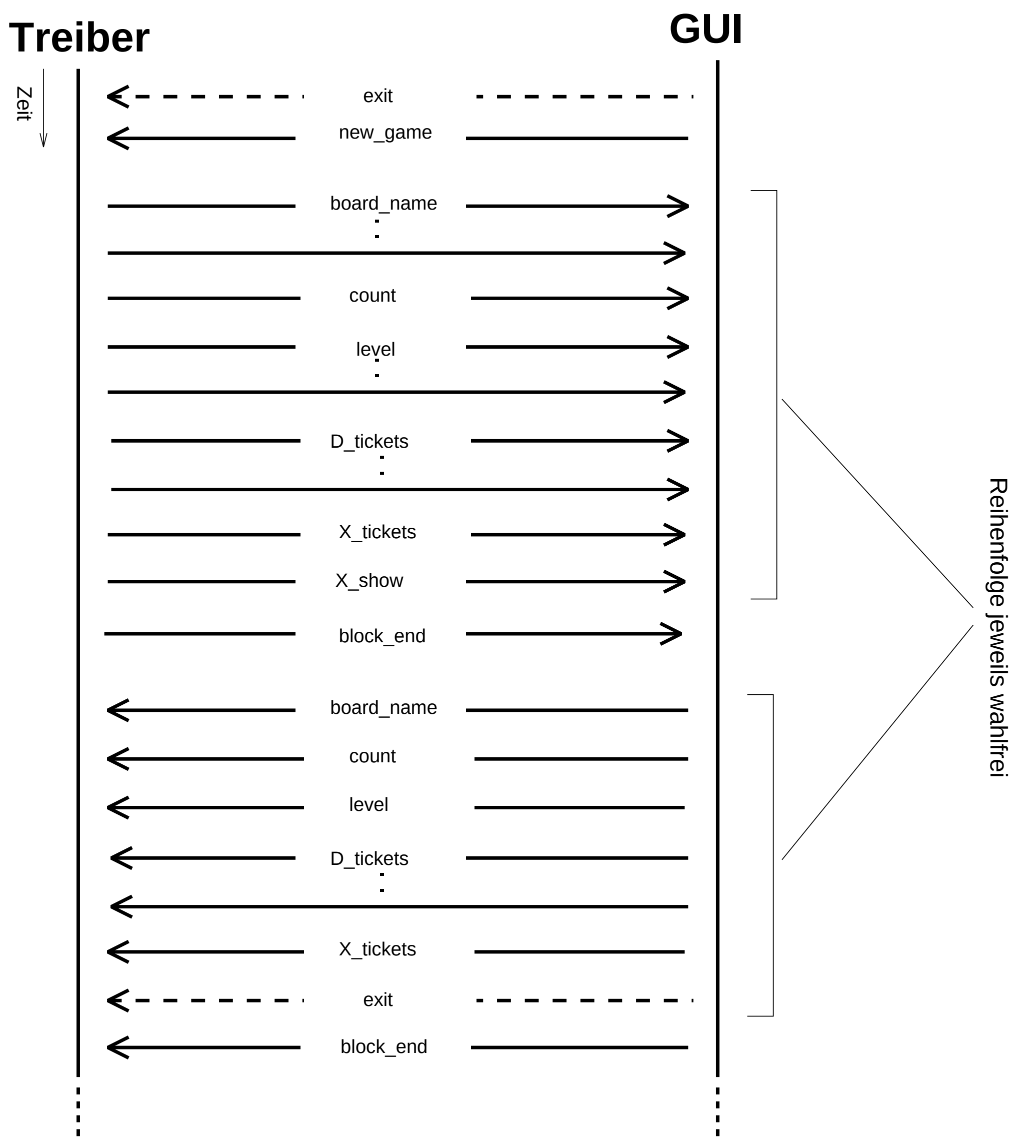


Abbildung 6.1: Protokoll des Austausches von Nachrichten zwischen der GUI und der Treiber-/Regelkomponente (Initialisierungsphase, Teil 1)

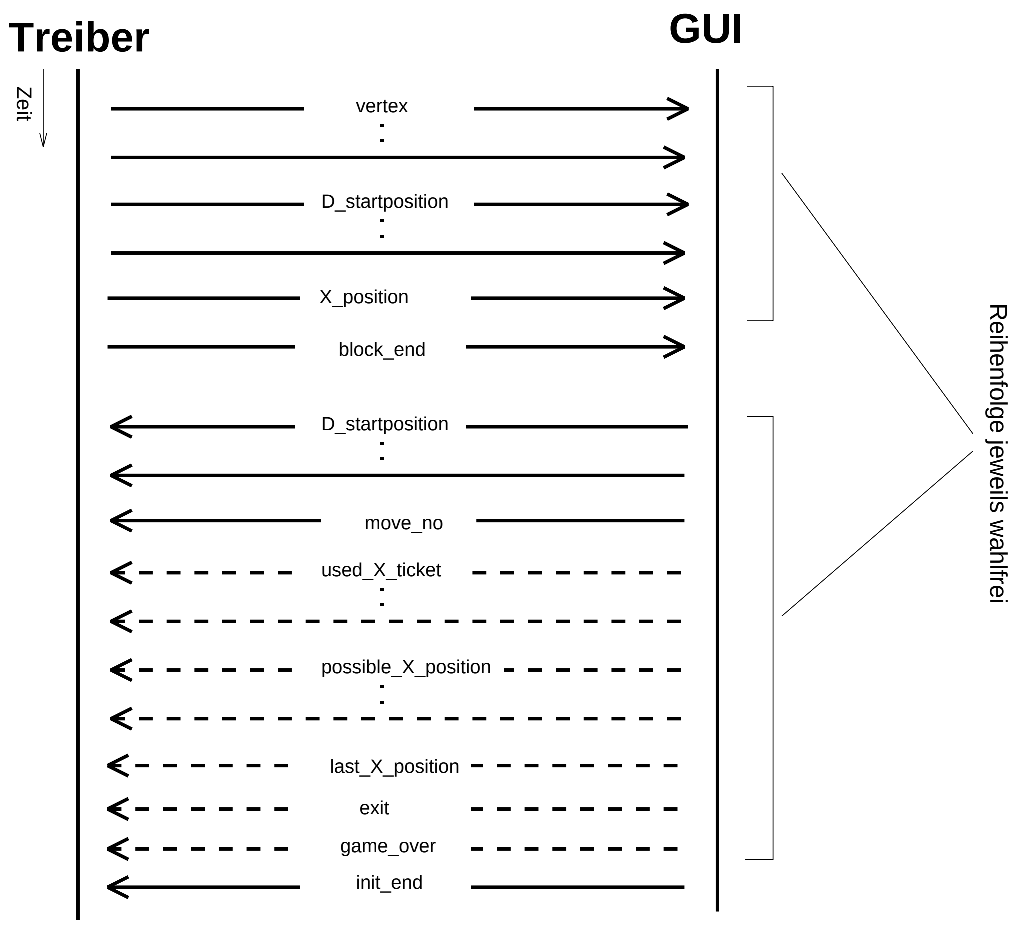


Abbildung 6.2: Protokoll des Austausches von Nachrichten zwischen der GUI und der Treiber-/Regelkomponente (Initialisierungsphase, Teil 2)

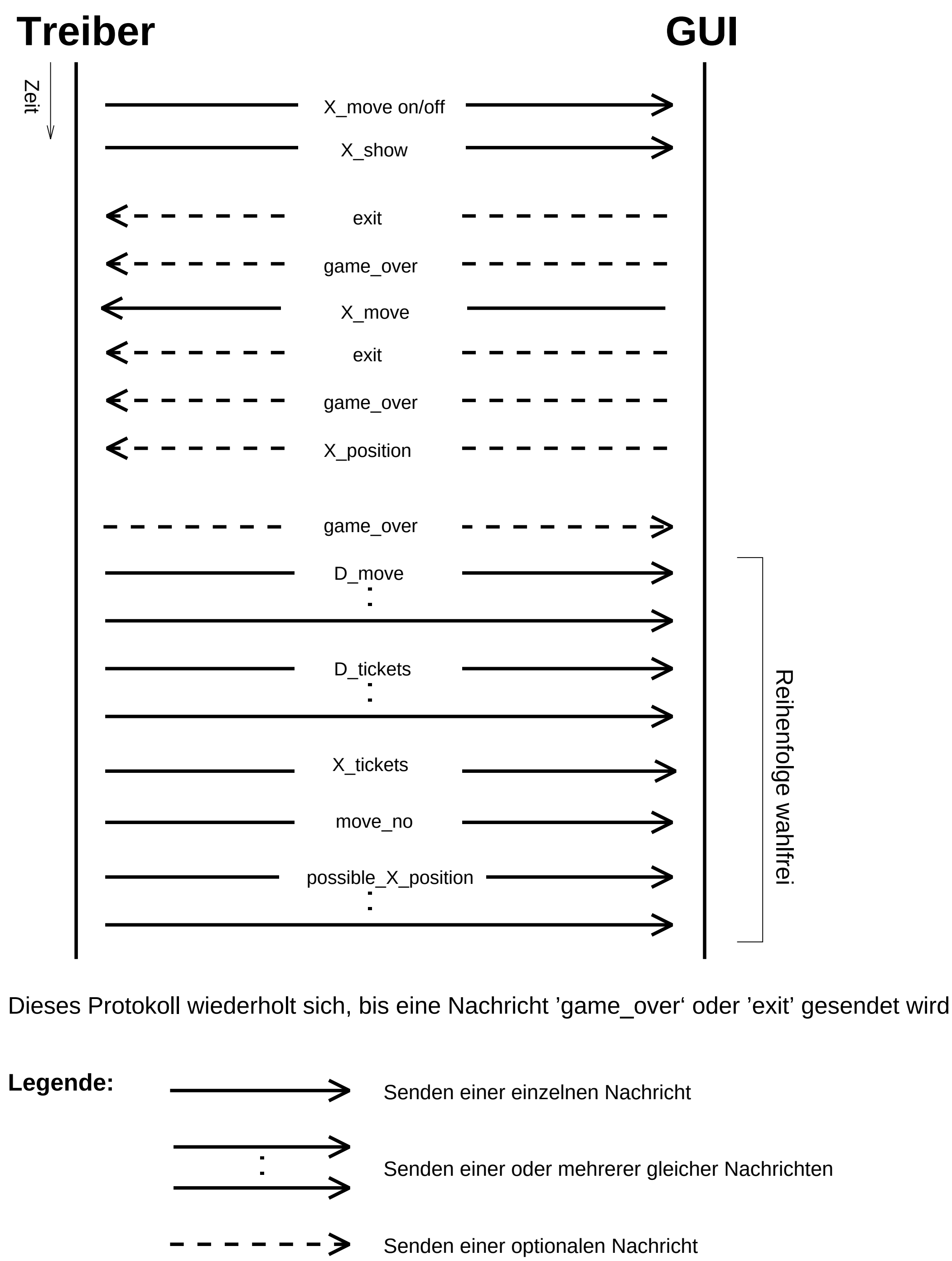


Abbildung 6.3: Protokoll des Austausches von Nachrichten zwischen der GUI und der Treiber-/Regelkomponente (Spielphase)

7. Spezifikation der Treiber- und Regelkomponente

In diesem Kapitel stellen wir die Spezifikation der Treiberkomponente vor. Nach einigen harten Auseinandersetzungen mit den beiden anderen Teilgruppen haben wir uns auf einen Satz von Funktionen geeinigt, die hier in klassicher Form, d.h. mit Namen, Funktionsbeschreibung sowie Ein- und Ausgabewerten beschreiben sind.

simulate_gui

Eingabe: keine
Ausgabe: keine
Beschreibung: Öffnet die Steuerungsdatei und belegt die globale Variable `simulate` mit TRUE.
Fehler: Falls keine Steuerungsdatei vorhanden ist, bleibt `simulate` auf FALSE.

start_gui

Eingabe: keine
Ausgabe: Hat das Starten geklappt, `bool`
Beschreibung: Startet das grafische Benutzerinterface und baut die Kommunikation auf.

init

Eingabe: keine
Ausgabe: keine
Beschreibung: Teilt der GUI die Standardwerte für das Spiel mit und initialisiert die globalen Variablen. Nimmt eventuelle Änderungen der GUI entgegen und paßt die globalen Variablen entsprechend an. Siehe Nachrichtenprotokoll für GUI und Treiber.

start_strategy

Eingabe: keine
Ausgabe: keine
Beschreibung: Erzeugt die Tupelräume für das Interface und für das Blackboard, sowie die Prozesse für die Detektive und stellt die Daten für die Planung in den Tupelraum.

play_the_game

Eingabe: keine
Ausgabe: keine
Beschreibung: Hauptschleife, die den Spielablauf organisiert.

game_over

Eingabe: keine
Ausgabe: keine
Beschreibung: Beendet das Programm.

load_board_names

Eingabe: keine
Ausgabe: Liste der verfügbaren Spielpläne, `tuple`
Beschreibung: Durchsucht das aktuelle Verzeichnis nach Dateien mit Spielplänen und liefert eine Liste der gefundenen Namen zurück.

load_board

Eingabe: keine
Ausgabe: keine
Beschreibung: Lädt den Spielplan in die globale Variable `board` ein. Die globale Variable `start_positions` wird ebenfalls geladen.

store_TS_data

Eingabe: Blackboard-Tupelraum, `atom`
Ausgabe: keine
Beschreibung: Stellt die von der Planung benötigten Daten in den Tupelraum. Außerdem wird in dem sonst nur von den Detektiven benutzten Tupelraum „Blackboard“ die Spielstufe eingetragen.

update_TS_data

Eingabe: Zug von X, `tuple`
Ausgabe: keine
Beschreibung: Aktualisiert die von der Planung benötigten Daten im Tupelraum.

serve_strategy_requests

Eingabe: keine
Ausgabe: Züge der Detektive, `tuple`
Beschreibung: Wartet auf Anfragen der Planungskomponente im Tupelraum und bearbeitet sie solange, bis alle Detektive ihre korrekten Züge bekannt gegeben haben.

wait_for_X

Eingabe: keine
Ausgabe: Von X verwendete(s) Ticket(s), `tuple`
Beschreibung: Wartet auf die Nachricht `X_move` von der GUI und liefert das/die von X verwendete(n) Ticket(s) zurück.

update_D_variables

Eingabe: Züge der Detektive, `tuple`
Ausgabe: keine
Beschreibung: Aktualisiert die globalen Variablen, die die Detektive betreffende Daten speichern.

send_move_to_gui

Eingabe: Züge der Detektive, `tuple`
Ausgabe: keine
Beschreibung: Teilt die geänderten Daten (Parameter und globale Variablen) der GUI mit. (Siehe Nachrichtenprotokoll für GUI und Treiber.)

send_gui_message

Eingabe: Nachricht, `string`
Ausgabe: keine
Beschreibung: Sendet die im Parameter angegebene Nachricht an die GUI.

get_gui_message

Eingabe: keine
Ausgabe: Nachricht, `string`
`""`, falls keine Nachricht anliegt.
Beschreibung: Liefert eine oder keine Nachricht von der GUI. Es ist wichtig, daß die Funktion zurückkehrt, falls keine Nachricht anliegt. Falls `simulate` TRUE ist, wird nicht die Pipe abgefragt, sondern das `simulation_file` ausgelesen. Ist dieses leer, wird das Programm beendet. Mögliche Nachrichten sind: `board_name`, `count`, `level`, `D_tickets`, `X_tickets`, `D_start_position`, `X_move`, `X_show`, `X_position`, `start_position`, `block_end`, `init_end`, `new_game`, `exit`, `game_over`.

X_moves_to_visible

Eingabe: keine
Ausgabe: Anzahl der Züge, `integer`
Beschreibung: Berechnet die Anzahl der Züge, nach der sich X zeigen muß.

possible_X_positions

Eingabe: keine
Ausgabe: Menge der Knoten, an denen sich X aur Zeit aufhalten kann, set
Beschreibung: Bestimmt alle Möglichen Aufenthaltsorte von Mister X. Solange sich Mister X noch nicht gezeigt hat, sind dies alle Orte, an denen keine Detektive stehen. Falls er sich soeben gezeigt hat, ist es eine einelementige Menge mit diesem Knoten.

generate_X_position

Eingabe: keine
Ausgabe: Position, `integer`
Beschreibung: Erzeugt eine zufällige Position für X.

generate_D_positions

Eingabe: keine
Ausgabe: Positionen, `set`
Beschreibung: Erzeugt zufällige Positionen für alle Detektive und legt sie in der globalen Variablen `D_positions` ab.

possible_moves

Eingabe: Position, `int`
Ausgabe: Mögliche Positionen mit entsprechenden Tickets, `set`
Beschreibung: Berechnet für die angegebene Position alle möglichen Orte, die mit einem Zug mit dem verwendetem Ticket zu erreichen sind.

shortest_path

Eingabe: Startposition, `integer`
Zielposition, `integer`
Ticketvorrat, `tuple`

Ausgabe: Folge von Knoten mit entsprechendem Ticket, `tuple`
Beschreibung: Berechnet unter Berücksichtigung des Ticketvorrats den kürzesten Weg von der Startposition zur Zielposition.

D_moves_correct

Eingabe: Züge der Detektive, `tuple`
Ausgabe: Züge zulässig, `boolean`
Beschreibung: Überprüft die Zulässigkeit der Züge aller Detektive.

8. Spezifikation der Schnittstelle zwischen Treiber- und Planungskomponente

8.1 Vorbemerkungen

Die Schnittstelle läuft über den Linda–Tupelraum. Die Tupel sind in die zwei Bereiche Daten und Anfragen geteilt:

Datum Diese Tupel werden von der Treiberkomponente in dem Tupelraum eingetragen und nur von dieser modifiziert. Die Planungskomponente liest die Werte nur aus (dies darf nur durch lesendes `meet` geschehen).

Anfrage Diese Tupel bauen eine Client–Server Kommunikation auf. Dazu wird als zweiter Parameter eine Identifikation mitgeliefert, die bei der Antwort als erster Teil des Tupels steht:

Beispiel:

```
deposit ["possible_move", ID, ...] at TS end deposit;
fetch [ID, ...] at TS end fetch;
```

Die Grobarchitektur der Kommunikation zwischen Detektiven (Planungskomponente) und Treiberkomponente ist in Abb. 8.1 festgehalten.

Die für die Planung benötigten Funktionalitäten sind nach Aufgaben in die verschiedenen Bereiche *Schnittstelle zum Spieltreiber*, *Schnittstelle zur Regelkomponente* sowie *Schnittstelle zur Initialisierungs– und Utility–Komponente* geteilt worden.

Die zum Verstehen der Schnittstellenspezifikationen benötigte Erläuterungen zur Backus-Naur-Form(BNF) erfolgen in Abschnitt 8.5

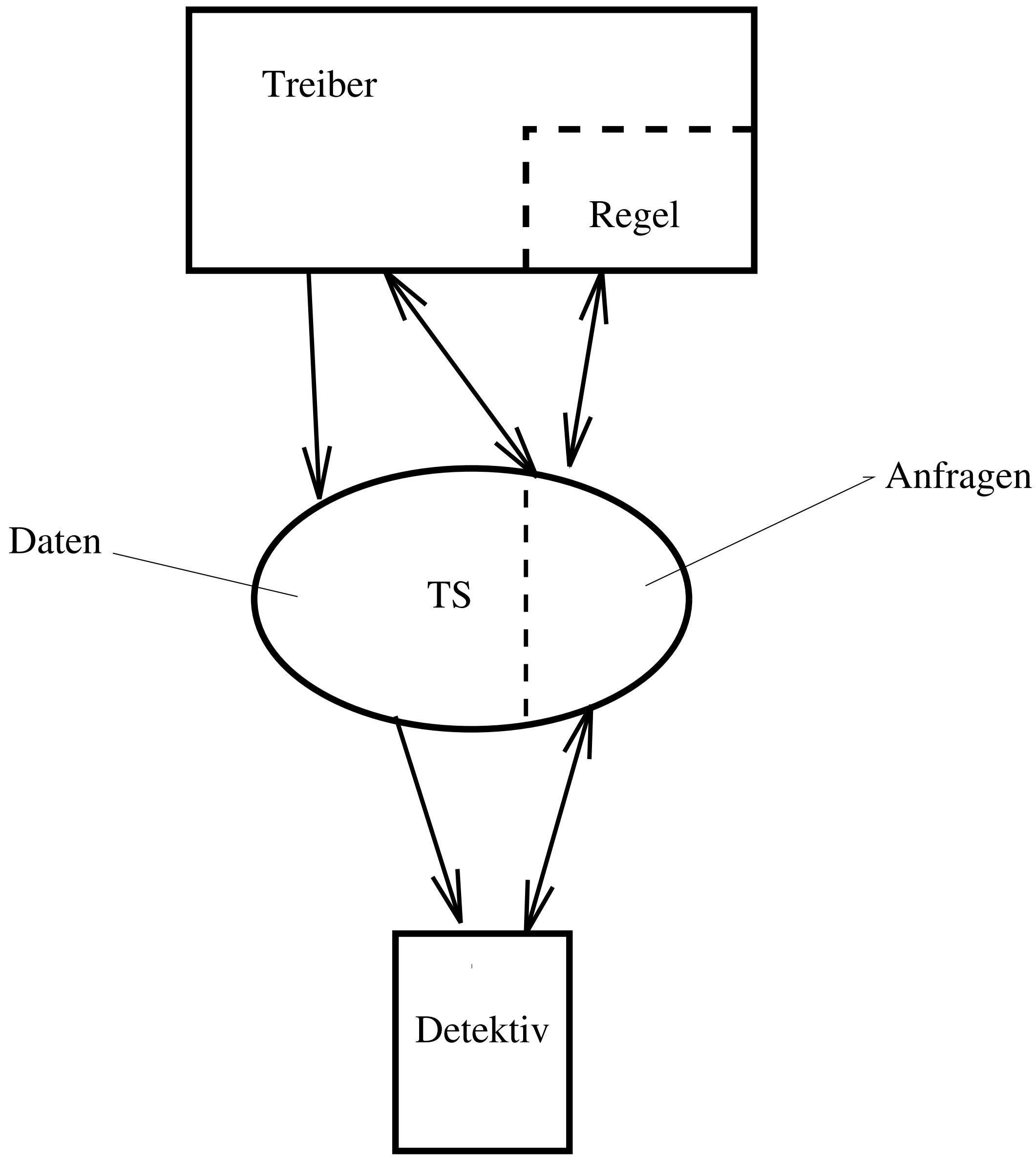


Abbildung 8.1: Grobarchitektur der Kommunikation zwischen Treiber und Planung.

8.2 Schnittstelle zum Spieltreiber

Nun erfolgt die Spezifikation der Tupel, die die Informationen über den momentanen Spielstand enthalten.

Datum: last_public_X_position

Syntax: ⟨*last_public_X_position*⟩ ::= `["last_public_X_position",` (*knotennummer* | `om`)`]`
Beschreibung: Enthält die Knotennummer, an der sich Mister X zuletzt gezeigt hat.
Ausgabe: Eine Knotennummer von 1 . . . 199. Wenn sich Mister X noch nicht gezeigt hat, wird `om` zurückgeliefert.

Datum: used_X_tickets

Syntax: ⟨*used_X_tickets*⟩ ::= `["used_X_tickets",` *ticketfolge*`]`
Beschreibung: Enthält die von Mister X seit dem letzten Auftauchen benutzten Tickets zurück.
Ausgabe: Tupel mit den möglichen Tickets. Das Tupel ist leer, falls Mister X gerade aufgetaucht ist.

Datum: number_of_X_moves

Syntax: ⟨*number_of_X_moves*⟩ ::= `["number_of_X_moves",` *int*`]`
Beschreibung: Enthält die Gesamtzahl der Züge von Mister X˙
Anzahl der Züge. Vor dem ersten Zug `0`.

Datum: X_moves_to_visible

Ausgabe:

Syntax: ⟨*X_moves_to_visible*⟩ ::= `["X_moves_to_visible",` (*int* | `om`)`]`
Beschreibung: Enthält die Anzahl der Züge von Mister X bis er sich wieder zeigen muß. Wenn sich Mister X gerade zeigt, ist dies die Anzahl der Züge, bis er sich zum nächsten Mal zeigt (insbesondere nicht 0!).
Ausgabe: Anzahl der Züge, `om` in der letzten Runde.

Datum: number_of_D_moves

Syntax: ⟨*number_of_D_moves*⟩ ::= `["number_of_D_moves"`, (*int* | `om`)`]`
Beschreibung: Enthält die Anzahl der Züge, die die Detektive gezogen haben, seit sich Mister X zuletzt gezeigt hat.
Ausgabe: Anzahl der Züge, `om` zu Beginn.

Datum: ticket_count

Syntax: ⟨*ticket_count*⟩ ::= `["ticket_count"`, `[`*ticketvorrat*`]]`
Beschreibung: Enthält für jede Figur die Anzahl der verfügbaren Tickets jeder Sorte zurück.
Eingabe: Mister X oder die Nummer des Detektives als zweite Komponente des Tupels
Ausgabe: Menge von Tupeln mit Ticketbezeichnung und Anzahl als dritte Komponente des Linda–Tupels. `"twice"` wird nur bei Mister X angegeben.

Datum: possible_X_positions

Syntax: ⟨*possible_X_positions*⟩ ::= `["possible_X_positions"`, '{'*pint*{, *pint*}'}'`]`
Beschreibung: Enthält *alle* möglichen Aufenthaltsorte von Mister X. D.h., wenn er sich gerade zeigt, ist in der Menge nur ein Ort enthalten. Anschließend sind es die Blätter des Zuggraphen für die Tickets von Mister X, ab diesem Aufenthaltsort. Beim ersten Zug sind es jeweils alle Orte, bis auf diejenigen, die von den Detektiven eingenommen wurden.
Ausgabe: Menge von Positionen

Datum: all_used_X_tickets

Syntax: ⟨*all_used_X_tickets*⟩ ::= `["all_used_X_tickets"`, (`[]` | `[` *ticket* , *ticket*`]`)`]`
Beschreibung: Enthält alle bisher von Mister X benutzten Tickets.
Ausgabe: Tupel von Tickets ohne `"twice"`.

Datum: current_X_move

Syntax: $\langle$*current_X_move*$\rangle$ ::= `["current_X_move"`, (`[`*ticket*`]` | `[` *ticket*, *ticket*`]`)`]`
Beschreibung: Gibt den aktuellen (letzten) Zug von Mister X an. Für jeden der Detektive wird ein Tupel in den Tupelraum geschrieben.
Ausgabe: Tupel von Tickets: `[`T`]` oder `[`T_1, T_2`]` bei einem Doppelzug.

Anfrage: D_position

Syntax: $\langle$*D_position*$\rangle$ ::= `["D_position"`, *id*, *detektiv*, *knotennummer*, *dticket*`]`
$\langle$*Antworttupel*$\rangle$ ::= `[`*id*, *wwert*`]`
Beschreibung: Gibt die Züge der Detektive in dieser Runde bekannt. Jeder Detektiv schreibt eine eigene Anfrage in den Tupelraum, die von der Treiberkomponente auf Korrektheit überprüft wird.
Eingabe: D_i, Position, Ticket
Ausgabe: `true` oder `false`, je nachdem, ob der Zug gültig war oder nicht.

8.3 Schnittstelle zur Regelkomponente (Zuggenerator)

Im folgenden werden alle Tupel beschrieben, die die Informationen enthalten, die von der Planungskomponente zur Berechnung der Strategien benötigt werden.

Anfrage: possible_moves

Syntax: $\langle$*possible_moves*$\rangle$ ::= `["possible_moves"`, *id*, *knotennummer*`]`
$\langle$*Antworttupel*$\rangle$ ::= `[`id, ('{'*zuege*'}' |`[]`)`]`
Beschreibung: Bestimmt für eine angebene Position alle möglichen Orte, die mit einem Zug zu erreichen sind. Die Funktion ist unabhängig vom aktuellen Spielstand und von der potentiellen Figur, von der es aufgerufen wird. D.h., daß auch Schwarzfahrten in Betracht gezogen werden, wenn damit die Themse befahren werden kann.
Eingabe: Position (1 . . . 199)
Ausgabe: Menge von Tupeln der Gestalt `[`Position, Ticket`]` . Wenn die Eingabe-Position fehlerhaft ist, wird `om` zurückgeliefert.

Anfrage: shortest_path

Syntax: ⟨*shortest_path*⟩ ::= `["shortest_path"`, *id*, *knotennummer*, *knotennummer*, *dticketvorrat*`]`

⟨*Antworttupel*⟩ ::= `[`*id*, (`[`*zuege*`]` | `[]`)`]`

Beschreibung: Bestimmt den kürzesten Pfad zwischen zwei Orten für einen angegebenen Ticketvorrat.

Eingabe: Start- und Zielort, Liste der Tickets

Ausgabe: Zugfolge als Tupel von Tupeln der Art `[`Position, Ticket`]`. Wenn die Orte mit den Ticktes nicht zu erreichen, wird `om` geliefert, wenn Start- und Zielort identisch sind, wird `[]` geliefert.

Anfrage: path_sum

Syntax: ⟨*path_sum*⟩ ::= `["path_sum"`, *id*, *knotennummer*, '{'*knotenfolge*'}'`]`

⟨*Antworttupel*⟩ ::= `[`id, *int*`]`

Beschreibung: Bestimmt zu jedem in der Menge angegebenen Knoten den Abstand zum Knoten mit der Nummer *knotennummer* (basierend auf den Taxiverbindungen) und berechnet die Summe dieser Abstände.

Eingabe: Startort und Menge der Zielorte

Ausgabe: Die berechnete Summe der Abstände. Diese Summe ist 0, wenn eine leere Menge übergeben wurde oder wenn die Menge nur mit dem Startknoten identische Knoten enthält.

8.4 Schnittstelle zu Initialisierungs– und Utitlity–Komponente

Die für den Spielstart und der benutzerspezifischen Einstellungen benötigten Informationen werden in den folgenden Tupeln übergeben:

Datum: board

Syntax: ⟨*board*⟩ ::= `["board"`, *brettrepraesentation*`]`

Beschreibung: Gibt die Brettrepräsentation zurück.

Ausgabe: Brettrepräsentation

Datum: level

Syntax: $\langle level \rangle$ ::= `["level",` *int*`]`
Beschreibung: Enthält den vom Benutzer eingestellten Schwierigkeitsgrad
Ausgabe: Schwierigkeitsgrad

Datum: D_count

Syntax: $\langle D\_count \rangle$ ::= `["D_count",` *max_detektive*`]`
Beschreibung: Enthält die Anzahl der am Spiel beteiligten Detektive
Ausgabe: Anzahl der Detektive

Datum: intial_D_position

Syntax: $\langle intial\_D\_position \rangle$ ::= `["intial_D_position",` *detektiv*, *knotennummer*`]`
Beschreibung: Enthält die Positionen der Detektive zu Beginn des Spiels. Für jeden Detektiv wird ein Tupel abgelegt.
Ausgabe: Detektiv und Position

8.5 Spezifikation in BNF

8.5.1 Erläuterungen

Geschweifte Klammern {} kennzeichnen optionale Werte, der senkrechte Strich | Alternativen, die innerhalb von runden Klammern () stehen. Strings werden in Hochkommata " gesetzt, alles andere kennzeichnet Terminale.

8.5.2 Deklarationen

$\langle \mathit{ticketfolge} \rangle ::= \texttt{[om} \mid \mathit{ticket}\ \{,\ \mathit{ticketfolge}\}\texttt{]}$

$\langle \mathit{ticketvorrat} \rangle ::= \mathit{vorratsliste}\ \{,\ \mathit{ticketvorrat}\}$

$\langle \mathit{vorratsliste} \rangle ::= \texttt{[}\mathit{ticket} \mid \texttt{"twice"},\ \mathit{int}\ \texttt{]}$

$\langle \mathit{dticketvorrat} \rangle ::= \mathit{dvorratsliste}\ \{,\ \mathit{dticketvorrat}\}$

$\langle \mathit{dvorratsliste} \rangle ::= \texttt{[}\mathit{dticket},\ \mathit{int}\ \texttt{]}$

$\langle \mathit{z\ddot{u}ge} \rangle ::= \texttt{[}\mathit{knotennummer},\ \mathit{ticket}\texttt{]}\quad \{,\ \text{züge}\}$

$\langle \mathit{figur} \rangle ::= \texttt{"X"} \mid \mathit{detektiv}$

$\langle \mathit{ticket} \rangle ::= \mathit{dticket} \mid \texttt{"black"}$

$\langle \mathit{dticket} \rangle ::= \texttt{"taxi"} \mid \texttt{"bus"} \mid \texttt{"underground"}$

$\langle \mathit{knotennummer} \rangle ::= \texttt{1} \mid \ldots \mid \texttt{199}$

$\langle \mathit{wwert} \rangle ::= \texttt{true} \mid \texttt{false}$

$\langle \mathit{detektiv} \rangle ::= \texttt{1} \mid \ldots \mid \mathit{max\_detektive}$

$\langle \mathit{int} \rangle ::= \texttt{0} \mid \mathit{pint}$

$\langle \mathit{pint} \rangle ::= \texttt{1} \mid \ldots \mid \mathit{maxint}$

$\langle \mathit{id} \rangle ::=$ eindeutige Identifikation

$\langle \mathit{maxint} \rangle ::=$ größter Integer-Wert

$\langle \mathit{max\_detektive} \rangle ::=$ Anzahl der mitspielenden Detektive

$\langle \mathit{knotenfolge} \rangle ::= \mathit{knotennummer}\ \{,\ \mathit{knotennummer}\}$

$\langle \mathit{adjazensliste} \rangle ::= \texttt{[}\mathit{knotenfolge}\texttt{]}\quad \{,\ \mathit{adjazensliste}\}$

Kommentar: Der 1. Knoten der jeweiligen Knotenfolge gibt den Startknoten an, von wo aus die folgenden Knoten der Folge erreichbar sind.

$\langle \mathit{streckenliste} \rangle ::= \texttt{[}\mathit{dticket},\ \mathit{adjazensliste}\texttt{]}$

$\langle \mathit{brettrepr\ddot{a}sentation} \rangle ::= \texttt{[}\mathit{streckenliste}\ \{,\ \mathit{streckenliste}\}\texttt{]}$

9. Spezifikation der Planungskomponente

9.1 Einleitung

Das Planungsmodul läßt sich in die Komponenten Detektiv-Prozeß, Kommunikation der Detektive untereinander und die Kommunikation mit dem Treiber unterteilen.

9.2 Grundlagen der Planungsalgorithmen

Der Algorithmus sollte die aktuelle Spielphase berücksichtigen; insbesondere ist eine Zweiteilung erforderlich, und zwar bis zum ersten Auftauchen von Mister X und der darauf folgenden Spielphase bis Mister X gefangen wird oder Mister X gewonnen hat.

Für die erste Phase bietet es sich an, daß die Detektive beim ersten Auftauchen von Mister X auf Orten stehen, von denen aus sie möglichst schnell möglichst nah zu Mister X gelangen können (sie sollten z.B. auf U–Bahn–Stationen oder Position 157 stehen).

In der zweiten Spielphase sollten mehrere Faktoren in die Planung einfließen, und zwar:

Abstände zwischen Detektiven und Mister X: Abstände zwischen zwei verschiedenen Orten A und B sind durch die Zahl der Spielzüge gekennzeichnet, die man benötigt, um mit einem gegebenen Ticketvorrat von A nach B zu kommen.

Aufenthaltsmöglichkeiten von Mister X: Mister X hält sich an einem für ihn günstigen Ort auf, d.h., daß er versuchen wird, sich

- nicht einkreisen zu lassen
- viele Fluchtmöglichkeiten zu beschaffen
- an einen Ort zu begeben, an dem er nicht gefangen werden kann (dies hängt aber auch von seiner Dreistigkeit ab!)

Ticketvorrat: Jeder Detektiv muß darauf achten, daß die Tickets möglichst gleichmäßig verbraucht werden. Eventuell muß er deshalb bei gleichwertigen Zielen das Ziel anwählen, daß mit dem Ticket zu erreichen ist, von dem der größere Vorrat vorliegt. Nach Möglichkeit braucht er keine Ticketsorte vorzeitig auf.

Pläne anderer Detektive: Die Detektive benötigen zur Abstimmung ihrer eigenen Pläne die Zugvorschläge der anderen (siehe Abschnitt 9.4.1, S. 102). Die Detektive müssen also ihre aktuellen Vorschläge und Bewertungen veröffentlichen. Dies geschieht am besten über eine Blackboard–Struktur in einem Linda–Tupelraum.

Feldbewertung: Alle Felder auf dem Spielfeld haben sicherlich unterschiedliche Wertigkeiten. Diese ist abhängig von

- der Lage des Feldes (Randfelder sind schlechter als Felder im Inneren)
- der Verkehrsanbindung in Anzahl und Ticketsorte

Beachtung der Mister X–Orte: Die möglichen Orte von Mister X sollten von den verschiedenen Detektiven unterschiedlich gewichtet werden. Damit ist es je nach Interpretation der Gewichtung möglich, daß sich die Detektive auf einen Ort für Mister X einigen oder möglichst viele Orte abdecken und es zu sehr wenigen Überschneidungen der Aufgabenbereiche der Detektive kommt.

9.3 Modellierung des Detektiv-Prozesses

Die allgemeine Struktur eines Detektiv–Prozesses ist — unabhängig von der verwendeten Strategie — in Abbildung 9.1, Seite 99, dargestellt. In Abbildung 9.2, Seite 100, wird die Strategie in der Anfangsphase, bevor Mister X zum ersten Mal aufgetaucht ist, dargestellt.

Initialisierung		
∀ Züge von Mister X		
	Womit hat Mister X gezogen	
		Berechne Zug (Planungsalgorithmen)
		Veröffentliche Zug
	BIS keine globale Verbesserung möglich	
	Zug durchführen	

Abbildung 9.1: Ablauf des Detektiv–Prozesses

9.3.1 Ein Plangenerator

Diese Strategie, *Schwachsinn* genannt, wählt zufällig einen Zielknoten aus allen möglichen Zielknoten aus (siehe Abb. 9.3, Seite 100).

WENN	Mister X noch nie aufgetaucht ist
DANN	Gehe in die Richtung eines Ortes, der schnelle Verkehrsanbindungen hat.
SONST	Führe spezifische Planung aus

Abbildung 9.2: Ablauf der Planung

∀ Iterationen der Zugberechnung	
	Ermittele alle möglichen Zielorte (bzgl. des Ticketvorrates)
	Wähle zufällig einen Ort aus
	WENN Konflikte vorliegen
	DANN Wähle einen anderen Ort aus

Abbildung 9.3: Ein Plangenerator

9.3.2 Ein trivialer Planungsalgorithmus

Ein trivialer Planungsalgorithmus, *Kurzsichtig* genannt, wählt zufällig einen Ort, der nur durch die statische Bewertung des Brettes und der Wahrscheinlichkeit, daß Mister X sich an diesem Ort aufhält, beeinflußt wird (siehe Abb. 9.4, Seite 100).

∀ Iterationen der Zugberechnung	
	Ermittele alle möglichen Zielorte (bzgl. des Ticketvorrates)
	Gewichte die Orte mit den Feldbewertungen
	Wähle zufällig einen Ort aus denen aus, die die maximale Gewichtung haben
	WENN Konflikte vorliegen
	DANN Wähle einen anderen Ort aus

Abbildung 9.4: Trivialer Planungsalgorithmus

9.3.3 Distanz

Die Strategie *Distanz* beruht auf der Berechnung des kürzesten Weges zu den möglichen Positionen von Mister XİIn der Abbildung 9.5 auf Seite 101 wird der Ablauf dieses Algorithmus' veranschaulicht.

- ∀ Iterationen der Zugberechnung
  - Besorge aktuelle Pläne und Informationen
  - Mögliche Mister X–Orte berechnen
  - Berechne Abstände
  - Berechne bestes Ziel
  - Konflikte lösen

Abbildung 9.5: Distanz und Distanz–Summe

9.3.4 Distanz–Summe

Diese Strategie benutzt als Gewichtung des Zielknotens die Summe der Abstände zu allen möglichen Positionen von Mister X. Der Ablauf ist äquivalent zu Abbildung 9.5 auf Seite 101, wobei die Anweisung *Berechne bestes Ziel* die unterschiedliche Gewichtungsfunktion kapselt.

9.3.5 Der Referenz–Algorithmus

Diese Strategie stellt eine Adaption des Algorithmus von Becker und Zell [BZ90] dar. Er basiert auf Abstandsberechnungen zwischen den Detektiven und Mister X (siehe Abb. 9.6, Seite 101).

- ∀ Iterationen der Zugberechnung
  - Besorge aktuelle Pläne und Informationen
  - Mögliche Mister X–Orte berechnen
  - ∀ eigene mögliche Zielorte
    - Berechne Mister X–Zukunftsziele
    - ∀ eigenen möglichen Zukuntsziele
      - Bewerte Zukunftsziel
    - Berechne bestes eigenes Zukunftsziel
    - Konflikte lösen
  - Berechne Abstände
  - Berechne bestes Ziel
  - Konflikte lösen

Abbildung 9.6: Referenz–Planungsalgorithmus

9.3.6 Mix

Diese Strategie wählt zufällig für jeden Detektiv und jeden Zug nach dem ersten Auftauchen von Mister X eine Strategie aus den letzten drei genannten aus.

9.4 Kommunikation der Detektive

Die Detektive müssen einen gemeinsamen Plan zur Problemlösung erstellen. Dazu müssen sie untereinander Informationen austauschen, um die Daten der anderen Detektive in ihre eigenen Pläne einzubeziehen.

9.4.1 Abstimmung und Konfliktlösung

Beim Informationsaustausch können grundsätzlich zwei Situationen entstehen. Zum ersten müssen die Detektive sich abstimmen, damit sie kooperativ planen. Zum anderen können Konflikte zwischen den Zügen der Detektive entstehen, weil sie auf den gleichen Ort ziehen wollen.

Als geeignete Strategie für diese Problematik hat sich das folgende Vorgehen im Spiel zwischen Menschen bewährt:

1. Die Pläne der Detektive enthalten jeweils einen gewünschten Zielort und eine Bewertung dieses Zuges. Zusätzlich wird eine Verlustwertung angegeben, die beschreibt, wie groß der Nachteil für den Detektiv ist, falls dieser Zug nicht durchgeführt werden kann.

2. Falls es zu einem Konflikt kommt, wird der Zug bevorzugt, der die höhere Bewertung hat. Falls diese für die Züge gleich ist, wird der Zug gewählt, der den maximalen Verlustwert hat, also für denjenigen, auf den man am wenigsten verzichten kann. Liegt dann immer noch Gleichheit vor, wird zufällig einer der beiden Züge ausgewählt.

3. Die iterative Berechnung des Gesamtplans sollte in endlicher Zeit terminieren. Deshalb muß jeder Detektiv einen Zug dann veröffentlichen, wenn sich die Gesamtwertung seit der letzten Iteration nicht mehr verbessert hat. Initial ist eine Gesamtwertung von 0 eingestellt.

9.4.2 Tupelraum als Blackboard für Daten

In dem Blackboard stehen Daten, die für die Planung eines jeden Detektivs von Belang sind. Ein Beispiel für ein Datum des Blackboards und Operationen darauf ist in Abb. 9.7 auf Seite 103 zu finden. Das Blackboard dient gleichsam als Datenbank und Synchronisator für die Detektive. Um die Daten auffinden zu können, sind sie eindeutig spezifiziert und strukturiert. Dennoch müssen nicht alle Daten auch von allen Planungsalgorithmen verwendet werden; somit wird verteiltes Planen unterstützt.

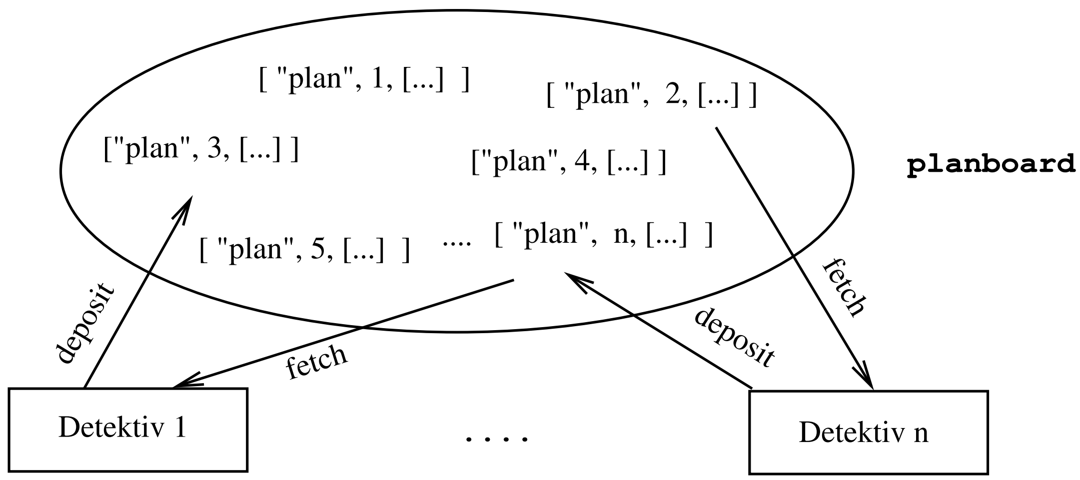


Abbildung 9.7: Plandaten im Blackboard

Die Datensätze im Blackboard sind:

Datum: Calculation_Ready

Syntax: ⟨*Calculation_Ready*⟩ ::= `["calculation_ready"`, *detektiv*`]`

Beschreibung: Dieses Flag wird im Tupelraum eingetragen, wenn der Detektiv seine Plan–Berechnung abgeschlossen hat. Die anderen Detektive dürfen dann seinen Plan nicht mehr modifizieren. Diese Tupel müssen von den Detektiven jeweils zu Beginn eines Zuges wieder aus dem Tupelraum entfernt werden.

Datum: Plan

Syntax: ⟨*Plan*⟩ ::= `["Plan"`, *detektiv*, *Zielort*, *FolgeZiel*, *Wert*, *Verlust*`]`

⟨*Zielort*⟩ ::= *Knotennummer* | `0`

⟨*FolgeZiel*⟩ ::= *Knotennummer* | `0` | `om`

⟨*Wert*⟩ ::= `0` | ... | `100`

⟨*Verlust*⟩ ::= *Wert*

Beschreibung: Dieses Tupel ist eine Planbewertung eines Detektivs. Er gibt an, wohin der Detektiv gehen möchte (*ZielOrt*), wohin er von dort aus gehen will (*FolgeZiel*), welchen Wert der Zielort für ihn hat und wie groß der Verlust für ihn ist, falls er den Zug nicht ausführt.
Ausgabe: *ZielOrt* und *FolgeZiel* sind *0*, wenn der Detektiv keinen Zug machen kann. Wenn der Detektiv keinen Folgezielort berechnet hat, trägt er den Wert `om` für *FolgeZiel* ein.

Datum: Beachtung

Syntax: $\langle Beachtung \rangle$ ::= `["Beachtung",` *detektiv*, *Knotenummer*, *Wert*`]`
Beschreibung: Hier trägt der Detektiv ein, welche Beachtung er diesem potentiellen Aufenthaltsort (*Knotennummer*) von Mister X schenkt.

9.5 Interne Schnittstellen

Die Funktionen der internen Schnittstellen werden genauso wie die verschiedenen Strategien in ProSet–Modulen implementiert, da nur damit die einzelnen Strategie–Module die Möglichkeit besitzen, auf diese Funktionen zuzugreifen.

Read_Plans

Syntax: `Read_Plans (rd Detective)`
Beschreibung: Liest die besten Pläne der anderen Detektive aus dem Blackboard–Tupelraum.
Eingabe: `Detective` der aktuelle Detektiv
Ausgabe: Die jeweils besten Pläne der anderen Detektive.

Resolve_Conflicts

Syntax: `Resolve_Conflicts (rd Plans, rd Detective)`
Beschreibung: Die berechneten Pläne des Detektives werden mit denen der anderen kooperativ in Übereinstimmung gebracht. Dabei werden Konflikte mit Hilfe der oben angeführten Regeln beseitigt.
Eingabe: `Plans` die Pläne des Detektives
`Detective` die Nummer des Detektivs

Show_Move

Syntax: `Show_Move (rd Plans, rd Detective)`
Beschreibung: Löscht die alten Pläne des Detektivs und ersetzt sie durch die neu berechneten.
Eingabe: `Plans` neu berechnete Pläne. Die Pläne sind ein Tupel aus Tupeln des Datums Plan ohne die Komponente `"Plan"` und Detektiv-Nummer.
`Detective` die Nummer des Detektivs

Best_Move_Found

Syntax: `Best_Move_Found ()`
Beschreibung: überprüft, ob der beste Zug gemäß unserer Definition gefunden wurde.

9.6 Exportierte Funktionen der Strategie-Module

Die verschiedenen Strategien werden durch Instanzen von ProSet–Modulen realisiert, die alle eine gleiche Schnittstelle aufweisen. Die folgenden Funktionen müssen mit dieser Semantik realisiert werden.

Calculate_Move

Syntax: `Calculate_Move ()`
Beschreibung: Berechnet den nächsten Zug–Vorschlag des Detektivs

Make_Move

Syntax: `Make_Move`
Beschreibung: Teilt den berechneten Zug der Treiberkomponente mit

9.7 Externe Schnittstellen zum Treiber

In der Planungskomponente werden bis auf den Start der Detektiv–Prozesse keine Funktionen nach außen sichtbar gemacht. Die Kommunikation mit der Treiberkomponente findet über einen Linda–Tupelraum statt. Die Definition der dazu benötigten Tupel ist in dem Abschnitt 9, *Spezifikation der Planungskomponente*, zu finden.

Start_Detective

Syntax:

```
Start_Detective (rd Number,
                 rd Communication_TS,
                 rd Blackboad_TS)
```

Beschreibung: Startet einen Detektivprozeß.

Eingabe: `Number` Nummer des Detektivs

`Communication_TS` Bezeichner des Tupelraum zur Kommunikation mit der Treiberkomponente

`Blackboard_TS` Bezeichner des Tupelraum zur Kommunikation zwischen den Detektiv–Prozessen.

Teil III

Implementation

10. Eingesetzte Werkzeuge

Bei der Implementierung des Scotland–Programmes haben wir die folgenden Werkzeuge benutzt:

- ProSet–Compiler `pst` in diversen Versionen (immer besser geworden!) zur Steuerung des Übersetzungsprozesses von ProSet zum ausführbaren Programm.
- Tcl/tk zur Implementierung der Oberfläche.
- gcc 2.5.8 zur Implementierung der IPC–Library.
- noweb 2.5a mit einem Pretty-Printer in Icon V.8, um Literate Programming zu realisieren. Dabei wird der ProSet–Code in eine etwas besser lesbare und mathematisch inspirierte Notation umgesetzt.
- LaTeX2e im Compatibility Mode zum Erzeugen der Dokumentation inklusive des gesamten Quelltextes.
- RCS 5.6 zur Versionsverwaltung sämtlicher Sourcen.
- GNU make 3.67 zur Steuerung des gesamten Übersetzungsprozesses (von den Dateien im RCS bis hin zum lauffähigen Programm und zur Dokumentation in PostScript)
- Last but not least die wirklich wichtigen Tools, wie Emacs, Axe, xpst, hypst, lpr, mail, xmh, xrn, Mosaic, Purify (für die ganz harten), tin, top (wie viel Speicher ist noch da?), ...

11. Implementation der GUI

11.1 Modularisierungskonzept

Da sich im Laufe der Implementation herausstellte, daß die Sourcen recht umfangreich wurden, teilten wir das Projekt in mehrere Module auf. Jetzt sind es folgende acht Module:

Hilfe.tcl enthält die Prozeduren für die verschiedenen Hilfefenster und zusätzlich zwei Prozeduren, die Sicherheitsabfragen auf dem Bildschirm ausgeben.

Kommunikation.tcl enthält die Prozeduren, die zum Senden und Empfangen von Nachrichten benötigt werden.

KonfigProcs.tcl hierin befinden sich alle Prozeduren, die zur Programmkonfiguration über das Konfigurationsfenster benötigt werden.

MainProcs.tcl in diesem Modul befinden sich die Prozeduren, die zur Modifikation des Hauptfensters, ohne das Spielfeld, benötigt werden.

Scottland.tcl enthält alle globalen Variablen und die Initialisierungsroutinen.

ShowKonfig.tcl in diesem Modul befindet sich die Prozedur zur Anzeige des Konfigurationsfensters.

ShowMain.tcl enthält die Prozedur zur Anzeige des Hauptfensters.

Spielfeld.tcl in diesem Modul befinden sich alle Prozeduren, die das Spielfeld initialisieren bzw. verändern.

11.2 Beschreibung der Module

Die Beschreibung der Module erfolgt in alphabetischer Reihenfolge, die Beschreibung der Prozeduren in der Reihenfolge ihres Auftretens. Eine Prozedur wird in folgendem Stil beschrieben:
Beispielfunktion

Syntax: `beispielfunktion` {`argumente`}
Beschreibung: hier wird die Prozedur beschrieben.
Eingabe: enthält eine Erläuterung zu den übergebenen Parametern.
Ausgabe: beschreibt die Bedeutung des Rückgabewertes.
Seiteneffekt: wenn Seiteneffekte auftreten können, zum Beispiel durch Änderung einer globalen Variable, dann wird dies an dieser Stelle vermerkt.
Fehler: beschreibt fehlerhaftes Verhalten, soweit bekannt und unvermeidbar.

11.3 Das Modul Hilfe.tcl

In diesem Modul befinden sich die Prozeduren, die zur Anzeige der Hilfe und für Sicherheitsabfragen benötigt werden.

spielende

Syntax: `spielende` {`ausgabe`}
Beschreibung: Erzeugt ein applikationsmodales Fenster, in dem die Nachricht `ausgabe` ausgegeben wird.
Eingabe: `ausgabe`, enthält die anzuzeigende Nachricht.
Ausgabe: —

hilfe

Syntax: `hilfe` {`name`}
Beschreibung: Öffnet ein nichtmodales Fenster, in dem die mit `name` spezifizierte Datei als ASCII-Text angezeigt wird.
Eingabe: `name`, Pfad und Name der anzuzeigenden Hilfedatei.
Ausgabe: —

11.4 Das Modul Kommunikation.tcl

In diesem Modul befinden sich alle Prozeduren, die zum Senden und Empfangen von Nachrichten benötigt werden.

sende

Syntax: `send {text}`
Beschreibung: Steht in der globalen Variable `nachrichtenquelle` der String „Treiber", dann wird die in `text` enthaltene Nachricht über die Pipe an den Treiber gesendet, ansonsten wird die Nachricht nur auf der Console ausgegeben.
Eingabe: `text`, der Text, der gesendet werden soll.
Ausgabe: —

empfange

Syntax: `empfange`
Beschreibung: Steht in der globalen Variable `nachrichtenquelle` der String „Treiber", dann wird die zu empfangende Nachricht mittels Aufruf der IPC-Funktion `preceive` gelesen, ansonsten wird auf der Console *Nachricht: (Leerzeile zum Auwerten)* ausgegeben und danach von der Tastatur gelesen.
Eingabe: —
Ausgabe: gibt die empfangene Nachricht als String zurück.

sende_block_1

Syntax: `sende_block_1 {}`
Beschreibung: Versendet die Nachrichten des ersten Teils der Initialisierungsphase. Hierbei handelt es sich um alle Daten, die beim Schließen des Konfigurationenfensters zur Verfügung stehen, also

- der Name des Spielbrettes,
- die Anzahl der Detektive,
- die Spielstärke,
- die Anzahl der einzelnen Tickets für jeden mitspielenden Detektiv und
- die Anzahl der einzelnen Tickets für Mister X.

Diese Nachrichten werden mittels der Prozedur `sende` gesendet.

Eingabe: —
Ausgabe: —

sende_block_2

Syntax: `sende_block_2 {}`

Beschreibung: Versendet den zweiten Teil der Initialisierungsphase. Hierbei handelt es sich um die Startpositionen der Figuren, also

- die Position von Mister X und
- die Startpositionen der Detektive.

Diese Nachrichten werden mittels der oben beschriebenen Prozedur `sende` gesendet.

Eingabe: —
Ausgabe: —

nachricht_auswerten

Syntax: `nachricht_auswerten {}`

Beschreibung: Diese Funktion wertet die vom Treiber ankommenden Nachrichten aus und leitet die entsprechenden Aktionen ein.

Es wird solange die Prozedur `empfange` aufgerufen und entsprechend der empfangenen Nachricht reagiert, bis die Prozedur `empfange` einen leeren String zurückgibt.

Enthält die globale Variable `nachrichtenquelle` den String „Treiber", dann ruft sich die Prozedur nach 2000 Millisekunden selbst wieder auf.

Eingabe: —
Ausgabe: —

11.5 Das Modul KonfigProcs.tcl

In diesem Modul befinden sich alle Prozeduren, die zur Programmkonfiguration über das Konfigurationsfenster benötigt werden.

reset_defaults

Syntax: `reset_defaults {}`
Beschreibung: Alle Ticketvorräte werden auf ihre Defaultwerte zurückgesetzt. Die einzelnen *Scalebars* werden entsprechend neu gesetzt.
Eingabe: —
Ausgabe: —
Seiteneffekt: Setzen der globalen Variablen `info()`.

gleicher_vorrat

Syntax: `gleicher_vorrat {}`
Beschreibung: Die Ticketvorräte aller Detektive werden auf die Werte des aktuellen Detektivs gesetzt.
Eingabe: —
Ausgabe: —
Seiteneffekt: Setzen der globalen Variablen `info()`.

ticketvorrat

Syntax: `ticketvorrat {}`
Beschreibung: Setzt die Anzeige auf Detektiv `detektiv`, verändert den Wert der globalen Variablen `akt_detektiv`.
Eingabe: `detektiv`, Nummer des Detekitivs, der dargestellt werden soll.
Ausgabe: —
Seiteneffekt: Setzen der globalen Varioablen `akt_detektiv`.

detektiv_anzahl

Syntax: `detektiv_anzahl {}`
Beschreibung: Aktualisiert die Anzahl der Buttons für die Auswahl eines Detektivs. Eventuell wird der ausgewählte Detektiv angepaßt, indem die Prozedur `ticketvorrat` mit dem Parameter `anzahl` aufgerufen wird. Durch Aufruf der Prozedur `detektiv_info` wird die Anzahl auch im Hauptfenster angepaßt.
Eingabe: —
Ausgabe: —
Seiteneffekt: Setzen der globalen Variablen `akt_anzahl`.

11.6 Das Modul MainProcs.tcl

In diesem Modul befinden sich die Prozeduren, die zur Modifikation des Hauptfensters ohne das Spielfeld, also die *Canvas*-Struktur, benötigt werden.

setze_X_tickets

Syntax: `setze_X_tickets {ticket anzahl}`
Beschreibung: In dem globalen Feld `info` wird der Wert an der Stelle `MrX,$ticket` auf den übergebenen Wert `anzahl` gesetzt. Ist der Wert von `anzahl` gleich Null, dann wird der entsprechende *Button* im Hauptfenster vertieft dargestellt, ansonsten wird der *Button* erhöht dargestellt.
Eingabe: `ticket`, Art des zu ändernden Tickets,
`anzahl`, neue Anzahl des Tickets.
Ausgabe: —
Seiteneffekt: Setzen der globalen Variablen `info`, unter Umständen Veränderung der Darstellungsweise des entsprechenden Tickets.

detektiv_info

Syntax: `detektiv_info {anzahl}`
Beschreibung: Paßt die Anzahl angezeigter Detektivinformationen im Hauptfenster auf den übergebenen Wert an. Die Prozedur wurde zum größten Teil mit XF erzeugt. Sie löscht überzählige *Frames*, falls `anzahl` kleiner als der Wert der globalen Variablen `akt_anzahl` ist bzw. erzeugt fehlende *Frames* mit den zugehörigen *Buttons* und belegt die *Buttons* mit ihren Textfeldern mit den entsprechenden Werten.

Eingabe: `anzahl`, neuer Wert für die Anzahl der mitspielenden Detektive.
Ausgabe: —

11.7 Das Modul Scottland.tcl

Dieses Modul enthält die globalen Variablen und Prozeduren zur Initialisierung der Oberfläche. Außerdem wird im Rumpf des Moduls die Prozedur `init_spiel` aufgerufen.

Die globalen Variablen sind:

nachrichtenquelle bezeichnet die verwendete Nachrichtenquelle für Nachrichten vom und zum Treiber bzw Console. Die Variable wird mit dem Wert `Console` vorbelegt.

c enthält den Namen des Spielfeld-*Canvases*. Diese Variable dient vorallem zum Abkürzen des Source-Codes.

figurfarbe ist ein Feld, in dem die Farben zu den Figuren abgelegt werden.

ticketfarbe ist ein Feld, in dem die Farben zu den einzelnen Ticketarten abgelegt werden.

akt_anzahl enthält die Anzahl der am Spiel teilnehmenden Detektive.

max_detektive enthält die maximale Anzahl von Detektiven, die an einem Spiel teilnehmen können.

akt_detektiv enthält den im Konfigurationenfenster ausgewählten Detektiv.

default ist ein zweidimensionales Feld, das zu jedem Spieler (erster Parameter) und jedem Ticket (zweiter Parameter) die voreingestellte Anzahl von Tickets enthält.

info wie `default`, enthält die aktuelle Anzahl von Tickets.

item2feld ist ein Feld, das die Feldnummer des übergebenen Items enthält.

feld2item ist ein Feld, das die Itemnummer des übergebenen Feldes enthält.

item2figur ist ein Feld, das die Detektivnummer des übergebenen Items enthält.

figur2item ist ein Feld, das die Itemnummer des übergebenen Detektivs enthält.

feldfarbe ist ein Feld, das ein Flag für die Feldfarbe des übergebenen Feldes enthält, `""` entspricht Gelb, sonst ist das Feld rot.

block Block der Initialisierungsphase. Diese Variable wird nur mit den Werten 1 und 2 belegt, wobei 1 für die Phase „Erfragen der Ticketvorräte“ und 2 für die Phase „Erfragen der Positionen“ steht.

neues_spiel

Syntax: `neues_spiel {}`

Beschreibung: In dieser Prozedur werden alle Inhalte der Spielfeld-*Canvas* mit Ausnahme der *Bitmap* gelöscht, also die *Items* `text`, `feld`, `detektiv` und `MrX`. Ebenso werden die globalen Datenstrukturen `item2feld`, `feld2item`, `item2figur` und `figur2item` gelöscht. Danach werden die globalen Variablen `block` und `akt_detektiv` jeweils mit 1 initialisiert.

Als nächstes wird die Prozedur `feldpositionen_lesen` aufgerufen. Außerdem wird das Hauptmenü aktualisiert, das heißt im DATEI-Menü wird der Eintrag *Neues Spiel* deaktiviert und der Eintrag *Spiel abbrechen* wird aktiviert.

Zum Schluß wird die Prozedur `sende` mit dem Parameter `new_game` aufgerufen.

Eingabe: —

Ausgabe: —

Seiteneffekt: Es werden die globalen Variablen `c`, `block`, `item2feld`, `feld2item`, `item2figur`, `figur2item` und `akt_detektiv` mit ihren jeweiligen Anfangswerten belegt bzw. geloescht.

spiel_abbrechen

Syntax: `spiel_abbrechen {}`

Beschreibung: In dieser Prozedur werden die Ticket-*Buttons* für Mister X deaktiviert, danach wird das Hauptmenü aktualisiert, das heißt im DATEI-Menü wird der Eintrag *Neues Spiel* aktiviert und der Eintrag *Spiel abbrechen* wird deaktiviert. Der Text der Statuszeile wird auf „Mr.X hat aufgegeben !“ gesetzt.

Es folgt der Aufruf der Prozedur `item_bindings` mit dem Parameter `loeschen`. Zum Schluß wird die Prozedur `sende` mit dem Parameter `game_over` aufgerufen.

Eingabe: —

Ausgabe: —

init_spiel

Syntax: `init_spiel {}`

Beschreibung: Diese Prozedur initialisiert globale Variablen.
Als erstes werden die Farben für Fensterelemente gesetzt.

```
option add *background snow2
option add *activeBackground DodgerBlue
option add *activeForeground snow2

option add *Scrollbar.foreground LightGray
option add *Scrollbar.activeForeground DodgerBlue
option add *Scale.sliderForeground LightGray
option add *Scale.activeForeground DodgerBlue
```

Danach werden die globalen Variablen `figurfarbe`, `ticketfarbe`, `max_detektive`, `akt_detektiv` und `block` mit ihren Startwerten belegt.

```
set figurfarbe(1)   DeepSkyBlue
set figurfarbe(2)   magenta
set figurfarbe(3)   brown
set figurfarbe(4)   green
set figurfarbe(5)   peru
set figurfarbe(6)   coral
set figurfarbe(7)   DarkGreen
set figurfarbe(8)   purple
set figurfarbe(9)   khaki
set figurfarbe(MrX) tomato

set ticketfarbe(Taxi)        gold
set ticketfarbe(Taxi,hell)   yellow
set ticketfarbe(Bus)         SeaGreen
set ticketfarbe(Bus,hell)    MediumSeaGreen
set ticketfarbe(U-Bahn)      red
set ticketfarbe(U-Bahn,hell) tomato
set ticketfarbe(Black)       #404040
set ticketfarbe(Black,hell)  DimGrey
set ticketfarbe(2x)          OrangeRed
set ticketfarbe(2x,hell)     orange

set max_detektive 9
```

```
set akt_detektiv 1
set block 1
```

Als nächstes werden die Werte von Mister X und den Detektiven initialisiert.

```
foreach wert {
    taxi bus ubahn black twice zuletzt zeigen} {
  set info(MrX,$wert) {-}
}

for {set i 1} {$i <= $max_detektive} {incr i} {
  foreach wert {
      taxi bus ubahn aktuell zuletzt benutzt} {
    set info($i,$wert) {-}
  }
}
```

Als nächstes wird die Prozedur `ShowWindow` aufgerufen. Danach wird die globale Variable `akt_anzahl` auf 0 gesetzt, mit dem Aufruf der Prozedur `detektiv_info 5` die Anzahl der im Hauptfenster dargestellten Detektive auf 5 gesetzt und anschließend wieder die globale Variable `akt_anzahl` auf 5 gesetzt.

Zum Schluß wird die Argumentenliste ausgewertet:

- Zuerst wird nach dem Argument `-port` gesucht. Ist das Argument nicht angegeben, wird die Console zum Nachrichtenaustausch verwendet, sonst wird die Prozedur `pconnect` mit den Parametern `client` und der Zeichenkette, die in der Argumentliste auf das Wort `-port` folgt, aufgerufen.
- Wird das Argument `-fun` gefunden, dann wird die Prozedur `init_tour` aufgerufen.
- Schließlich wird die Variable `sound` auf 1 gesetzt, wenn das Argument `-sound` gefunden wird, sonst erhält diese Variable den Wert 0.

Eingabe: —
Ausgabe: —
Seiteneffekt: Es werden die globalen Variablen `max_detektive`, `akt_detektiv`, `akt_anzahl`, `block`, `info`, `figurfarbe`, `ticketfarbe`, `nachrichtenquelle` und `sound` initialisiert.

11.8 Das Modul ShowKonfig.tcl

In diesem Modul befindet sich die Prozedur zum Anzeigen des Konfigurationenfensters. Diese Prozedur wurde überwiegend mit XF erzeugt.

ShowWindow.wi_konfig

Syntax: `ShowWindow.wi_konfig {}`
Beschreibung: Es wird das Fenster mit dem Menü und den benötigten Rahmen erzeugt und angezeigt.
Eingabe: —
Ausgabe: —
Seiteneffekt: Es werden unter Umständen die globalen Variablen `akt_anzahl`, `max_detektive`, `spielplan`, `spielstaerke`, `boards`, `levels` und `info` verändert.

11.9 Das Modul ShowMain.tcl

In diesem Modul befindet sich die Prozedur zum Anzeigen des Hauptfensters.

ShowWindow.

Syntax: `ShowWindow. {}`
Beschreibung: Diese Prozedur erzeugt das Hauptfenster mit allen untergeordneten Elementen und zeigt es an.
Eingabe: —
Ausgabe: —

11.10 Das Modul Spielfeld.tcl

In diesem Modul befinden sich alle Prozeduren, die das Spielfeld initialisieren bzw. verändern.

feldpositionen_lesen

Syntax: `feldpositionen_lesen {}`
Beschreibung: Diese Prozedur liest aus der Datei **feldpositionen.dat** die zu jedem Feld gehörenden Koordinaten und ein Flag für die Farbe. Alle Felder werden auf dem Spielfeld angezeigt.
Eingabe: —

Ausgabe: —
Seiteneffekt: Es werden die Variablen `item2feld`, `feld2item` und `fedlfarbe` initialisiert.

feld_aktivieren

Syntax: `feld_aktivieren {feld}`
Beschreibung: Markiert das Feld Nummer `feld` als ein zum aktuellen Spielplan gehörendes Feld. In Abhängigkeit von `feldfarbe(feld)` wird es entweder gelb (= `""`) oder rot (sonst) gefärbt.
Eingabe: `feld`, Nummer des zu aktivierenden Feldes.
Ausgabe: —

figur_enter

Syntax: `figur_enter {}`
Beschreibung: Markiert die ausgewählte Figur und sichert die zugehörige Farbe.
Eingabe: —
Ausgabe: —
Seiteneffekt: Setzen der globalen Variablen `oldfill`.

figur_leave

Syntax: `figur_leave {}`
Beschreibung: Setzt die Farbe der ausgewählten Figur auf `oldfill` zurück.
Eingabe: —
Ausgabe: —

feld_enter

Syntax: `feld_enter {}`
Beschreibung: Markiert das ausgewählte Feld und sichert die zugehörige Farbe.
Eingabe: —
Ausgabe: —
Seiteneffekt: Setzen der globalen Variablen `oldfill`.

feld_leave

Syntax: `feld_leave {}`
Beschreibung: Setzt die Farbe des ausgewählten Feldes auf `oldfill` zurück.
Eingabe: —
Ausgabe: —

setze_figur

Syntax: `setze_figur {nr feld}`
Beschreibung: Setzt Figur `nr` auf das Feld `feld`.
Eingabe: —
Ausgabe: —
Seiteneffekt: Verändern der globalen Variablen `item2figur` und `figur2item`.

move_figur_start

Syntax: `move_figur_start {}`
Beschreibung: Löscht die *Bindings* der in der globalen Variablen `move_det_nr` stehenden Figur und ruft die Prozedur `item_bindings` mit dem Parameter `feld` auf.
Eingabe: —
Ausgabe: —

move_figur_end

Syntax: `move_figur_end {}`
Beschreibung: Das Gegenstück zur Prozedur `move_figur_start`:
Beendet das Setzen einer Figur auf dem Spielfeld.
Eingabe: —
Ausgabe: —

setze_mrx_end

Syntax: `setze_mrx_end {}`
Beschreibung: Beendet den Positionierungsvorgang von Mister X.
Eingabe: —
Ausgabe: —

X_move

Syntax: `X_move {t}`

Beschreibung: Diese Prozedur wird aufgerufen, wenn Mister X zieht. Sie besteht aus einem Aufruf der Prozedur `play` mit dem weitergerreichten Parameter `t`, aus einer Schleife, in der alle Ticket-Buttons deaktiviert werden und aus einem Aufruf der Prozedur `sende` mit dem Parameter `"X_move $t"`. Außerdem wird der Text der Statuszeile auf „Bitte warten – Detektive ziehen" gesetzt.

Eingabe: `t`, die verwendete Ticketart.

Ausgabe: —

item_bindings

Syntax: `item_bindings {typ}`

Beschreibung: In Abhängigkeit des Parameters `typ` werden die *Bindings* der Elemente des Spielfeldes gsetzt bzw. gelöscht.

```
switch $typ {
mrx {
  $c bind feld <Any-Enter> "feld_enter"
  $c bind feld <Any-Leave> "feld_leave"
  $c bind feld <1> "setze_mrx_end"
  $c bind text <Any-Enter> "feld_enter"
  $c bind text <Any-Leave> "feld_leave"
  $c bind text <1> "setze_mrx_end"
}
feld {
  $c bind mrx <Any-Enter> ""
  $c bind mrx <Any-Leave> ""
  $c bind mrx <1> ""
  $c bind detektiv <Any-Enter> ""
  $c bind detektiv <Any-Leave> ""
  $c bind detektiv <1> ""
  $c bind feld <Any-Enter> "feld_enter"
  $c bind feld <Any-Leave> "feld_leave"
  $c bind feld <1> "move_figur_end"
  $c bind text <Any-Enter> "feld_enter"
  $c bind text <Any-Leave> "feld_leave"
  $c bind text <1> "move_figur_end"
  $c configure -cursor hand2
```

```
}
figur {
  $c bind mrx <Any-Enter> "figur_enter"
  $c bind mrx <Any-Leave> "figur_leave"
  $c bind mrx <1> "move_figur_start"
  $c bind detektiv <Any-Enter> "figur_enter"
  $c bind detektiv <Any-Leave> "figur_leave"
  $c bind detektiv <1> "move_figur_start"
  $c bind feld <Any-Enter> ""
  $c bind feld <Any-Leave> ""
  $c bind feld <1> ""
  $c bind text <Any-Enter> ""
  $c bind text <Any-Leave> ""
  $c bind text <1> ""
  $c configure -cursor left_ptr
}
loeschen {
  $c bind mrx <Any-Enter> ""
  $c bind mrx <Any-Leave> ""
  $c bind mrx <1> ""
  $c bind detektiv <Any-Enter> ""
  $c bind detektiv <Any-Leave> ""
  $c bind detektiv <1> ""
  $c bind feld <Any-Enter> ""
  $c bind feld <Any-Leave> ""
  $c bind feld <1> ""
  $c bind text <Any-Enter> ""
  $c bind text <Any-Leave> ""
  $c bind text <1> ""
  $c configure -cursor left_ptr
}
```

Eingabe: `typ`, spezifiziert die Art der *Bindings*, die gesetzt werden sollen.
Ausgabe: —

12. Implementation der IPC

12.1 Ausgewählte IPC

In der engeren Wahl sind die IPC via FiFO's und die SystemV-IPC.

12.1.1 Ein FiFo Prototyp

Zunächst wurde für die FiFo-IPC ein PortLibrary Prototyp erstellt. Es stellte sich heraus, daß es für die Kommunikation zwingend notwendig ist, ein nicht-blockierendes Empfangen von Nachrichten zu ermöglichen. Sonst würde der ProSet-Prozeß bei dem Versuch eine *mögliche* (und eventuell gar nicht vorhandene) Benutzereingabe zu lesen, bis zur nächsten Benutzereingabe blockieren. Dies erfordert bei der Verwendung von FiFO's einen zusätzlichen Mechanismus. Dieser muß die PortLibrary befähigen auch Nachrichtenfragmente korrekt zu empfangen, ohne dabei zu blockieren. Der Empfänger will jedoch nur komplette oder gar keine Nachrichten empfangen. Das bedeutet, die PortLibrary muß dem Empfänger die Nachricht vorenthalten, bis die Nachricht durch spätere Empfangsanforderungen des Empfängers komplettiert wurde. Die SystemV-IPC bietet nichtblockierendes Lesen/Schreiben, bei dem entweder nur die komplette Message oder gar keine Message gelesen/geschrieben wird implizit an.

12.1.2 Der SystemV Prototyp

Der SystemV Prototyp erfüllte nicht nur die Forderungen, sondern seine Implementierung gestaltete sich auch einfacher als die der FiFo Variante.

12.2 Implementation

12.2.1 Schnittstellenänderung

Die Schnittstelle wird nicht auf Strings begrenzt, sondern ist auch in der Lage andere Datentypen zu transportieren. Dies stellt zwar eine Erweiterung der geforderten Funktionalität dar, ist aber übersichtlicher implementierbar. Wird eine Schnittstelle gewünscht, die nur noch Strings transportiert, kann diese durch eine sehr einfache zusätzliche Protokollschicht oberhalb der PortLibrary-Schnittstelle erreicht werden. Diese könnte aus je einer neuen Sende- und Empfangsfunktion, die auf den Sende und Empfangsfunktionen der PortLibrary aufgesetzt werden, bestehen.

12.2.2 IPC Schlüsselbildung

Für die Implementation der SystemV Variante der PortLibrary ist es notwendig, einen Kommunikationsschlüssel zu vergeben, da mehrere Scotland Yard Programme parallel benutzt werden können und sich nicht gegenseitig beeinflussen sollen. Hierzu werden IPC Identifier vergeben. Diese Identifier können durch eine UNIX Systemfunktion erzeugt werden. Diese Funktionen benötigt als Parameter einen Dateinamen und einen beliebigen Buchstaben, anhand dessen eine eindeutige Nummer berechnet wird. Um sicherzustellen, daß nicht mehrere Scotland Yard Prozesse hierzu dieselbe Datei benutzen, wird diese Datei bei dem Aufruf der `port_server_connect(...)` exclusiv erzeugt. Der Pfad zu dieser Datei läßt sich über ein Macro im Header "ipcid.h" definieren. Entspricht dieser Pfad dem Arbeitsverzeichnis, oder ist relativ angegeben, so ist darauf zu achten sowohl Server als auch Client im demselben Verzeichnis zu starten. Scheitert die Erzeugung der Datei, so scheitert auch die serverseitige Eröffnung des Ports.

Bugs - Es ist peinlichst darauf zu achten, niemals einen Clienten zu starten, wenn der Server den entsprechenden Port nicht eröffnen konnte. Dieser Port könnte möglicherweise bereits von einem anderem Server-Clienten Paar unterhalten werden. Da in der momentanen Version kein Anmeldeprozedere existiert, endstehen Konflikte, wenn ein Port fälschlicherweise von mehreren Clienten gleichzeitig benutzt wird.

12.3 Testwerkzeuge

Zum Test wärend der Implementierung der PortLibrary wurden zunächst ein Testserver `stest` und ein Testclient `ctest` implementiert. Diese Programme öffnen einen Port und senden sich gegenseitig die Nachrichen "ping" und "pong".

12.3.1 Porttalk

Für weitergehende Tests von späteren, die PortLibrary verwendenden Applikationen wurde zusätzlich das Programm `Porttalk` erstellt. `Porttalk` bietet dem Benutzer eine interaktive Kommunikation wahlweise als Server oder als Client. Das Testprogramm `Porttalk` akzeptiert die folgenden Optionen:

–h Gibt einen Hilfetext aus

–N Definiert den Applikationsnamen für die Fehler und Debugging Ausgaben.

−S Porttalk startet als Server

−C Porttalk startet als Client

−P <port> Definiert den Namen des zu verwendenden Ports

−a Asynchrones Empfangen (nicht-blockierend) von Nachrichten

−s Synchrones Empfangen (blockierend) von Nachrichten

−q Unterdrücken der PortLibrary Debugging Ausgaben

−v Diverse Informationen ausgeben

−A Automatisch generierte Nachrichten versenden, wie `stest` und `ctest`.

13. Die Implementierung der Treiberkomponente

In diesem Kapitel ist die ProSet-Implementierung der Treiber- und Regelkomponente dokumentiert. Das Hauptprogramm und die wichtigen Prozeduren und Funktionen sind jeweils mit ihren Hilfs- und Unterprozeduren in eigenen Abschnitten aufgeführt. Daran anschließend finden sich die Programmteile, die zum Test verwendet wurden.

13.1 Das Hauptprogramm der Treiberkomponente und globale Hilfsprozeduren

Das Hauptprogramm „ScotlandYard“ kann mit dem Parameter -s gestartet werden. Dadurch werden die Benutzereingaben durch eine Datei mit dem Namen `ScotlandYard.sim` simuliert. In dieser Simulationsdatei sind die Nachrichten, sie sonst von der GUI kommen, gespeichert. Beispiel für den Aufruf: `ScotlandYard -s`. Mit dem Parameter `-d` kann ein Arbeitsverzeichnis für das Programm spezifiziert werden. Beispiel: `ScotlandYard -d /prg/games/Scotland`.

⟨*ScotlandYard.pst* 127⟩≡
```
program ScotlandYard;
  visible max_D_count;
  visible D_count;
  visible level;
  visible D_positions;
  visible X_positions;
  visible D_tickets;
  visible X_tickets;
  visible D_moves;
  visible number_of_D_moves;
  visible X_moves;
  visible number_of_X_moves;
  visible last_public_X_position;
  visible used_X_tickets;
```

```
  visible all_used_X_tickets;
  visible simulate;
  visible board_name;
  visible board;
  visible matrix;
  visible matrix_laenge;
  visible taxi;
  visible bus;
  visible underground;
  visible interface; -- Kommunikationsschnittstelle zur Planungskomponente
  visible constant MAX_KNOTEN          ← 199;
  visible constant SCOTLAND_YARD       ← "ScotlandYard";
  visible constant SIMULATION_FILENAME ← SCOTLAND_YARD + ".sim";
  visible constant SCOTLAND_PIPE       ← SCOTLAND_YARD + "." + pipename();
  visible constant SCOTLAND_GUI        ← SCOTLAND_YARD + ".gui";
  visible constant SCOTLAND_PLANS      ← SCOTLAND_YARD + ".planlist";
  visible SCOTLAND_DIRECTORY            ← "";
begin
  ⟨Parameter einlesen 129⟩
  start_gui();
  repeat
    ok ← init() when game_over_message use game_over;
    if ok = Ω
       then start_strategy();
            again ← play_the_game() when game_over_message use game_over;
            end_game();
       elseif ¬ ok
              then quit; -- Programmabbruch
       else again ← true; -- Spiel neu starten
    end if;
  until ¬ again end repeat;
  if simulate
     then fclose(SCOTLAND_DIRECTORY + SIMULATION_FILENAME);
     else c_fct_call port_disconnect();
  end if;

  ⟨pipename 130a⟩
  ⟨start_gui 135⟩
  ⟨init 136⟩
  ⟨start_strategy 146⟩
  ⟨play_the_game 148⟩
  ⟨send_gui_message 181b⟩
```

```
    ⟨get_gui_message 182a⟩
    ⟨X_moves_to_visible 182b⟩
    ⟨possible_X_positions 183⟩
    ⟨is_X_arrested 133a⟩
    ⟨Exceptionhandler: game_over 149b⟩
    ⟨end_game 130b⟩

    ⟨Ticketvorraete der GUI melden 131a⟩
    ⟨Ticketvorraete von X der GUI melden 131b⟩
    ⟨erreichbar 132⟩

  @include "detect.pst"

    ⟨string2integer 133b⟩
    ⟨capsulated_str 134a⟩
    ⟨capsulated_system 134b⟩
    ⟨newbreak 185b⟩
    ⟨newsubset 186⟩

  end ScotlandYard;
```

Root chunk (not used in this document).

Der folgende Programmteil interpretiert mögliche Parameter, die mit der Kommandozeile übergeben wurden. Mögliche Parameter sind `-s` für die Simulation der GUI über eine Datei und `-d` für die Angabe eines Arbeitsverzeichnisses.

⟨*Parameter einlesen* 129⟩≡

```
    if argv(2) = "-d"
       then SCOTLAND_DIRECTORY ← argv(3);
       elseif argv(3) = "-d"
              then SCOTLAND_DIRECTORY ← argv(4);
    end if;
    if SCOTLAND_DIRECTORY = Ω
       then put("Parameter '-d': Kein Arbeitsverzeichnis angegeben.");
            stop 1; -- Abbruch mit Fehler
    end if;
    if #SCOTLAND_DIRECTORY > 0
       then if SCOTLAND_DIRECTORY(#SCOTLAND_DIRECTORY) ≠ "/"
               then SCOTLAND_DIRECTORY +← "/";
            end if;
    end if;
    simulate ← (argv(2) = "-s") ∨ (argv(4) = "-s");
```

This code is used in chunk 127.

Die Funktion `pipename()` erzeugt einen Namen für die Kommunikationsschnittstelle zwischen der GUI und dem Treiber. Dieser Name ist eindeutig für jede Instantiierung des Programmes *Scotland Yard*.

⟨*pipename* 130a⟩≡
```
procedure pipename();
  hidden constant name ← "pipename";
begin
  pid  ← c_fct_call getpid() c_integer;
  cmd ← "echo \\\"‘hostname‘"echo \\\ + capsulated_str(pid)
         + "\\\" > "\\\ + name;
  capsulated_system(cmd);
  fopen(name, "r");
  fget(name, filename);
  fclose(name);
  capsulated_system("rm -f " + name);
  return (filename);
end pipename;
```
This code is used in chunk 127.

Die folgende Prozedur fordert die Prozesse der Planung für die Detektive über den Tupelraum auf, sich zu beenden. Nachdem dies geschehen ist, wird der Tupelraum für die Kommunikation mit der Planung entfernt.

⟨*end_game* 130b⟩≡
```
procedure end_game();
begin
  if ExistsTS(interface)
     then for nr ∈ [1..D_count] do
              deposit ["commit_suicide"] at interface end deposit;
          end for;

(* Zur Zeit der Implementierung war es nicht moeglich,
   vom zuletzt gestarteten Detektiv eine Bestaetigung
   zu erhalten. (Problem von ProSet-Linda)

          for nr ∈ [1..D_count] do
              fetch ( "died" ) at interface end fetch;
          end for;
          RemoveTS(interface);
*)
  end if;
end end_game;
```
This code is used in chunk 127.

Die folgende Prozedur teilt der GUI die Ticketvorräte der Detektive und von X mit. Die Prozedur wird global im Hauptprogramm definiert, weil sie von den Prozeduren `init()` und `send_move_to_gui()` verwendet wird.

⟨*Ticketvorraete der GUI melden* 131a⟩≡

```
procedure send_ticket_data(rd maximum);
begin
  for nr ∈ [1..maximum] do
      message ← "D_tickets " + capsulated_str(nr) + " "
                + capsulated_str(D_tickets(nr)(1)(2)) + " "
                + capsulated_str(D_tickets(nr)(2)(2)) + " "
                + capsulated_str(D_tickets(nr)(3)(2));
      send_gui_message(message);
  end for;
  send_X_tickets();
end send_ticket_data;
```

This code is used in chunks 127, 215, and 221.

⟨*Ticketvorraete von X der GUI melden* 131b⟩≡

```
procedure send_X_tickets();
begin
  message ← "X_tickets " + capsulated_str(X_tickets(1)(2)) + " "
                         + capsulated_str(X_tickets(2)(2)) + " "
                         + capsulated_str(X_tickets(3)(2)) + " "
                         + capsulated_str(X_tickets(4)(2)) + " "
                         + capsulated_str(X_tickets(5)(2));
  send_gui_message(message);
end send_X_tickets;
```

This code is used in chunk 127.

`erreichbar` errechnet aus einem übergebenen Startknoten und einem Ticket ein Tupel aller Knoten, die erreicht werden können. Die Prozedur wird von `shortest_path` und `possible_X_positions` verwendet und ist deshalb global im Hauptprogramm definiert.

⟨*erreichbar* 132⟩≡

```
procedure erreichbar(knoten,ticket);
begin
   strecken ← [x(2) : x ∈ board | x(1) = ticket](1);
   z ← ∅;
   if ((strecken ≠ Ω) ∧ (strecken ≠ ∅))
   then
      zz ← strecken (knoten);
      if (zz ≠ Ω)
      then
         z ← –k : k ∈ zz";
      end if;
   end if;
   if (ticket = "black")
   then
      for t ∈ ["taxi", "bus", "underground"] do
         z +← erreichbar (knoten,t);
      end for;
   end if;
   return (z);
end erreichbar;
```

This code is used in chunk 127.

Die nächste Prozedur stellt fest, ob Mister X von den Detektiven gefangen wurde. Sie muß im Laufe des Programms zweimal aufgerufen werden.

1. Nach dem Zug von Mister X um festzustellen, ob Mister X auf einer von Detektiven umzingelten Position stand.
2. Nach den Zügen der Detektive, um festzustellen, ob Mister X jetzt nur noch auf von Detektiven besetzten Feldern steht.

⟨*is_X_arrested* 133a⟩≡

```
procedure is_X_arrested();
begin
          if newsubset(X_positions, -x: x ∈ D_positions")
             then winners ← "";
                  for i ∈ [1..D_count] do
                      if D_positions(i) ∈ X_positions
                         then winners ← winners + ","
                                        + capsulated_str(i);
                      end if;
                  end for;
                  return ( "Mister X ist von einem\n "
                          +"der folgenden Detektive\n "
                          +"gefangen worden:\n"
                          + winners(2..));
             else return Ω;
          end if;
end is_X_arrested;
```

This code is used in chunks 127 and 216.

ie folgende Funktion erlaubt im Gegensatz zur Standardfunktion `atoi()` auch die Verwendung von leeren Strings als Parameter. Bei einem leeren String wird `om` zurückgeliefert.

⟨*string2integer* 133b⟩≡

```
procedure string2integer(text);
  persistent constant atoi : "StdLib";
begin
  if #text > 0
     then return (atoi(text));
     else return (Ω);
  end if;
end string2integer;
```

This code is used in chunks 127, 216, and 221.

Die beiden folgenden Funktionen sind nötig, weil es zum Zeitpunkt der Implementierung nicht möglich war, persistente Funktionen mittels `visible` global bekannt zu machen.

⟨*capsulated_str* 134a⟩≡
```
procedure capsulated_str(value);
  persistent constant str: "StdLib";
begin
  return str(value);
end capsulated_str;
```
This code is used in chunk 127.

⟨*capsulated_system* 134b⟩≡
```
procedure capsulated_system(command);
  persistent constant system: "StdLib";
begin
  system(command);
end capsulated_system;
```
This code is used in chunk 127.

13.2 Die Prozedur simulate_gui

Diese Prozedur wurde in `start_gui()` integriert, weil sie nur die Aufgabe hat, die Datei zu öffnen, die die Simulationsanweisungen enthält.

13.3 Die Prozedur start_gui

Diese Prozedur öffnet den Port zur Kommunikation als Server und startet den GUI-Prozess, wenn der Simulationsmodus nicht aktiviert ist. Ansonsten wird die Simulation von Nachrichten der GUI vorbereitet, indem versucht wird, die Datei zu öffnen, die die Simulationsanweisungen enthält. Falls dies scheitert, bricht das Programm ab.

⟨*start_gui* 135⟩≡

```
procedure start_gui();
begin
  if simulate
     then fopen (SCOTLAND_DIRECTORY + SIMULATION_FILENAME, "r");
     else c_fct_call port_server_connect(SCOTLAND_PIPE : c_string);
          nonblocking ← 0;
          c_fct_call port_toggle_receive(nonblocking : c_integer);
          capsulated_system(SCOTLAND_GUI + " -port " + SCOTLAND_PIPE
                            + if #SCOTLAND_DIRECTORY > 0
                                then " -dir " + SCOTLAND_DIRECTORY
                                else ""
                              end if
                            + " &");
  end if;
end start_gui;
```

This code is used in chunks 127, 206, 215, 216, and 221.

13.4 Die Prozedur `init` und ihre Unterprozeduren

Diese Prozedur teilt der Graphischen Benutzerschnittstelle die Standardwerte für das Spiel mit und initialisiert die globalen Variablen. Nimmt eventuelle Änderungen der GUI entgegen und paßt die globalen Variablen entsprechend an. Siehe Nachrichtenprotokoll für GUI und Treiber, Abschnitt 6.

⟨*init* 136⟩≡
```
procedure init();
  hidden constant list_of_level_names ←
        ["Schwachsinn",
         "Kurzsichtig",
         "Distanz",
         "Distanz-Summe",
         "Stuttgarts",
         "Mix",
         "Anti-Knubble"];
begin
  list_of_board_names ← load_board_names();
  ⟨Initialisierung der globalen Variablen 137⟩
  ⟨Standardwerte senden 138⟩
  ⟨Endgueltige Startwerte empfangen 139⟩
  ⟨Startpositionen senden 140⟩
  ⟨Endgueltige Startpositionen empfangen 141⟩

  ⟨load_board_names 142⟩
  ⟨load_board 144⟩
  ⟨generate_X_position 143b⟩
  ⟨generate_D_positions 143a⟩
end init;
```
This code is used in chunks 127 and 221.

Das folgende Programmsegment setzt alle globalen Variablen auf ihre Startwerte.

Die Reihenfolgen der Tupelelemente in `D_tickets` und in `X_tickets` sind festgelegt. Die erste Komponente enthält der Vorrat an Taxitickets, die zweite den Vorrat an Bustickets und die dritte den der U-Bahntickets. Zusätzlich enthält `X_tickets` in der vierten Komponente den Vorrat an Blacktickets und in der fünften den der Doppelzugtickets.

⟨*Initialisierung der globalen Variablen* 137⟩≡

```
max_D_count            ← 10;
D_count                ← 5;
level                  ← 1;
D_tickets              ← [];
for nr ∈ [1..max_D_count] do
    D_tickets(nr) ← [["taxi",10],["bus",8],["underground",4]];
end for;
X_tickets  ← [["taxi",4],["bus",3],["underground",3],
              ["black",D_count],["twice",2]];
D_moves                ← 0;
number_of_D_moves      ← Ω; -- später Integer
X_moves                ← 0;
number_of_X_moves      ← Ω; -- später Integer
last_public_X_position ← Ω; -- später Integer
used_X_tickets         ← [];
all_used_X_tickets     ← [];
```

This code is used in chunk 136.

Im folgenden Programmsegment werden der GUI alle erforderlichen Standardwerte übermittelt. Dieses sind: Spielbrettnamen, Anzahl der Detektive, Schwierigkeitsgrad, die Ticketvorräte der Detektive, der Ticketvorrat von Mister X und die Anzahl der Züge bis zum nächsten Auftauchen von Mister X.

⟨*Standardwerte senden* 138⟩≡

```
-- Nachricht  new_game empfangen:
repeat
  message ← get_gui_message();
until message ≠ "" end repeat;
⟨DEBUG_init_3 191a⟩
for name ∈ list_of_board_names do
    send_gui_message("board_name " + name);
end for;
send_gui_message("count " + capsulated_str(D_count)
                          + " " + capsulated_str(max_D_count));
send_gui_message("level " + list_of_level_names(level));
for level_name ∈ list_of_level_names |
    (level_name ≠ list_of_level_names(level)) do
  send_gui_message("level " + level_name);
end for;
send_ticket_data(max_D_count);
send_gui_message("X_show " + capsulated_str(X_moves_to_visible()));
send_gui_message("block_end");
```

This code is used in chunk 136.

Im Anschluß werden alle zuvor gesendeten Nachrichten von der GUI bearbeitet und zurückgesendet. Mit diesen Werten werden die globalen Variablen aktualisiert.

⟨*Endgueltige Startwerte empfangen* 139⟩≡

```
repeat
  message_save ← get_gui_message();
  message ← message_save;
  messagetext ← newbreak(message, " ");
  if (#message ≥ 2)
     then message ← message(2..);
  end if;
  case messagetext
       when "board_name"
         ⇒ board_name ← message;
       when "level"
         ⇒ i ← 1;
           ⟨DEBUG_init_4 191b⟩
           while (message ≠ list_of_level_names(i)) do
           i + ← 1;
           end while;
           level ← i;
       when "count"
         ⇒ D_count ← string2integer(message);
       when "D_tickets"
         ⇒ hold ← newbreak(message, " ");
           nr ← string2integer(hold);
           message ← message(2..);
           -- taxi:
           D_tickets(nr)(1)(2) ← string2integer(newbreak(message,
                                                          " "));
           message ← message(2..);
           -- bus:
           D_tickets(nr)(2)(2) ← string2integer(newbreak(message,
                                                          " "));
           message ← message(2..);
           -- underground:
           D_tickets(nr)(3)(2) ← string2integer(message);
       when "X_tickets"
         ⇒ -- taxi:
           X_tickets(1)(2) ← string2integer(newbreak(message,
                                                    " "));
           message ← message(2..);
```

```
                -- bus:
                X_tickets(2)(2) ← string2integer(newbreak(message,
                                                          " "));
                message ← message(2..);
                -- underground:
                X_tickets(3)(2) ← string2integer(newbreak(message,
                                                          " "));
                message ← message(2..);
                -- black:
                X_tickets(4)(2) ← string2integer(newbreak(message,
                                                          " "));
                message ← message(2..);
                -- twice:
                X_tickets(5)(2) ← string2integer(message);
           ⟨DEBUG_init_1 190b⟩
      end case;
    until (message_save = "block_end") end repeat;
```

This code is used in chunk 136.

Erst jetzt ist der Spielplan vom Benutzer ausgewählt. Da die Startpositionen von diesem abhängen, werden die möglichen Positionen zusammen mit den zufällig gewählten Startpunkten für Mister X und die Detektive der GUI mitgeteilt:

⟨*Startpositionen senden* 140⟩≡

```
    start_nodes ← load_board(board_name);
    taxi        ← [x(2) : x ∈ board | x(1) = "taxi"       ](1);
    bus         ← [x(2) : x ∈ board | x(1) = "bus"        ](1);
    underground ← [x(2) : x ∈ board | x(1) = "underground"](1);
    X_positions ← domain(taxi);
    for node ∈ domain(taxi) do
        send_gui_message("vertex " + capsulated_str(node));
    end for;
    D_positions ← generate_D_positions(start_nodes);
    for nr ∈ [1..D_count] do
        send_gui_message("D_startposition " +
                         capsulated_str(nr) + " " +
                         capsulated_str(D_positions(nr)));
    end for;
    send_gui_message("X_position " +
                     capsulated_str(generate_X_position(start_nodes)));
    send_gui_message("block_end");
```

This code is used in chunk 136.

Der Benutzer kann diese Startpositionsvorschläge ändern. Deshalb wird auf die endgültigen Startpunkte der Detektive gewartet. Die Initialisierungskommunikation wird beendet, wenn "init_end" empfangen wurde:

⟨*Endgueltige Startpositionen empfangen* 141⟩≡

```
new_X_positions ← true;
new_all_used_X_tickets ← true;
loop
  message ← get_gui_message();
  if message = "init_end"
     then quit;
  end if;
  msg ← newbreak(message, " "); -- aus  D_startposition 2 134 wird  2 134
  message ← message(2..);
  case msg
       when "D_startposition"
            ⇒ hold ← newbreak(message, " ");
              nr ← string2integer(hold);
              message ← message(2..);
              D_positions(nr) ← string2integer(message);
       when "move_no"
            ⇒ X_moves ← string2integer(message);
              D_moves ← X_moves;
       when "possible_X_position"
            ⇒ if new_X_positions
                 then X_positions ← ∅;
                      new_X_positions ← false;
              end if;
              if message ≠ "-" -- keine möglichen Positionen
                 then X_positions with ← string2integer(message);
              end if;
       when "used_X_ticket"
            ⇒ if new_all_used_X_tickets
                 then all_used_X_tickets ← [];
                      new_all_used_X_tickets ← false;
              end if;
              if message ≠ ""
                 then all_used_X_tickets with ← message;
              end if;
       when "last_X_position"
            ⇒ last_X_position ← string2integer(message);
       else ⟨DEBUG_init_2 190c⟩
```

```
    end case;
  end loop;
```
This code is used in chunk 136.

Im folgenden werden die von der Prozedur `init()` verwendeten Unterroutinen beschrieben.

Die folgende Prozedur sucht im aktuellen Verzeichnis nach Spielplannamen und liefert dann eine Liste der gefunden Namen zurück. Ist der Standardspielplan in dieser Liste enthalten, wird er an den Anfang der Liste gesetzt. Für die Ausführung wird der *Stream Editor* `sed` von UNIX und ein zugehöriges Script *ScotlandYard.sed* benötigt.

⟨*load_board_names* 142⟩≡
```
procedure load_board_names();
  hidden planlist ← [];
  hidden planname ← "";
begin
  capsulated_system("ls " + SCOTLAND_DIRECTORY + "*.plan | sed -f "
                    + SCOTLAND_DIRECTORY + "ScotlandYard.sed > "
                    + SCOTLAND_PLANS);
  fopen(SCOTLAND_PLANS, "r");
  loop
    fget(SCOTLAND_PLANS, planname);
    if planname ≠ ""
      then planlist with ← planname(#SCOTLAND_DIRECTORY+1..#planname-5);
      else quit;
    end if;
  end loop;
  fclose(SCOTLAND_PLANS);
  capsulated_system("rm -f " + SCOTLAND_PLANS);
  if SCOTLAND_YARD ∈ planlist
     then planlist ← [SCOTLAND_YARD] +
                      [ x: x ∈ planlist | (x ≠ SCOTLAND_YARD) ];
  end if;

  return (planlist);
end load_board_names;
```
This code is used in chunks 136 and 199.

Die nächste Funktion erzeugt aus den verfügbaren Startknoten zufällige Startpositionen für die Detektive. Dazu müssen mindestens `D_count`+1 (eine Position wird noch an Mister X vergeben) mögliche Startpositionen vorhanden sein. Die erforderlichen Daten (Spielplan, Anzahl der Detektive) werden den entsprechenden globalen Variablen entnommen. Die erzeugten Positionen werden als Tupel von Knotennummern zurückgeliefert.

⟨*generate_D_positions* 143a⟩≡
```
procedure generate_D_positions(rd start_positions);
begin
  positions ← [];
  for nr ∈ [1..D_count] do
    repeat
      startposition ← random(start_positions);
    until startposition ∉ positions end repeat;
    positions with ← startposition;
  end for;
  return positions;
end generate_D_positions;
```
This code is used in chunks 136 and 210a.

Das gleiche wird für die Startposition von Mister X durchgeführt. Allerdings muß darauf geachtet werden, daß Mister X sich keinen Startknoten mit einem Detektiv teilt:

⟨*generate_X_position* 143b⟩≡
```
procedure generate_X_position(rd start_positions);
begin
  repeat
    startposition ← random(start_positions);
  until startposition ∉ D_positions end repeat;
  return startposition;
end generate_X_position;
```
This code is used in chunks 136 and 209.

13.5 Die Funktion `load_board`

Setzt die globale Variable `board` und liefert die möglichen Startpositionen als Tupel zurück. Der Spielplan ist als Datei gespeichert, die zuerst das Tupel mit der internen Spielbrettrepräsentation enthält und an zweiter Stelle ein Tupel für die möglichen Startpositionen der Detektive und Mister X.

⟨*load_board* 144⟩≡
```
procedure load_board(rd name);
visible filename;
begin
  filename ← SCOTLAND_DIRECTORY + name + ".plan";
  fopen(filename, "r");
  fget(filename, board);
  fget(filename, start_positions);
  fclose(filename);
  -- Einlesen der Matrix: Anpassung nötig: Filename, Test auf Existenz
  -- In der Datei steht entweder das leere Tupel (Matrix nicht berechnet)
  -- oder die Matrix in zeilenweisem Aufbau.
  filename ← SCOTLAND_DIRECTORY + name + ".matrix";
  fopen (filename, "r") when io_file_error use create_matrix;
  matrix ← [];
  repeat
     fget (filename, t);
     matrix with← t;
  until (t = []) end repeat;
  matrix_laenge ← [];
  repeat
     fget (filename, t);
     matrix_laenge with← t;
  until (t = []) end repeat;
  fclose (filename);
  return(start_positions);

  ⟨Exceptionhandler: create_matrix 145a⟩
end load_board;
```
This code is used in chunks 136 and 208.

Die folgende Ausnahmebehandlung sorgt bei einer fehlenden Informationsdatei für einen Spielplan (die entsprechende Datei mit der Endung `.matrix`) dafür, daß eine solche Datei erstellt wird. Danach kann das Programm normal fortgeführt werden.

(In der aktuellen ProSet–Implementation wird die Ausnahme allerdings noch nicht erkannt.)

⟨*Exceptionhandler: create_matrix* 145a⟩≡
```
handler create_matrix();
visible matrix_anzahl_bus ← [];
begin
    ⟨berechne matrix und matrix_laenge 164⟩
    ⟨Abspeichern derselben 145b⟩
    fopen(filename, "r");

    ⟨init_matrix 165⟩
end create_matrix;
```
This code is used in chunk 144.

⟨*Abspeichern derselben* 145b⟩≡
```
        fopen (filename, "w");
        for i ∈ [1..max_knoten] do
                fput (filename, matrix(i));
        end for;
        fput (filename, []);
        for i ∈ [1..max_knoten] do
                fput (filename, matrix_laenge(i));
        end for;
        fput (filename, []);
        fclose(filename);
```
This code is used in chunk 145a.

13.6 Die Prozedur `start_strategy`

Erzeugt die Tupelräume für das Interface und für das Blackboard sowie die Prozesse für die Detektive und stellt die Daten für die Planung in den Tupelraum.

⟨*start_strategy* 146⟩≡

```
procedure start_strategy();
  hidden blackboard;
  hidden board;
  hidden matrix;
  hidden matrix_laenge;
  hidden taxi;
  hidden bus;
  hidden underground;
begin
  interface  ← CreateTS(Ω);
  blackboard ← CreateTS(Ω);
  store_TS_data(blackboard); -- Initialdaten fuer die Planung
  for nr ∈ [1..D_count] do
      || closure Start_Detective(nr, interface, blackboard);
  end for;
  ⟨store_TS_data 147⟩
end start_strategy;
```

This code is used in chunks 127 and 211.

Stellt die Daten für die Planungskomponente während der Initialisierung in den Tupelraum.

⟨*store_TS_data* 147⟩≡

```
procedure store_TS_data(rd blackboard);
begin
  deposit ["level", level] at blackboard end deposit;
  deposit ["put_debug"] at blackboard end deposit;
  deposit ["Resolve_Conflicts"] at blackboard end deposit;
  deposit ["board", board] at interface end deposit;
  deposit ["D_count", D_count] at interface end deposit;
  deposit ["level", level] at interface end deposit;
  deposit ["used_X_tickets", used_X_tickets]
       at interface end deposit;
  deposit ["last_public_X_position", last_public_X_position]
       at interface end deposit;
  deposit ["number_of_X_moves", X_moves]
       at interface end deposit;
  deposit ["X_moves_to_visible", X_moves_to_visible()]
       at interface end deposit;
  deposit ["number_of_D_moves", number_of_D_moves]
       at interface end deposit;
  deposit ["possible_X_positions", X_positions]
       at interface end deposit;
  deposit ["all_used_X_tickets", all_used_X_tickets]
       at interface end deposit;
  for nr ∈ [1..D_count] do
      deposit ["initial_D_position", nr, D_positions(nr)]
           at interface end deposit;
      deposit ["ticket_count", nr, D_tickets(nr)]
           at interface end deposit;
  end for;
  deposit ["ticket_count", 0, X_tickets]
       at interface end deposit;
end store_TS_data;
```

This code is used in chunks 146, 200, and 204.

13.7 Die Funktion play_the_game

In dieser Funktion befindet sich die Hauptschleife, die den Spielablauf organisiert. Nachdem die GUI den Zug von Mister X mitgeteilt hat, werden die Daten für die Planung im Tupelraum aktualisiert, sofern nicht die Nachricht game_over empfangen wurde, was zum Abbruch des Spiels führt. Ansonsten werden mögliche Anfragen der Planung solange verarbeitet, bis diese die Züge der Detektive bekanntgegeben hat. Danach werden diese der GUI mitgeteilt.

Falls sich Mister X soeben gezeigt hat, seine Position also genau bekannt ist, wird überprüft, ob die Detektive Mister X gefangen haben. Ist das der Fall, wird das Spiel beendet, und der GUI der Siegerdetektiv mitgeteilt.

Nach spätestens 24 Zügen wird das Spiel beendet, und der Gewinner an die GUI gemeldet.

⟨*play_the_game* 148⟩≡

```
procedure play_the_game();
begin
  Detectives_all_standing ← false;
  X_gewonnen ← false;
  repeat
    if X_moves = 23
       then doppelzug_erlaubt ← false;
       else doppelzug_erlaubt ← true;
    end if;
    move_of_X ← wait_for_X(doppelzug_erlaubt);
    if ("arrested" ∉ move_of_X) ∧ (¬ Detectives_all_standing)
       then update_TS_data(move_of_X);
            moves_of_D ← serve_strategy_requests();
            Detectives_all_standing ← ⟨all Detectives are standing 149a⟩;
            update_D_variables(moves_of_D);
            send_move_to_gui(moves_of_D);
            gewinner ← is_X_arrested();
            if gewinner ≠ Ω
               then move_of_X ← ["arrested", gewinner];
            end if;
       else X_gewonnen ← (("arrested" ∉ move_of_X)
                            ∧
                            Detectives_all_standing);
    end if;
  until ("arrested" ∈ move_of_X) ∨
        (X_moves = 24) ∨
        X_gewonnen
```

```
    end repeat;
    if ("arrested" ∈ move_of_X)
    then
      winner frome move_of_X;
      send_gui_message("game_over {" + winner + "}");
    elseif Detectives_all_standing
    then
      send_gui_message ("game_over {Alle Detektive sitzen fest:\n" +
                        "Mister X hat\n das Spiel gewonnen!}");
    else
      send_gui_message ("game_over {Mister X hat\n das Spiel gewonnen!}");
    end if;
    return (true);

    ⟨wait_for_X 150⟩
    ⟨update_TS_data 155⟩
    ⟨serve_strategy_requests 157⟩
    ⟨update_D_variables 179b⟩
    ⟨send_move_to_gui 180c⟩
  end play_the_game;
```

This code is used in chunk 127.

Folgender Ausdruck liefert `true`, wenn kein Detektiv ziehen kann (`I am still standing!`).

⟨*all Detectives are standing* 149a⟩≡

```
  #[x(1) : x ∈ moves_of_D | x(3) = "I am still standing!" ] = D_count
```

This code is used in chunk 148.

13.7.1 Die Behandlung der Ausnahme `game_over`

Diese Funktion zur Behandlung der Ausnahme `game_over` sorgt dafür, daß die Prozedur `play_the_game()` bzw. `init()` verlassen wird. Je nach übergebenem Argument wird das Spiel dann neu gestartet oder das Programm beendet.

⟨*Exceptionhandler: game_over* 149b⟩≡

```
  handler game_over(new_game);
  begin
    return (new_game);
  end game_over;
```

This code is used in chunks 127, 216, 220, and 221.

13.7.2 Die Funktion wait_for_X

Diese Funktion wartet im wesentlichen auf die Nachricht "X_move" von der GUI und liefert das/die von Mister X verwendete(n) Ticket(s) zurück. Gemäß dem Nachrichtenprotokoll zwischen der GUI und dem Treiber wird vorher zusätzlich die Erlaubnis zur Durchführung eines Doppelzuges und die Anzahl der Züge, bis sich Mister X zeigen muß, übermittelt. Eventuell wird dann noch eine Nachricht mit der Information über den aktuellen Standort von Mister X empfangen.

Zeigt sich Mister X an einer Position, an der ein Detektiv steht, oder kann Mister X sich nicht mehr bewegen (d.h. in der nächsten Runde würde er gefangen), so ist das Spiel beendet und die Prozedur liefert ein spezielles Tupel ["arrested", meldung] mit einer Meldung für den Benutzer zurück.

⟨*wait_for_X* 150⟩≡

```
procedure wait_for_X(rd doppelzug_erlaubt);
begin
  if doppelzug_erlaubt
     then send_gui_message("X_move on");
     else send_gui_message("X_move off");
  end if;
  repeat
    message ← get_gui_message(); -- Message X˙move
  until message ≠ "" end repeat;
  ⟨DEBUG_wait_for_X_1 187⟩
  if message = "X_move twice"
     then ⟨Doppelzug ausfuehren 151a⟩
     else ⟨DEBUG_wait_for_X_4 189a⟩
          X_move ← [message(8..)];
          ⟨Hat jemand das Spiel gewonnen? 151b⟩
          ⟨Aktualisierung der Variablen fuer X 152⟩
          if (X_moves > 3)
             then ⟨send_X_positions_to_gui 181a⟩
          end if;
          send_gui_message("X_show " +
                           capsulated_str(X_moves_to_visible()));
          if (X_moves_to_visible() = 0)
             then ⟨X zeigt sich 153⟩
                  ⟨send_X_positions_to_gui 181a⟩
          end if;
          return (X_move);
  end if;

  ⟨Ticketvorrat von X aktualisieren 154⟩
```

```
  end wait_for_X;
```

This code is used in chunks 148 and 216.

Bei einem Doppelzug wird die Funktion zweimal rekursiv aufgerufen und die Züge dann gemeinsam zurückgeliefert.

⟨*Doppelzug ausfuehren* 151a⟩≡

```
            ⟨DEBUG_wait_for_X_3 188b⟩
            update_X_tickets("twice");
            X_move ← wait_for_X(false);
            send_X_tickets();
            X_move +← wait_for_X(false);
            return (X_move);
```

This code is used in chunk 150.

Im folgenden Abschnitt wird anhand der möglichen Positionen für Mister X geprüft, ob er das angegebene Ticket überhaupt benutzen konnte. Hat Mister X eine nicht bestehende Verbindung befahren wollen, oder ist er von einem der Detektive gefangen worden, so wird ein entsprechendes Tupel von der Funktion zurückgeliefert.

⟨*Hat jemand das Spiel gewonnen?* 151b⟩≡

```
    --- ACHTUNG: Seiteneffekt = X_positions wird aktualisiert
    if possible_X_positions(X_move) = ∅
       then return (["arrested",   "Mister X hat unzulaessig\n"
                                  + "gezogen und hat damit\n"
                                  + "leider verloren!"]);
    end if;
    gewinner ← is_X_arrested();
    if gewinner ≠ Ω
       then return (["arrested", gewinner]);
    end if;
```

This code is used in chunk 150.

Nach dem Zug von Mister X werden verschiedene globale Variablen auf den neuesten Stand gebracht: Die Menge möglicher Positionen für Mister X die Gesamtzahl der Züge, seit dem letzten Auftauchen gemachte Züge und deren Anzahl, die Menge aller Züge von Mister X und seine Ticketvorräte.

⟨*Aktualisierung der Variablen fuer X* 152⟩≡

```
X_positions - := x: x in D_positions;
X_moves ← X_moves + 1;
if number_of_X_moves = Ω
   then number_of_X_moves ← 1;
   else number_of_X_moves + ← 1;
end if;
used_X_tickets + ← X_move;
all_used_X_tickets + ← X_move;
update_X_tickets(message(8..));
```

This code is used in chunk 150.

Muß sich Mister X zeigen, so wird seine von der GUI übermittelte aktuelle Position entgegengenommen und ist dann auch die einzige mögliche Position. Zusätzlich wird geprüft, ob sich ein Detektiv auf eben dieser Position befindet und damit Mister X gefangen wurde. Schließlich werden die Variablen für die seit dem letzten Auftauchen verwendeten Tickets von Mister X und die Anzahl der seitdem gemachten Züge für Mister X und die Detektive zurückgesetzt.

⟨*X zeigt sich* 153⟩≡

```
repeat
  message ← get_gui_message();
until message ≠ "" end repeat;
⟨DEBUG_wait_for_X_2 188a⟩
last_public_X_position ← string2integer(message(12..));
-- Ist Mister X gefangen worden?
for nr ∈ [1..D_count] do
    if (D_positions(nr) = last_public_X_position)
       then return (["arrested", "Mister X ist von Detektiv "
                    + capsulated_str(nr) + " gefangen worden!"]);
    end if;
end for;
-- Falsch gezeigt?
if (¬ (last_public_X_position ∈ X_positions)) then
   return (["arrested", "Mister X hat sich an einer\n"
                      + "unmoeglichen Position gezeigt!\n"
                      + "Mit Schummlern spielen\n"
                      + "wir nicht!\n\n"
                      + "LEIDER VERLOREN!"]);
end if;
-- Korrekt gezeigt und nicht gefangen
X_positions ← –last_public_X_position″;
used_X_tickets ← [];
number_of_X_moves ← 0;
number_of_D_moves ← 0;
```

This code is used in chunk 150.

Die nächste Prozedur vermindert den Vorrat an Tickets der angegeben Art von Mister X um Eins.

⟨*Ticketvorrat von X aktualisieren* 154⟩≡

```
procedure update_X_tickets(rd ticket);
begin
  for i ∈ [1..#X_tickets] do
      if X_tickets(i)(1) = ticket
         then
            ⟨DEBUG_wait_for_X_5 189b⟩
              X_tickets(i)(2) ← X_tickets(i)(2) - 1;
      end if;
  end for;
end update_X_tickets;
```

This code is used in chunk 150.

13.7.3 Die Prozedur update_TS_data

Die nächste Prozedur aktualisiert die Daten im Tupelraum für die Planungskomponente. Folgende Daten werden geändert: Letzte Position, an der sich Mister X zeigte, von Mister X bisher benutzte Tickets seit dem letzten Auftauchen und insgesamt, Anzahl der Züge von Mister X sowie den Detektiven nach dem letzten Auftauchen, Ticketvorräte von Mister X und den Detektiven, mögliche Positionen von Mister X die Anzahl der Züge bis zum nächsten Auftauchen von Mister X und der letzte Zug von Mister X.

⟨*update_TS_data* 155⟩≡

```
procedure update_TS_data(rd current_X_move);
begin
  fetch ("used_X_tickets", ? ) at interface end fetch;
  deposit ["used_X_tickets", used_X_tickets]
       at interface end deposit;

  fetch ("last_public_X_position", ? )
          ⇒ deposit ["last_public_X_position", last_public_X_position]
                  at interface end deposit;
    xor ("last_public_X_position")
          ⇒ deposit ["last_public_X_position", last_public_X_position]
                  at interface end deposit;
     at interface
  end fetch;

  fetch ("number_of_X_moves", ? )
          ⇒ deposit ["number_of_X_moves", X_moves]
                  at interface end deposit;
    xor ("number_of_X_moves")
          ⇒ deposit ["number_of_X_moves", X_moves]
                  at interface end deposit;
    at interface
  end fetch;

  fetch ("number_of_D_moves", ? )
          ⇒ deposit ["number_of_D_moves", number_of_D_moves]
                  at interface end deposit;
    xor ("number_of_D_moves")
          ⇒ deposit ["number_of_D_moves", number_of_D_moves]
                  at interface end deposit;
    at interface
  end fetch;
```

```
    fetch ("all_used_X_tickets", ? ) at interface end fetch;
    deposit ["all_used_X_tickets", all_used_X_tickets]
         at interface end deposit;

    for nr ∈ [1..D_count] do
        fetch ("ticket_count", nr, ? ) at interface end fetch;
        deposit ["ticket_count", nr, D_tickets(nr)]
             at interface end deposit;
    end for;

    fetch ("ticket_count", 0, ? ) at interface end fetch;
    deposit ["ticket_count", 0, X_tickets]
         at interface end deposit;

    fetch ("possible_X_positions", ? ) at interface end fetch;
    deposit ["possible_X_positions", X_positions]
         at interface end deposit;

    fetch ("X_moves_to_visible", ? ) at interface end fetch;
    deposit ["X_moves_to_visible",
             if X_moves = 24 then Ω else X_moves_to_visible() end if]
         at interface end deposit;

    ⟨DEBUG_update_TS_data 190a⟩

    for nr ∈ [1..D_count] do
      deposit ["current_X_move", nr, current_X_move]
           at interface end deposit;
    end for;
  end update_TS_data;
```

This code is used in chunks 148 and 204.

13.7.4 Die Funktion serve_strategy_requests und ihre Unterprogramme

Die folgende Prozedur wartet auf Anfragen der Planungskomponente im Tupelraum und bearbeitet sie. Folgende Anfragen werden erkannt: `D_position`, `possible_moves` und `shortest_path`. Wenn alle Detektive gezogen haben, werden die Züge überprüft und, falls korrekt, an die aufrufende Prozedur zurückgeliefert. Ansonsten werden die Detektive über den Tupelraum über den Fehler informiert und es wird auf weitere Anfragen gewartet.

Aus der Variablen `detectives` werden die Nummern der Detektive entfernt, die ihren Zug bereits gemeldet haben. Die Variable `detectives_info` speichert die von diesen Detektiven gelieferten Daten, um sie für die Korrektheitsprüfung der Züge und die nachfolgende Rückmeldung an die Detektive verwenden zu können.

⟨*serve_strategy_requests* 157⟩≡

```
procedure serve_strategy_requests();
  visible detectives      ← -1..D_count";
  visible detectives_info ← [];
  hidden  moves_of_D      ← [];
begin
  repeat
    repeat
      fetch ("D_position", ? id, ? detective, ? node, ? ticket)
              ⇒ put ("DRIVER: Got Information D_Position, " +
                     "detective, node, ticket, id:",
                     detective, node, ticket, id);
                register_detectives(id, detective, node, ticket);
        xor ("possible_moves", ? id, ? node)
              ⇒ put ("DRIVER: Got Request possible_moves, id:", id);
                deposit [id, possible_moves(node)]
                     at interface end deposit;
        xor ("shortest_path", ? id, ? start, ? ende, ? tickets)
              ⇒ deposit [id, shortest_path(start, ende, tickets)]
                     at interface end deposit;
        xor ("path_sum", ? id, ? start, ? ziele)
              ⇒ deposit [id, path_sum(start, ziele)]
                     at interface end deposit;
      at interface end fetch;
    until detectives = ∅ end repeat;
    ⟨Detektivzuege zusammenstellen 158a⟩
    ⟨Ueberpruefung der Zuege 158b⟩
  until (idiots = ∅) end repeat;
  return (moves_of_D);
```

```
    ⟨register_detectives 159a⟩
    ⟨possible_moves 159b⟩
    ⟨shortest_path 163b⟩
    ⟨path_sum 160⟩
    ⟨D_moves_correct 161⟩
  end serve_strategy_requests;
```

This code is used in chunks 148 and 213.

In der folgenden Schleife wird das Ergebniss der Funktion zusammengestellt. Dieses muß allerdings erst auf Korrektheit getestet werden (s.u.).

⟨*Detektivzuege zusammenstellen* 158a⟩≡

```
      for nr ∈ [1..D_count] do
          moves_of_D with ← [nr,
                               detectives_info(nr)(1),  -- Knotennummer
                               detectives_info(nr)(2)]; -- Ticket
      end for;
```

This code is used in chunk 157.

Jeder Zug wird überprüft und jeder Detektiv erhält eine Mitteilung darüber, ob sein Zug korrekt war.

⟨*Ueberpruefung der Zuege* 158b⟩≡

```
      idiots ← D_moves_correct(moves_of_D);
  put("Idioten:", idiots);
      for nr ∈ [1..D_count] do
          if (nr ∈ idiots)
            then
  put ("Detektiv ",nr," wird benachrichtigt (falscher Zug)");
              deposit [detectives_info(nr)(3), false]
                      at interface end deposit;
            else
  put ("Detektiv ",nr," wird benachrichtigt (korrekter Zug)");
               deposit [detectives_info(nr)(3), true]
                       at interface end deposit;
          end if;
      end for;
```

This code is used in chunk 157.

Im folgenden werden die von `serve_strategy_requests()` verwendeten Prozeduren beschrieben.

Die folgende Unterroutine vermerkt die erhaltenen Züge und Daten der Detektive in den Variablen `detectives` und `detectives_info`, die lokal zur aufrufenden Prozedur `serve_strategy_requests()` sind.

⟨*register_detectives* 159a⟩≡

```
procedure register_detectives(rd id, rd detective, rd node, rd ticket);
begin
  detectives less ← detective;
  detectives_info(detective) ← [node, ticket, id];
end register_detectives;
```

This code is used in chunk 157.

Die Funktion `possible_moves` berechnet für die angegebene Position alle möglichen Orte, die von dieser Position in einem Zug mit diesem Ticket zu erreichen sind. Diese werden dann als Tupel zurückgegeben.

⟨*possible_moves* 159b⟩≡

```
procedure possible_moves (startposition);
begin
   antwort ← [];
   for ticket ∈ ["taxi", "bus", "underground", "black"] do
      zwischen ← [x(2) : x ∈ board | x(1) = ticket](1);
      if zwischen ≠ Ω
      then
         if (startposition ∈ domain (zwischen))
         then
            antwort +← [[x,ticket] : x ∈ zwischen(startposition)];
         end if;
      end if;
   end for;
   return (antwort);
end possible_moves;
```

This code is used in chunk 157.

Berechnen einer Abstandssumme von Knoten `start` zu einer Menge von Knoten (`ziele`) anhand der Matrix `matrix_laenge`.

⟨*path_sum* 160⟩≡

```
procedure path_sum (start,ziele);
begin
   sum ← 0;
   for i ∈ ziele do
      sum +← if (start > i)
               then
                  matrix_laenge(start)(i)
               elseif (start = i)
               then
                  0
               else
                  matrix_laenge(i)(start)
               end if;
   end for;
   return sum;
end path_sum;
```

This code is used in chunk 157.

Die folgende Funktion überprüft die Korrektheit aller von der Planungskomponente gemeldeten Züge der Detektive. Zurückgeliefert wird die Menge der Nummern derjenigen Detektive, die falsch gezogen haben.

⟨*D_moves_correct* 161⟩≡
```
  procedure D_moves_correct(rd moves);
    hidden wrong_D ← ∅;
    hidden new_D_positions ← [];
  begin
    put ("DRIVER: D_moves_correct: moves = ", moves);
    for nr ∈ [1..D_count] do
  put("Detektiv:", nr);
        if moves(nr)(3) = "I am still standing!"
            then ⟨Darf der Detektiv stehenbleiben? 162a⟩
            else ⟨Ist der Zielknoten erreichbar? 162b⟩
                   ⟨Ist das Ticket verfuegbar? 162c⟩
                   ⟨Ist der Zielknoten frei? 163a⟩
        end if;
    end for;

    return (wrong_D);

  end D_moves_correct;
```
This code is used in chunks 157 and 195.

⟨*Darf der Detektiv stehenbleiben?* 162a⟩≡
```
reachable ← ∅;
for i ∈ [1..3] do
   if (D_tickets(nr)(i)(2) > 0)
   then
      zwischen ← board(i)(2)(D_positions(nr));
      if (zwischen ≠ Ω)
      then
         reachable + ← –x : x ∈ zwischen";
      end if;
   end if;
end for;
reachable ← reachable - ( x : x in new_D_positions +
                          –x : x ∈ D_positions(nr+1..)" );
if (reachable ≠ ∅)
then
   wrong_D with ← nr;
end if;
```
This code is used in chunk 161.

Für jeden Zug wird überprüft, ob der angefahrene Knoten überhaupt existiert, und ob er mit dem verwendeten Ticket zu erreichen ist:

⟨*Ist der Zielknoten erreichbar?* 162b⟩≡
```
possible_destinations ←
   [y(2): y ∈ board | (y(1) = moves(nr)(3)) ](1)(D_positions(nr));
if (possible_destinations = Ω)
then
   wrong_D with ← nr;
elseif
   (moves(nr)(2) ∉ possible_destinations)
then
   wrong_D with ← nr;
end if;
```
This code is used in chunk 161.

Nachfolgend wird geprüft, ob der Detektiv noch über ein Ticket der verwendeten Sorte verfügt:

⟨*Ist das Ticket verfuegbar?* 162c⟩≡
```
put("Ist das Ticket verfuegbar?");
      if ([x(2): x ∈ D_tickets(nr) | (x(1) = moves(nr)(3)) ](1) < 1)
          then wrong_D with ← nr;
      end if;
```
This code is used in chunk 161.

Zusätzlich wird geprüft, ob kein anderer Detektiv auf dem Zielknoten steht. Dazu müssen alle Detektive, die noch ziehen, und alle die schon gezogen haben, auf anderen Knoten stehen:

⟨*Ist der Zielknoten frei?* 163a⟩≡
```
if ( moves(nr)(2) ∈ D_positions(nr..) ) ∨
   ( moves(nr)(2) ∈ new_D_positions )
then
   wrong_D with ← nr;
   new_D_positions with ← D_positions(nr);
else
   new_D_positions with ← moves(nr)(2);
end if;
```
This code is used in chunk 161.

Die Funktion `shortest_path` und ihre Unterprogramme

Die Funktion `shortest_path` liefert (meistens) den kürzesten Weg zwischen zwei Knoten unter Berücksichtigung der übergebenen Ticketvorräte. Dabei werden Taxitickets gegenüber Bustickets bevorzugt, wenn dadurch der Weg nicht länger wird. Zurückgegeben wird der Weg als Tupel von Paaren mit Knoten und dem Ticket, mit dem dieser Knoten angefahren wird. Der Startknoten kommt in dem Tupel nicht mehr vor.

⟨*shortest_path* 163b⟩≡
```
procedure shortest_path (von,nach,tickets);
visible busknoten ← domain(bus);
visible undergroundknoten ← domain(underground);
begin
   return (verkuerze_weg(path (von,nach),tickets));
   ⟨verkuerze_weg 171⟩
   ⟨delete_components 178b⟩
   ⟨path 179a⟩
end shortest_path;
```
This code is used in chunk 157.

Wenn die Kürzeste-Pfade-Matrix der Taxistrecken noch leer ist, wird sie jetzt berechnet. Es wird aus Effizienzgrunden kein Shortest-Path auf dem gesamten Scotland-Yard-Graphen durchgeführt, sondern nur auf dem Taxigraphen. Der verwendete Algorithmus entspricht bis auf einige Modifikationen der Idee der Dynamischen Programmierung. Siehe z.B. Ingo Wegener "Effiziente Algorithmen„ (Skript zur gleichnamigen Vorlesung, Dortmund, Informatik LS 2). Die weitere Verkürzung der Wege in der Funktion `verkuerze_weg` basiert auf der Idee, daß Bus- und U-Bahnstrecken, die Taxiwege, auf denen sie liegen, verkürzen. Um bei gleichlangen Taxiwegen, die mit den meisten Busknoten zu selektiern – auf ihnen liegen wahrscheinlich mehr Busstrecken –, wird die Anzahl der Busknoten auf den Strecken ebenfalls abgespeichert. Es werden drei Matrizen gebildet, in denen eine für die Speicherung der Wege, die zweite für die Länge der Wege und die dritte für die Anzahl der Busknoten zuständig ist. Die beiden letztgenannten sind nur untere Dreiecksmatrizen, da sie symmetrisch zur Hauptdiagonalen sind. In der Wegematrix wird der kürzeste Weg nicht komplett abgelegt, sondern immer nur der Nachfolger auf dem Weg zum Endknoten. `matrix_laenge` wird anschließen nochmal aktualisiert, so daß in ihr die kürzesten Entfernungen zwischen zwei Knoten gespeichert sind.

⟨*berechne* `matrix` *und* `matrix_laenge` 164⟩≡
```
init_matrix();
for l ∈ [1..MAX_KNOTEN] do
   update (l);
end for;
update_matrix_laenge();
```
This code is used in chunk 145a.

Hier wird die Matrix initialisiert, in der die kürzesten Taxiwege zwischen zwei Knoten abgespeichert werden. In Zeile a und Spalte b wird der Wert b eingetragen, wenn b von a mit genau einem Taxiticket erreicht werden kann, ansonsten 0. Zusätzlich werden zwei Untere-Dreiecks-Matrizen aufgebaut, in denen die Länge des Weges (initialisiert mit 0 oder 1) und die Anzahl der Busknoten auf dem Weg (initialisiert mit 0, 1 oder 2) abgelegt werden.

⟨*init_matrix* 165⟩≡

```
procedure init_matrix();
begin
   matrix ← [];
   matrix_laenge ← [];
   for x ∈ [1..MAX_KNOTEN] do
      zeile_matrix ← [];
      zeile_laenge ← [];
      zeile_anzahl_bus ← [];
      x_wege ← taxi(x);
      if (x_wege = Ω)
         then x_wege ← [];
         end if;
      if x ∈ busknoten
         then x_bus_knoten ← 1;
         else x_bus_knoten ← 0;
         end if;
      for y ∈ [1 .. x] | (y ≠ x) do
         if y ∈ busknoten
            then anzahl_bus ← x_bus_knoten + 1;
            else anzahl_bus ← x_bus_knoten;
            end if;
         if (y ∈ x_wege)
         then
            zeile_matrix with← y;
            matrix(y) ← matrix(y) with x;
            zeile_laenge with← 1;
            zeile_anzahl_bus with← anzahl_bus;
         else
            zeile_matrix with ← 0;
            matrix(y) ← matrix(y) with 0;
            zeile_laenge with← 0;
            zeile_anzahl_bus with ← 0;
         end if;
      end for;
```

```
        matrix with← zeile_matrix with 0;
        matrix_laenge with← zeile_laenge with 0;
        matrix_anzahl_bus with ← zeile_anzahl_bus with 0;
    end for;
    ⟨update 167⟩
    ⟨update_matrix_laenge 169⟩
  end init_matrix;
```

This code is used in chunk 145a.

In `update` werden die Einträge aller drei Matrizen geändert, wenn der Weg über den Knoten `l` kürzer ist als der bisher kürzeste Weg, der Knoten `l` nicht betritt.

⟨*update* 167⟩≡

```
procedure update(l);
begin
   if (l ∈ busknoten)
      then l_ist_bus ← 1;
      else l_ist_bus ← 0;
   end if;
   for x ∈ [1..MAX_KNOTEN] | (x ≠ l) do
      x_nach_l ← matrix(x)(l);
      if (x_nach_l ≠ 0)
      then
         for y ∈ [1..x] | ((x ≠ y) ∧ (y ≠ l)) do
            l_nach_y ← matrix(l)(y);
            y_nach_l ← matrix(y)(l);
            if (l_nach_y ≠ 0)
            then
               alt_laenge ← matrix_laenge(x)(y);
               alt_anzahl_bus ← matrix_anzahl_bus(x)(y);
               if (x > l)
               then
                  neu_laenge ← matrix_laenge(x)(l);
                  neu_anzahl_bus ← matrix_anzahl_bus(x)(l);
               else
                  neu_laenge ← matrix_laenge(l)(x);
                  neu_anzahl_bus ← matrix_anzahl_bus(l)(x);
               end if;
               if (y > l)
               then
                  neu_laenge +← matrix_laenge(y)(l);
                  neu_anzahl_bus +← matrix_anzahl_bus(y)(l);
               else
                  neu_laenge +← matrix_laenge(l)(y);
                  neu_anzahl_bus +← matrix_anzahl_bus(l)(y);
               end if;
               if besser (neu_laenge,neu_anzahl_bus,
                          alt_laenge,alt_anzahl_bus)
               then
                  matrix(x)(y) ← x_nach_l;
                  matrix_laenge(x)(y) ← neu_laenge;
```

```
                    matrix_anzahl_bus(x)(y) ← neu_anzahl_bus;
                    -- oberhalb der Hauptdiagonalen
                    matrix(y)(x) ← y_nach_l;
                end if;
            end if;
        end for;
    end if;
end for;

⟨besser 168⟩
end update;
```
This code is used in chunk 165.

`besser` vergleicht die zwei Wege bezüglich der Länge und der Anzahl der Busknoten auf dem Weg. Ist der erste Weg kürzer oder ist er genauso lang wie der zweite, hat aber mehr Busknoten, liefert `besser` den Wert `true`, ansonsten `false`.

⟨*besser* 168⟩≡
```
procedure besser (neu_laenge,neu_anzahl_bus,alt_laenge,alt_anzahl_bus);
begin
    if (alt_laenge ≠ 0)
    then
        if (neu_laenge < alt_laenge)
            then return true;
        elseif (neu_laenge > alt_laenge)
            then return false;
        elseif (neu_anzahl_bus ≤ alt_anzahl_bus)
        -- Strecken gleich lang : neu˙laenge = alt˙laenge
            then return false;
            else return true;
        end if;
    else
        return true;
    end if;
end besser;
```
This code is used in chunk 167.

Hier wird die Entfernungsinformation in `matrix_laenge` durch Hinzunahme von Bus- und U-Bahn-Strecken erweitert. Dadurch ist in `matrix_laenge` die Länge des kürzesten Weges zwischen allen Knoten gespeichert. Diese Informationen werden in der Funktion `path_sum` benötigt.

⟨*update_matrix_laenge* 169⟩≡

```
procedure update_matrix_laenge();
begin
   for x ∈ domain(underground) do
      for y ∈ underground(x) do
         if (x > y)
            then matrix_laenge(x)(y) ← 1;
            else matrix_laenge(y)(x) ← 1;
         end if;
      end for;
   end for;
   for x ∈ domain(bus) do
      for y ∈ bus(x) do
         if (x > y)
            then matrix_laenge(x)(y) ← 1;
            else matrix_laenge(y)(x) ← 1;
         end if;
      end for;
   end for;
   for l ∈ domain(bus)+domain(underground) do
      update_laenge(l);
   end for;

procedure update_laenge(l);
begin
   for x ∈ [1..max_knoten] | (x ≠ l) do
      if (x > l)
         then laenge_x_l ← matrix_laenge(x)(l);
         else laenge_x_l ← matrix_laenge(l)(x);
      end if;
      if (laenge_x_l > 0) then
        for y ∈ [1..x] | ((x ≠ y) ∧ (y ≠ l)) do
           alt_laenge ← matrix_laenge(x)(y);
           if ((laenge_x_l < alt_laenge) ∨ (alt_laenge = 0)) then
              if (y > l)
                 then laenge_l_y ← matrix_laenge(y)(l);
                 else laenge_l_y ← matrix_laenge(l)(y);
```

```
                end if;
                if ((laenge_l_y > 0)
                    ∧
                   (((laenge_x_l + laenge_l_y) < alt_laenge)
                     ∨
                    (alt_laenge = 0)))
                   then
                     matrix_laenge(x)(y) ← laenge_x_l + laenge_l_y;
                end if;
             end if;
          end for;
       end if;
     end for;
 end update_laenge;

 end update_matrix_laenge;
```

This code is used in chunk 165.

Die nächste Funktion verkürzt die Wege durch Hinzunahme von Bus- und U-Bahnstrecken. Diese Strecken werden berechnet, dann werden zuerst U-Bahnstrecken eingebunden, danach Busstrecken. Bei den Ersetzungen hat immer die längste Strecke Vorrang. Es werden nur soviel Verbindungen ersetzt, wie der übergebene Ticketvorrat zuläßt. Busstrecken der Länge 1 werden nur verwendet, wenn keine Taxitickets mehr vorhanden sind. Bustickets werden, bei gleicher Verbesserung, den U-Bahntickets vorgezogen.

```
⟨verkuerze_weg 171⟩≡
 procedure verkuerze_weg (weg,tickets);
 begin
    -- bisher benötigte taxi-tickets
    taxi_moves          ← #weg - 1;
    -- ticketvorräte
    taxi_tickets        ← tickets ("taxi");
    bus_tickets         ← tickets ("bus");
    underground_tickets ← tickets ("underground");
    -- weg_neu := weg mit ticket-art und endemarkierung
    weg_neu ← [[x,"taxi"] : x ∈ weg]  with ["ende"];
    ⟨berechnen der u-bahn-strecken auf weg 172a⟩
    ⟨strecken auf weg_neu mit u-bahn verbessern 172b⟩
    ⟨weitere verbesserungen von weg_neu mit u-bahn 173⟩
    ⟨berechnen der bus-strecken auf weg_neu 175a⟩
    ⟨strecken auf weg_neu mit bus verbessern 175b⟩
    ⟨weitere verbesserungen von weg_neu mit bus 176⟩
    -- löschen der ende-markierung und des start-knotens
    weg_neu ← weg_neu (2..(#weg_neu - 1));
    if (taxi_moves > taxi_tickets)
    then
       -- ticketvorrat reichte nicht aus
       return ([]);
    else
       return (weg_neu);
    end if;
    ⟨longest_path 177⟩
    ⟨position_in 178a⟩
 end verkuerze_weg;
```
This code is used in chunk 163b.

Aus den U-Bahn-Knoten auf `weg` die möglichen U-Bahnstrecken berechnen:

⟨*berechnen der u-bahn-strecken auf* `weg` 172a⟩≡
```
u ← [x : x ∈ weg | x ∈ undergroundknoten];
u_strecken ← [];
if (#u > 1)
then
  for i ∈ [1 .. (#u-1)] do
     if (u(i+1) ∈ underground(u(i)))
        then u_strecken with ← [u(i), u(i+1), #path (u(i), u(i+1)) - 1];
     end if;
  end for;
end if;
```
This code is used in chunk 171.

Bis keine U-Bahn-Tickets oder -Strecken mehr vorhanden sind, werden Folgen von Taxifahrten durch eine U-Bahn-Fahrt ersetzt.

⟨*strecken auf* `weg_neu` *mit u-bahn verbessern* 172b⟩≡
```
while ((underground_tickets > 0) ∧ (u_strecken ≠ [])) do
   max_index ← longest_path (u_strecken);
   -- löschen der wege, die ersetzt werden
   von ← position_in (u_strecken(max_index)(1),weg_neu);
   nach ← position_in (u_strecken(max_index)(2),weg_neu);
   weg_neu(von+1..nach) ← [[u_strecken(max_index)(2), "underground"]];
   -- anpassen von anzahl der benötigten taxi-tickets
   taxi_moves -:= u_strecken(max_index)(3);
   -- löschen der benutzten u-bahn-strecke
    u_strecken ← delete_components (u_strecken,max_index,max_index);
    underground_tickets -:= 1;
end while;
```
This code is used in chunk 171.

Wenn noch U-Bahn-Tickets vorhanden sind, wird versucht, den Weg

- durch „Rückschritte“ zu einem U-Bahn-Knoten
- durch „Zuweitfahren“ zu einem U-Bahn-Knoten

zu verbessern.

⟨*weitere verbesserungen von* weg_neu *mit u-bahn* 173⟩≡

```
-- nur möglich, wenn u-tickets vorhanden und mindestes ein u-knoten auf weg_neu
if (underground_tickets > 0) ∧ (#u > 0)
then
   laenge_anfang ← position_in (u(1),weg_neu);
   vom_start ← [x : x ∈ [[[x,"taxi"]
                   : x ∈ path(weg(1),y)]
                   : y ∈ undergroundknoten]
                 | #x < laenge_anfang];
   if (#vom_start > 0)
   then
      ab ← delete_components (weg_neu, 1, laenge_anfang);
      zum_ersten_u_knoten ← [[[u(1), "underground"]]];
      if (underground_tickets > 1)
      then
         zum_ersten_u_knoten +←
            [[[x,"underground"],[u(1),"underground"]]
             : x ∈ erreichbar(u(1),"underground")];
      end if;
      neue_wege ←
         [x + y + ab : x ∈ vom_start, y ∈ zum_ersten_u_knoten
          | x(#x)(1) ∈ underground(y(1)(1))];
      if (#neue_wege > 0)
      then
         -- weg_neu wird der beste (= kürzeste) weg aus neue_wege zugewiesen
         for i ∈ neue_wege do
           if (#weg_neu > #i) then weg_neu ← i; end if;
         end for;
         -- underground_tickets anpassen
         underground_tickets ←   tickets ("underground")
                               - #[x(1) : x in weg_neu(2..)
                                   | x(2) = "underground"];
      end if;
   end if;
end if;
if (underground_tickets > 0) ∧ (#u > 0)
```

```
then
   pos ← position_in (u(#u),weg_neu);
   laenge_ende ← #weg_neu - pos - 1;
   vom_ende ← [x : x ∈ [[[x,"taxi"]
                  : x ∈ path(y,weg(#weg))]
                  : y ∈ undergroundknoten]
                | #x < laenge_ende];
   if (#vom_ende > 0)
   then
      bis ← delete_components (weg_neu, pos, #weg_neu);
      ticket_typ ← weg_neu(pos)(2);
      zum_letzten_u_knoten ← [[[u(#u), ticket_typ]]];
      if (underground_tickets > 1)
      then
         zum_letzten_u_knoten +←
            [[[u(#u),ticket_typ],[x,"underground"]]
             : x ∈ erreichbar(u(#u),"underground")];
      end if;
      neue_wege ←
         [bis + x + [[y(1)(1), "underground"]] + y(2..) with ["ende"]
          : x ∈ zum_letzten_u_knoten, y ∈ vom_ende
          | x(#x)(1) ∈ underground(y(1)(1))];
      -- weg_neu wird der beste (= kürzeste) weg aus neue_wege zugewiesen
      for i ∈ neue_wege do
        if (#weg_neu > #i) then weg_neu ← i; end if;
      end for;
  end if;
end if;
-- taxi_moves anpassen
taxi_moves ← #[x(1) : x ∈ weg_neu(2..) | x(2) = "taxi"];
```

This code is used in chunk 171.

Berechnen der Bus-Strecken auf dem verbesserten Weg. Dazu werden erst der Reihe nach alle Knoten auf `weg_neu` bestimmt, die Bus-Knoten sind. Anschließend werden die Strecken, die vor der Verbesserung schon Bus-Strecken waren, und deren Start- und Endknoten noch auf der verbesserten Strecke liegen, übernommen.

⟨*berechnen der bus-strecken auf* `weg_neu` 175a⟩≡
```
b ← [x(1) : x ∈ weg_neu(1..#weg_neu - 1) | x(1) in busknoten];
b_strecken ← [];
if (#b > 1)
then
  for i ∈ [1 .. (#b-1)] do
     if (b(i+1) ∈ bus(b(i)))
        then b_strecken with ← [b(i), b(i+1), #path (b(i), b(i+1)) - 1];
     end if;
  end for;
end if;
```
This code is used in chunk 171.

Ersetzen von Taxi-Strecken auf `weg_neu` durch Bus-Strecken, bis keine Bus-Tickets oder Bus-Strecken mehr vorhanden sind. Bus-Strecken der Länge eins nur ersetzen, wenn der Vorrat an Taxi-Tickets nicht ausreicht.

⟨*strecken auf* `weg_neu` *mit bus verbessern* 175b⟩≡
```
if (b_strecken ≠ [])
   then max_index ← longest_path (b_strecken);
end if;
while ((bus_tickets > 0) ∧ (b_strecken ≠ []) ∧
       ((b_strecken(max_index)(3) > 1) ∨
        (taxi_moves > taxi_tickets))) do
   -- löschen der wege, die ersetzt werden und einfügen der bus-strecke
   von ← position_in (b_strecken(max_index)(1),weg_neu);
   nach ← position_in (b_strecken(max_index)(2),weg_neu);
   weg_neu(von+1..nach) ← [[b_strecken(max_index)(2), "bus"]];
   -- anpassen der anzahl der benötigten taxi-tickets
   taxi_moves -:= b_strecken(max_index)(3);
   -- löschen der benutzten bus-strecke
   b_strecken ← delete_components (b_strecken,max_index,max_index);
   bus_tickets -:= 1;
   if (b_strecken ≠ [])
      then max_index ← longest_path (b_strecken);
   end if;
end while;
```
This code is used in chunk 171.

Wenn noch Bus-Tickets vorhanden sind, wird versucht, den Weg

- durch „Rückschritte“ zu einem Bus-Knoten
- durch „Zuweitfahren“ zu einem Bus-Knoten

zu verbessern.

⟨*weitere verbesserungen von* weg_neu *mit bus* 176⟩≡

```
-- nur möglich, wenn bus-tickets vorhanden
-- und mindestes ein bus-knoten auf weg_neu
if (bus_tickets > 0) ∧ (#b > 0)
then
   laenge_anfang ← position_in (b(1),weg_neu);
   vom_start ← [x : x ∈ [[[x,"taxi"] : x ∈ path(weg(1),y)]
                 : y ∈ busknoten]
                 | #x < laenge_anfang];
   if (#vom_start > 0)
   then
      ab ← delete_components (weg_neu, 1, laenge_anfang);
      zum_ersten_bus_knoten ← [[[b(1), "bus"]]];
      if (bus_tickets > 1)
      then
         zum_ersten_bus_knoten +← [[[x,"bus"],[b(1),"bus"]]
                                   : x ∈ erreichbar(b(1),"bus")];
      end if;
      neue_wege ← [x + y + ab
                   : x ∈ vom_start, y ∈ zum_ersten_bus_knoten
                   | x(#x)(1) ∈ bus(y(1)(1))];
      if (#neue_wege > 1)
      then
         -- weg_neu wird der beste (= kürzeste) weg aus neue_wege zugewiesen
         for i ∈ neue_wege do
           if (#weg_neu > #i) then weg_neu ← i; end if;
         end for;
         -- bus_tickets anpassen
         bus_tickets ← tickets ("bus") - #[x(1) : x in weg_neu(2..)
                                           | x(2) = "bus"];
      end if;
   end if;
end if;
if (bus_tickets > 0) ∧ (#b > 0)
then
   pos ← position_in (b(#b),weg_neu);
```

```
    laenge_ende ← #weg_neu - pos - 1;
    vom_ende ← [x : x ∈ [[[x,"taxi"] : x ∈ path(y,weg(#weg))]
                 : y ∈ busknoten]
                 | #x < laenge_ende];
    if (#vom_ende > 0)
    then
      bis ← delete_components (weg_neu, pos, #weg_neu);
      ticket_typ ← weg_neu(pos)(2);
      zum_letzten_bus_knoten ← [[[b(#b), ticket_typ]]];
      if (bus_tickets > 1)
      then
         zum_letzten_bus_knoten +← [[[b(#b), ticket_typ],[x,"bus"]]
                                   : x ∈ erreichbar(b(#b),"bus")];
      end if;
      neue_wege ← [bis + x + [[y(1)(1), "bus"]] + y(2..) with ["ende"]
                   : x ∈ zum_letzten_bus_knoten, y ∈ vom_ende
                   | x(#x)(1) ∈ bus(y(1)(1))];
      -- weg_neu wird der beste (= kürzeste) weg aus neue_wege zugewiesen
      for i ∈ neue_wege do
        if (#weg_neu > #i) then weg_neu ← i; end if;
      end for;
   end if;
 end if;
 -- taxi_moves anpassen
 taxi_moves ← #[x(1) : x ∈ weg_neu(2..) | x(2) = "taxi"];
```

This code is used in chunk 171.

Die folgende Funktion berechnet den Index des längsten Bus- oder U-Bahnpfades in dem Tupel der Bus- oder U-Bahnstrecken (gemessen in Taxitickets = 3. Komponente eines Eintrages)

⟨*longest_path* 177⟩≡

```
 procedure longest_path (path);
 begin
    max_index ← 1;
    for i ∈ [1 .. #path] do
       if path(i)(3) ≥ path(max_index)(3)
          then max_index ← i;
       end if;
    end for;
    return max_index;
 end longest_path;
```

This code is used in chunk 171.

Die Funktion `position_in` errechnet den Index eines Knotens in dem Tupel `of`, dessen erster Eintrag `start` ist. Wenn der Knoten nicht gefunden wurde, gibt die Funktion den Wert `0` zurück.

⟨*position_in* 178a⟩≡

```
procedure position_in (start,of);
begin
   i ← 0;
   repeat
      i +← 1;
      anfang fromb of;
   until ((of = []) ∨ (start = anfang(1))) end repeat;
   if (start ≠ anfang(1))
      then return 0;
      else return i;
   end if;
end position_in;
```

This code is used in chunk 171.

`delete_components` löscht aus einem Tupel die Komponeneten vom Index `p1` bis `p2` (inklusive). Ist der erste Index nicht kleiner als der zweite oder einer oder beiden kleiner als `1`, wird eine Ausnahme `falsche_parameter` erzeugt und die Funktion verlassen.

⟨*delete_components* 178b⟩≡

```
procedure delete_components (tup,p1,p2);
begin
   if (p1 > p2) ∨ (p1 ≤ 0) ∨ (p2 ≤ 0)
   then
      escape falsche_parameter (["delete_components",tup,p1,p2]);
   elseif (p1 = p2) ∧ (#tup = 1)
   then
      tup(p1) ← tup(p1 + 1);
   else
      tup(p1..p2 + 1) ← [tup(p2 + 1)];
   end if;
   return tup;
end delete_components;
```

This code is used in chunk 163b.

Die nächste Funktion berechnet aus der Shortest-Path-Matrix den kürzesten Taxiweg vom Knoten `von` zum Knoten `nach`. Dabei wird abgefangen, daß `von` und `nach` die gleichen Knoten sind.

⟨*path* 179a⟩≡
```
procedure path (von,nach);
begin
   if (von = nach)
      then return ([von]);
      else return ([von] + path(matrix(von)(nach), nach));
   end if;
end path;
```
This code is used in chunk 163b.

13.7.5 Die Prozedur `update_D_variables`

Nachfolgend werden die für die Detektive relevanten Daten angepaßt.

⟨*update_D_variables* 179b⟩≡
```
procedure update_D_variables(rd moves);
begin
  ⟨Anzahl der Detektivzuege anpassen 179c⟩
  for nr ∈ [1..D_count] do
      ⟨Aendern der aktuellen Position 179d⟩
      ⟨Aendern des Ticketvorrates (Detektive) 180a⟩
      ⟨Aendern des Ticketvorrates (Mister X) 180b⟩
  end for;
end update_D_variables;
```
This code is used in chunks 148 and 194.

⟨*Anzahl der Detektivzuege anpassen* 179c⟩≡
```
  D_moves + ← 1;
  if number_of_D_moves ≠ Ω
     then number_of_D_moves + ← 1;
  end if;
```
This code is used in chunk 179b.

⟨*Aendern der aktuellen Position* 179d⟩≡
```
          D_positions(nr) ← moves(nr)(2);
```
This code is used in chunk 179b.

⟨*Aendern des Ticketvorrates (Detektive)* 180a⟩≡
```
for i ∈ [1..3] do
    if D_tickets(nr)(i)(1) = moves(nr)(3)
       then
            ⟨DEBUG_update_D_variables 192b⟩
            D_tickets(nr)(i)(2) ← D_tickets(nr)(i)(2) - 1;
    end if;
end for;
```
This code is used in chunk 179b.

Die von den Detektiven verwendeten Tickets werden Mister X zur Verfügung gestellt:

⟨*Aendern des Ticketvorrates (Mister X)* 180b⟩≡
```
for i ∈ [1..5] do
    if X_tickets(i)(1) = moves(nr)(3)
       then X_tickets(i)(2) ← X_tickets(i)(2) + 1;
    end if;
end for;
```
This code is used in chunk 179b.

13.7.6 Die Prozedur `send_move_to_gui`

Erzeugt aus den Zügen der Detektive Nachrichten, die an die GUI gesendet werden. Außerdem werden die aktualisierten Werte für die Ticketvorräte der Detektive und Mister X an die GUI übermittelt.

⟨*send_move_to_gui* 180c⟩≡
```
procedure send_move_to_gui(rd moves);
begin
  for x ∈ moves do
      send_gui_message("D_move " + capsulated_str(x(1)) + " " +
                       if x(2) = 0
                          then capsulated_str(D_positions(x(1)))
                               + " " + "-"
                          else capsulated_str(x(2)) + " " + x(3)
                       end if);
  end for;
  send_ticket_data(D_count);
  send_gui_message("move_no " + capsulated_str(X_moves));
end send_move_to_gui;
```
This code is used in chunks 148 and 215.

⟨*send_X_positions_to_gui* 181a⟩≡
```
    if X_positions ≠ ∅
       then for x ∈ X_positions do
                send_gui_message("possible_X_position "
                                  + capsulated_str(x));
            end for;
    end if;
```
This code is used in chunk 150.

13.8 Die Prozedur send_gui_message und die Funktion get_gui_message

Diese Funktion sendet eine Nachricht an die GUI.

⟨*send_gui_message* 181b⟩≡
```
  procedure send_gui_message(rd message);
  begin
    if simulate
       then put("Nachricht gesendet:", message);
       else laenge ← #message + 1;
            c_fct_call port_send(message : c_string , laenge : c_integer);
  put("Nachricht gesendet:", message);
    end if;
  end send_gui_message;
```
This code is used in chunks 127, 206, 207, 215, 216, and 221.

Diese Funktion liest die nächste Nachricht von der GUI aus der Pipe. Falls die GUI simuliert werden soll wird die nächste Nachricht aus der Simulationsdatei gelesen. Die Nachrichten `"game_over"` und `"exit"` werden gesondert behandelt: Wird die Nachricht `"game_over"` empfangen, wird das Spiel durch Erzeugung einer Ausnahme neu gestartet; beim Empfang der Nachricht `"exit"` wird eine Ausnahme zum Beenden des Programms erzeugt.

⟨*get_gui_message* 182a⟩≡
```
procedure get_gui_message();
begin
  if simulate
     then fget(SCOTLAND_DIRECTORY + SIMULATION_FILENAME, message);
          put("Nachricht empfangen:", message);
     else message ← c_fct_call port_receive_string() c_string;
put("Nachricht empfangen:", message);
  end if;

  case message
       when "game_over" ⇒ escape game_over_message(true);
       when "exit"      ⇒ escape game_over_message(false);
       else  return(message);
  end case;
end get_gui_message;
```
This code is used in chunks 127, 206, 216, and 221.

13.9 Die Funktion `X_moves_to_visible`

Berechnet die Anzahl der Züge nach der sich Mister X zeigen muß.

⟨*X_moves_to_visible* 182b⟩≡
```
procedure X_moves_to_visible();
begin
  case X_moves
       when 3,8,13,18,24 ⇒ return(0);
       when 2,7,12,17,23 ⇒ return(1);
       when 1,6,11,16,22 ⇒ return(2);
       when 0,5,10,15,21 ⇒ return(3);
       when 4,9,14,20    ⇒ return(4);
       when 19           ⇒ return(5);
  end case;
end X_moves_to_visible;
```
This code is used in chunks 127, 193, 200, 204, 211, 216, and 221.

13.10 Die Funktion possible_X_positions

Nachfolgend werden alle Positionen berechnet, auf denen sich Mister X zur Zeit aufhalten könnte. Dafür wird auf die Menge aller Positionen, die für Mister X vor seinem letzten Zug in Frage kamen und auf die Positionen, an denen sich Detektive momentan aufhalten, zurückgegriffen. Zu Beginn des Spieles befinden sich in der Menge alle Knoten außer denen, die von Detektiven besetzt sind. Zeigt sich Mister X im aktuellen Zug, steht in dieser Menge nur der Knoten, auf dem er sich zeigt. Nach einem Doppelzug von Mister X (`twice`), stehen im Übergabetupel `ticket_tuple` die zwei Tickets, die er nach `twice` benutzt hat.

⟨*possible_X_positions* 183⟩≡

```
procedure possible_X_positions (ticket_tuple);
begin
   D_positions_set ← –k : k ∈ D_positions″;
   for ticket ∈ ticket_tuple do
      new ← ∅;
      for position ∈ (X_positions - D_positions_set) do
         new + ← erreichbar (position,ticket);
      end for;
      X_positions ← new;
   end for;
   return X_positions;
end possible_X_positions;
```

This code is used in chunks 127, 200, 204, and 211.

13.11 Wir umschiffen die Klippen von ProSet

Da ProSet zur Zeit der Implementierung keine Funktion zum Umwandeln von Strings in Integers zur Verfügung stellte, behalfen wir uns mit der folgenden:

⟨*newint* 184a⟩≡
```
procedure newint(text);
-- konvertiert maximal bis 2147483647
begin
  if #text > 0
     then zahl ← 0;
          for i ∈ [1..#text] do
              zahl ← zahl * 10;
              case text(i)
                   when "0" ⇒ pass;
                   when "1" ⇒ zahl +← 1;
                   when "2" ⇒ zahl +← 2;
                   when "3" ⇒ zahl +← 3;
                   when "4" ⇒ zahl +← 4;
                   when "5" ⇒ zahl +← 5;
                   when "6" ⇒ zahl +← 6;
                   when "7" ⇒ zahl +← 7;
                   when "8" ⇒ zahl +← 8;
                   when "9" ⇒ zahl +← 9;
                   else escape Scan_Fehler_in_newint(text);
              end case;
          end for;
          return (zahl);
       else return (Ω);
  end if;
end newint;
```
Root chunk (not used in this document).

Zur Zeit der Entwicklung war die Bibliotheksfunktion `str()` noch nicht verfügbar. Weil aber zumindest eine Konvertierung von Integerwerten in Strings benötigt wurde, behalfen wir uns mit einer C-Funktion:

⟨*newstr* 184b⟩≡
```
procedure newstr(rd wert);
begin
  ergebnis ← c_fct_call mystr(wert : c_integer) c_string;
  return (ergebnis);
end newstr;
```
Root chunk (not used in this document).

Auch die Bibliotheksfunktion `system()` war noch nicht verfügbar, daher:

⟨*newsystem* 185a⟩≡
```
procedure newsystem(rd command);
begin
  c_fct_call mysystem(command : c_string);
end newsystem;
```
Root chunk (not used in this document).

Die vorgenannten Funktionen sind inzwischen verfügbar und auch schon im Programm durch die entsprechenden Bibliotheksfunktionen ersetzt.

Weiterhin ist die Funktion `break()` zum Zeitpunkt der Erstellung dieses Dokumentes noch nicht verfügbar:

⟨*newbreak* 185b⟩≡
```
procedure newbreak(rw line, rd divider);
  hidden characters ← [];
  hidden prefix ← "";
begin
  if (line = "") ∨ (divider = "")
     then return("");
  end if;
  for i ∈ [1..#divider] do
      characters with ← divider(i);
  end for;
  i ← 1;
  while (i ≤ #line) ∧ (line(i) ∉ characters) do
    prefix + ← line(i);
    i + ← 1;
  end while;
  if i = 1
     then return;
  elseif i > #line
     then line ← "";
          return(line);
   else line ← line(i..);
        return(prefix);
  end if;
end newbreak;
```
This code is used in chunks 127, 216, and 221.

⟨*newsubset* 186⟩≡

```
procedure newsubset(a, b);
-- ist Menge a Untermenge von b?
begin
  if (#a > #b)
  then
    return false;
  else
    ist_untermenge ← true;
    while ((#a > 0) ∧ ist_untermenge) do
       x from a;
       ist_untermenge ← x ∈ b;
    end while;
    return (ist_untermenge);
  end if;
end newsubset;
```

This code is used in chunks 127 and 216.

13.12 Test und Fehlerbeseitigung

Da der Treiber eine zentrale Funktion im Programm einnimmt, wurde besonderer Wert auf Stabilität und Fehlerfreiheit gelegt. Da die Integration erst zu einem späten Zeitpunkt erfolgen konnte, wurde mit den folgenden Tests versucht, möglichst viele Fälle im Spielverlauf abzutesten, insbesonders im Bezug auf die Kommunikation mit der GUI und der Planung.

Die Testumgebungen rufen jeweils die zu testende Funktion bzw. Prozedur mit einer Auswahl an Testdaten auf. Dabei wurden zunächst die Basisprozeduren und -funktionen getestet und anschließend bottom-up die zusammengesetzten Prozeduren, die diese Basisprozeduren verwenden.

13.12.1 Die Programmteile für das Testen

Die folgenden Codefragmente werden nur in der Testphase über bedingte Compilierung eingebunden. Sie kontrollieren jeweils das Eintreten eines bestimmten Ereignisses und brechen das gesamte Programm bei fehlerhaftem Verhalten mit einer Meldung ab.

Der folgende Programmteil prüft, ob die Nachricht `X_move` von der GUI empfangen wurde.

⟨*DEBUG_wait_for_X_1* 187⟩≡

```
@ifdef DEBUG
  tmp  ← message;
  tmp2 ← newbreak(tmp, " ");

  if (tmp2 ≠ "X_move")
     then put("wrong message from GUI in wait_for_X()");
          put("message read   :", message);
          put("message awaited: 'X_move ...'");
          stop;
  end if;
@endif
```

This code is used in chunk 150.

Der nächste Teil arbeitet wie der vorhergehende, allerdings wird die Nachricht `X_position` erwartet.

⟨*DEBUG_wait_for_X_2* 188a⟩≡

```
@ifdef DEBUG
  tmp  ← message;
  tmp2 ← newbreak(tmp, " ");

  if (tmp2 ≠ "X_position")
     then put("wrong message from GUI in wait_for_X()");
          put("message read   :", message);
          put("message awaited: 'X_position ...'");
          stop;
  end if;
@endif
```

This code is used in chunk 153.

Mit folgendem Programmteil wird gewährleistet, daß kein weiterer Doppelzug während eines bereits stattfindenden Doppelzuges von Mister X durchgeführt wird.

⟨*DEBUG_wait_for_X_3* 188b⟩≡

```
@ifdef DEBUG
  if ¬ doppelzug_erlaubt
     then put("X macht Doppelzug waehrend eines Doppelzuges in wait_for_X()");
     stop;
  end if;
@endif
```

This code is used in chunk 151a.

Das nächste Programmfragment kontrolliert, ob mit einer erwarteten Nachricht `X_move` auch ein gültiges Ticket übermittelt wird.

⟨*DEBUG_wait_for_X_4* 189a⟩≡
```
@ifdef DEBUG
  tmp  ← message;
  tmp2 ← newbreak(tmp, " ");

  if (tmp2 ≠ "X_move")
     then put("wrong message from GUI in wait_for_X()");
          put("message read   :", message);
          put("message awaited: 'X_move ...'");
          stop;
  end if;
  if message(8..) ∉ –"taxi","bus","underground","black"″
     then put("X verwendet ein unbekanntes Ticket in wait_for_X()");
          put("Ticket:", message(8..));
          put("Erwartete Tickets: taxi, bus, underground, black (, twice)");
          stop;
  end if;
@endif
```
This code is used in chunk 150.

Im folgenden wird das Programm abgebrochen, wenn Mister X ein Ticket verwenden will, das er gar nicht besitzt.

⟨*DEBUG_wait_for_X_5* 189b⟩≡
```
@ifdef DEBUG
  if X_tickets(i)(2) = 0
     then put("X verwendet ein nicht vorhandenes Ticket in wait_for_X()");
          put("Ticket:", X_tickets(i)(1));
          stop;
  end if;
@endif
```
This code is used in chunk 154.

Das nächste Programmfragment bricht das Programm ab, wenn ein Tupel zur Bekanntgabe des zuletzt von Mister X gemachten Zuges von einem Detektiv der Planungskomponente nicht abgeholt wurde.

⟨*DEBUG_update_TS_data* 190a⟩≡
```
@ifdef DEBUG
  meet ("current_X_move", ? eins , ? zwei)
         ⇒ put("Ein Tupel 'current_X_move' fuer Detektiv",
                eins, "und Zug", zwei);
           put("wurde nicht abgeholt in update_TS_data()");
           stop;
    at interface
       else pass;
  end meet;
@endif
```
This code is used in chunk 155.

Mit dem folgenden Teil wird das Programm abgebrochen, wenn in der Initialisierungsphase erwartete Informationen von der GUI nicht übermittelt werden.

⟨*DEBUG_init_1* 190b⟩≡
```
@ifdef DEBUG
   when "block_end", ""  ⇒ pass;
   else put("wrong message from GUI in init() ");
        put("(Endgueltige Startwerte empfangen");
        put("message read:", message);
        stop;
@endif
```
This code is used in chunk 139.

Der nächste Teil bricht ab, wenn die GUI es unterläßt, die Startpositionen der Detektive zu senden.

⟨*DEBUG_init_2* 190c⟩≡
```
@ifdef DEBUG
   put("wrong message from GUI in init()");
   put("message read:", msg);
   put("messages awaited: 'D_startposition', 'move_no',");
   put("                  'possible_X_position', 'used_X_ticket',");
   put("                  'last_X_position' or 'block_end'");
   stop;
@endif
```
This code is used in chunk 141.

Der nächste Teil prüft, ob die erwartete Nachricht `new_game` gesendet wurde.

⟨*DEBUG_init_3* 191a⟩≡
```
@ifdef DEBUG
  if message ≠ "new_game"
     then put("wrong message from GUI in init()");
          put("message read:", message);
          put("message awaited: 'new_game'");
          stop;
  end if;
@endif
```
This code is used in chunk 138.

Das folgende Fragment kontrolliert, ob ein gültiger Name für eine Spielstufe von der GUI gesendet wurde.

⟨*DEBUG_init_4* 191b⟩≡
```
@ifdef DEBUG
  if message ∉ list_of_level_names
     then put("Unzulaessigen Level-Namen angegeben in init()");
          put("Erlaubte Namen:", list_of_level_names);
          put("Angegeben:", message);
          stop;
  end if;
@endif
```
This code is used in chunk 139.

Der nächste Programmteil kontrolliert, ob der Funktion `X_moves_to_visible` ein korrekter Parameter übergeben wurde.

⟨*DEBUG_X_moves_to_visible* 191c⟩≡
```
@ifdef DEBUG
    else put("wrong no. of X_moves");
         put("no. get:", X_moves);
         stop;
@endif
```
Root chunk (not used in this document).

die beiden folgenden Abfragen überprüfen einmal in der Funktion `serve_strategy_requests` und einmal in der Prozedur `update_D_variables`, ob ein Detektiv ein nicht verfügbares Ticket benutzen will.

⟨*DEBUG_serve_strategy_requests* 192a⟩≡
```
@ifdef DEBUG
   if D_tickets(nr)(i)(2) = 0
      then put("Detektiv", nr,
               "verwendet ein nicht vorhandenes " +
               "Ticket in serve_strategy_requests()");
           put("Ticket:", D_tickets(nr)(i)(1));
           stop;
   end if;
@endif
```
Root chunk (not used in this document).

⟨*DEBUG_update_D_variables* 192b⟩≡
```
@ifdef DEBUG
   if D_tickets(nr)(i)(2) = 0
      then put("Detektiv", nr,
               "verwendet nicht vorhandenes Ticket " +
               "in update_D_variables()");
           put("Ticket:", D_tickets(nr)(i)(1));
           stop;
  end if;
@endif
```
This code is used in chunk 180a.

Die Testumgebung zu X_moves_to_visible

In dieser Umgebung werden alle möglichen Werte für die (globale) Variable `X_moves` getestet.

⟨*TEST_X_moves_to_visible* 193⟩≡

```
program test;
  visible X_moves;
begin
  for i ∈ [0..25] do
      X_moves ← i;
      put(X_moves, X_moves_to_visible());
  end for;

  ⟨X_moves_to_visible 182b⟩

end test;
```

Root chunk (not used in this document).

Die Testumgebung zu update_D_variables

⟨*TEST_update_D_variables* 194⟩≡

```
program test;
  visible D_count ← 5;
  visible D_positions ← [ 14, 118, 199, 108, 7];
  visible D_tickets ← [
                         [["taxi", 10], ["bus", 8], ["underground", 4]],
                         [["taxi", 10], ["bus", 8], ["underground", 4]],
                         [["taxi", 10], ["bus", 8], ["underground", 4]],
                         [["taxi", 10], ["bus", 8], ["underground", 4]],
                         [["taxi", 10], ["bus", 8], ["underground", 4]]
                       ];
  visible D_moves ← 0; -- Detektivzüge gesamt
  visible number_of_D_moves ← 3;
begin
  zuege ← [[1,15,"taxi"], [2,119,"bus"],
           [3,180,"underground"], [4,105,"bus"], [5,6,"taxi"]];
  put(zuege);
  update_D_variables(zuege);
  put("D_position, D_tickets, D_moves und number_of_D_moves:");
  put(D_positions);
  put(D_tickets);
  put(D_moves);
  put(number_of_D_moves);

  ⟨update_D_variables 179b⟩
end test;
```

Root chunk (not used in this document).

Die Testumgebung zu D_moves_correct

⟨*TEST_D_moves_correct* 195⟩≡

```
program test;
  visible D_count     ← 3;
  visible D_positions ← [9, 20, 1];
  visible D_tickets   ← [
                          [["taxi",1],["bus",0],["underground",0]],
                          [["taxi",1],["bus",0],["underground",0]],
                          [["taxi",1],["bus",0],["underground",0]]
                         ];
  visible board;
begin
  set_board();

  zuege ← [[1,19,"taxi"],[2,33,"taxi"],[3,8,"taxi"]];
  put("1. Fall: Alle ziehen korrekt.");
  put("Idiotische Detektive:");
  put(D_moves_correct(zuege));
  put("D_count, D_positions und D_tickets");
  put(D_count);
  put(D_positions);
  put(D_tickets);

  zuege ← [[1,1,"taxi"],[2,33,"taxi"],[3,8,"taxi"]];
  put("2. Fall: D1 zieht auf besetzten Knoten");
  put("Idiotische Detektive:");
  put(D_moves_correct(zuege));
  put("D_count, D_positions und D_tickets");
  put(D_count);
  put(D_positions);
  put(D_tickets);

  zuege ← [[1,19,"taxi"],[2,33,"taxi"],[3,58,"bus"]];
  put("3. Fall: D3 hat kein benoetigtes Busticket");
  put("Idiotische Detektive:");
  put(D_moves_correct(zuege));
  put("D_count, D_positions und D_tickets");
  put(D_count);
  put(D_positions);
  put(D_tickets);
```

```
zuege ← [[1,19,"taxi"],[2,33,"taxi"],[3,58,"taxi"]];
put("4. Fall: Knoten 58 nicht mit Taxi erreichbar");
put("Idiotische Detektive:");
put(D_moves_correct(zuege));
put("D_count, D_positions und D_tickets");
put(D_count);
put(D_positions);
put(D_tickets);

zuege ← [[1,19,"taxi"],[2,33,"taxi"],[3,18,"taxi"]];
put("5. Fall: D3 hat keine Verbindung zu Knoten 18");
put("Idiotische Detektive:");
put(D_moves_correct(zuege));
put("D_count, D_positions und D_tickets");
put(D_count);
put(D_positions);
put(D_tickets);

zuege ← [[1,19,"taxi"],[2,19,"taxi"],[3,18,"taxi"]];
put("6. Fall: D2 zieht auf Knoten, " +
    "auf den D1 unmittelbar vorher gezogen hat");
put("Idiotische Detektive:");
put(D_moves_correct(zuege));
put("D_count, D_positions und D_tickets");
put(D_count);
put(D_positions);
put(D_tickets);

zuege ← [[1,9,"taxi"],[2,20,"taxi"],[3,1,"taxi"]];
put("7. Fall: Alle Detektive bleiben stehen");
put("Idiotische Detektive:");
put(D_moves_correct(zuege));
put("D_count, D_positions und D_tickets");
put(D_count);
put(D_positions);
put(D_tickets);

zuege ← [[1,19,"taxi"],[2,0,"I am still standing!"],[3,8,"taxi"]];
put("8. Fall: D2 bleibt trotz Zugmoeglichkeit stehen");
put("Idiotische Detektive:");
put(D_moves_correct(zuege));
put("D_count, D_positions und D_tickets");
```

```
    put(D_count);
    put(D_positions);
    put(D_tickets);

    D_positions ← [19, 1, 9];
    zuege ← [[1,8,"taxi"],[2,0,"I am still standing!"],[3,20,"taxi"]];
    put("9. Fall: D2 kann nicht ziehen und bleibt auch stehen");
    put("Idiotische Detektive:");
    put(D_moves_correct(zuege));
    put("D_count, D_positions und D_tickets");
    put(D_count);
    put(D_positions);
    put(D_tickets);

    ⟨D_moves_correct 161⟩
    ⟨Testspielplan 198⟩
  end test;
```

Root chunk (not used in this document).

Die folgende Prozedur weist einer übergebenen Variablen die Repräsentation eines Test-Spielplanes zu.

⟨*Testspielplan* 198⟩≡

```
procedure set_board();
begin
  board ← [ ["taxi",- [5,[15,16]],[10,[2,11,34,21]],[15,[14,26,28,16,5]],
                      [20,[9,2,33]],[25,[14,38,39]],[30,[17,42]],
                      [35,[22,48,65,36]],[40,[27,41,52,53]],
                      [45,[32,58,46,59,60]],[50,[37,38,49]],[55,[54,71]],
                      [60,[45,61,76]],[65,[35,64,66]],[70,[54,71]],
                      [75,[74,58,59]],[3,[11,12,4]],[13,[4,23,24,14]],
                      [18,[8,31,43]],[23,[12,22,37,13]],[77,[76,78]],
                      [28,[15,16,27,41]],[33,[20,32,46,21]],
                      [43,[18,31,57]],[48,[34,62,63,35]],[53,[40,69,54]],
                      [58,[44,57,74,75,59,45]],[78,[61,67,79]],
                      [63,[48,79,64]],[68,[51,67,69]],[73,[57,74]],
                      [1,[8,9]],[6,[29,7]],[11,[3,10,22]],[9,[1,19,20]],
                      [21,[10,33]],[26,[15,39,27]],[31,[18,43,44]],
                      [41,[28,40,54,29]],[46,[33,45,61,47]],[7,[6,17]],
                      [56,[42]],[61,[46,60,76,78,62]],[66,[65,67,49]],
                      [71,[55,70,72]],[76,[59,60,61,77]],[4,[3,13]],
                      [14,[13,25,15]],[19,[8,9,32]],[24,[13,37,38]],
                      [29,[6,16,41,42,17]],[34,[10,74,48,22]],
                      [44,[31,32,58]],[49,[36,50,66]],[54,[41,53,70,55]],
                      [59,[58,45,76,75]],[64,[63,65]],[69,[52,68,53]],
                      [74,[73,58,75]],[79,[62,78,63]],[2,[20,10]],
                      [12,[3,23]],[17,[7,29,30]],[22,[11,34,35,23]],
                      [16,[5,15,28,29]],[36,[35,49,37]],
                      [37,[23,36,50,24]],[52,[39,51,69,40]],
                      [51,[38,39,52,67,68]],[39,[25,26,51,52]],
                      [27,[26,28,40]],[32,[19,44,45,33]],
                      [42,[29,30,56,72]],[47,[46,34,62]],
                      [57,[43,58,73]],[62,[47,61,79,48]],
                      [67,[66,68,51]],[72,[42,71]],[8,[1,19,18]],
                      [38,[24,25,51,50]] " ],
            ["bus",- [15,[14,41,29]],[55,[29]],[65,[22,63,67]],
                      [13,[23,14,52]],[23,[22,3,67,13]],[58,[1,74,77,46]],
                      [63,[34,79,65]],[78,[77,46,79]],[1,[58,46]],
                      [46,[1,58,78,34]],[14,[13,15]],[29,[15,41,55,42]],
                      [34,[46,63,22]],[74,[58]],[79,[78,63]],[7,[42]],
                      [22,[34,3,23,65]],[42,[29,7,72]],[52,[13,67,41]],
```

```
                         [67,[23,65,52]],[72,[42]],[77,[58,78]],
                         [3,[22,23]] " ],
        ["underground",– [13,[46,64]],[1,[46]],[46,[1,13,74,79]],[74,[46]],
                         [79,[46,67]],[67,[13,79]],[41,[52,15,29]] " ],
              ["black",– " ] ];
  end set_board;
```

This code is used in chunks 195, 200, 204, 211, and 213.

Die Testumgebung zu load_board_names

Für diesen Test müssen mehrere Dateien mit der Endung `.plan` im aktuellen Verzeichnis vorhanden sein.

⟨*TEST_load_board_names* 199⟩≡

```
  program test;
    persistent constant system: "StdLib";
    visible constant SCOTLAND_YARD ← "ScotlandYard";
    visible constant SCOTLAND_PLANS ← "ScotlandYard.planlist";
  begin
    put(load_board_names());

    ⟨load_board_names 142⟩
  end test;
```

Root chunk (not used in this document).

Die Testumgebung zu store_TS_data

```
⟨TEST_store_TS_data 200⟩≡
 program test;
   visible interface;
   visible board;
   visible D_count ← 3;
   visible level ← 1;
   visible last_public_X_position ← 58;
   visible used_X_tickets ← ["taxi"];
   visible all_used_X_tickets ← ["taxi"];
   visible number_of_X_moves;
   visible number_of_D_moves;
   visible D_positions ← [9, 20, 1];
   visible D_tickets ← [
                         [["taxi",1],["bus",1],["underground",1]],
                         [["taxi",2],["bus",2],["underground",2]],
                         [["taxi",3],["bus",3],["underground",3]]
                        ];
   visible X_tickets ← [["taxi",4],["bus",3],["underground",3],
                         ["black",D_count],["twice",2]];
 begin
   interface ← CreateTS(Ω);
   blackboard ← CreateTS(Ω);
   set_board();

   number_of_X_moves ← Ω;
   number_of_D_moves ← Ω;

   store_TS_data(blackboard);
   fetch_TS_data(blackboard);

   number_of_X_moves ← 1;
   number_of_D_moves ← 2;

   store_TS_data(blackboard);
   fetch_TS_data(blackboard);

   ⟨store_TS_data 147⟩
   ⟨fetch_TS_data 202⟩

   ⟨X_moves_to_visible 182b⟩
```

⟨*possible_X_positions* 183⟩

⟨*Testspielplan* 198⟩

```
end test;
```

Root chunk (not used in this document).

Die folgende Prozedur liest alle Daten aus dem Tupelraum, die durch *store_TS_data* dort hineingeschrieben wurden. Außerdem wird kontrolliert, ob dabei alle Tupel aus dem Tupelraum entfernt wurden.

⟨*fetch_TS_data* 202⟩≡

```
procedure fetch_TS_data(rd blackboard);
begin
  fetch ("level", ? etwas) at blackboard end fetch;
  put("level im Blackboard:"); put(etwas);
  fetch ("board", ? etwas) at interface end fetch;
  put("board:", board);
  fetch ("D_count", ? etwas) at interface end fetch;
  put("D_count:"); put(etwas);
  fetch ("level", ? etwas) at interface end fetch;
  put("level:"); put(etwas);
  fetch ("used_X_tickets", ? etwas) at interface end fetch;
  put("used_X_tickets:");
  put("last_public_X_position:"); put(etwas);
  fetch ("last_public_X_position", ? etwas) ⇒ put(etwas);
    xor ("last_public_X_position") ⇒ put("OK (om)");
     at interface
  end fetch;
  put("number_of_X_moves:");
  fetch ("number_of_X_moves", ? etwas) ⇒ put(etwas);
    xor ("number_of_X_moves") ⇒ put("OK (om)");
     at interface
  end fetch;
  put("X_moves_to_visible:");
  fetch ("X_moves_to_visible", ? etwas) ⇒ put(etwas);
    xor ("X_moves_to_visible") ⇒ put("OK (om)");
     at interface
  end fetch;
  put("number_of_D_moves:");
  fetch ("number_of_D_moves", ? etwas) ⇒ put(etwas);
    xor ("number_of_D_moves") ⇒ put("OK (om)");
     at interface
  end fetch;
  fetch ("possible_X_positions", ? etwas)
     at interface end fetch;
  put("possible_X_positions:"); put(etwas);
  fetch ("all_used_X_tickets", ? etwas)
     at interface end fetch;
```

```
    put("all_used_X_tickets:"); put(etwas);
    for nr ∈ [1..D_count] do
        fetch ("initial_D_position", ? etwas, ? anderes)
           at interface end fetch;
        put("initial_D_position: D-Nr. und Position");
        put(etwas, anderes);
    end for;
    for nr ∈ [1..D_count+1] do
        fetch ("ticket_count", ? etwas, ? anderes)
           at interface end fetch;
        put("ticket_count: D-Nr. und Tickets");
        put(etwas, anderes);
    end for;
  end fetch_TS_data;
```

This code is used in chunks 200, 204, and 211.

Die Testumgebung zu update_TS_data

⟨*TEST_update_TS_data* 204⟩≡

```
program test;
  visible interface;
  visible board;
  visible D_count ← 3;
  visible level ← 1;
  visible last_public_X_position ← 58;
  visible used_X_tickets ← ["taxi"];
  visible all_used_X_tickets ← ["taxi"];
  visible X_moves ← 15;
  visible number_of_X_moves;
  visible D_moves ← 14;
  visible number_of_D_moves;
  visible D_positions ← [9, 20, 1];
  visible D_tickets ← [
                        [["taxi",1],["bus",1],["underground",1]],
                        [["taxi",2],["bus",2],["underground",2]],
                        [["taxi",3],["bus",3],["underground",3]]
                      ];
  visible X_tickets ← [["taxi",4],["bus",3],["underground",3],
                        ["black",D_count],["twice",2]];
begin
  interface ← CreateTS(Ω);
  blackboard ← CreateTS(Ω);
  set_board();

  number_of_X_moves ← Ω;
  number_of_D_moves ← Ω;

  store_TS_data(blackboard);

  level ← 2;
  last_public_X_position ← 68;
  used_X_tickets ← ["taxi","bus"];
  all_used_X_tickets ← ["underground","taxi","bus"];
  number_of_X_moves ← 3;
  number_of_D_moves ← 4;
  D_positions ← [10, 21, 2];
  D_tickets ← [
                [["taxi",0],["bus",0],["underground",0]],
```

```
                 [["taxi",11],["bus",11],["underground",11]],
                 [["taxi",22],["bus",22],["underground",22]]
                ];
   X_tickets ← [["taxi",3],["bus",2],["underground",2],
                 ["black",D_count-1],["twice",3]];

   update_TS_data(["black"]);
   fetch_TS_data(blackboard);

   ⟨store_TS_data 147⟩
   ⟨update_TS_data 155⟩
   ⟨fetch_TS_data 202⟩
   ⟨X_moves_to_visible 182b⟩
   ⟨possible_X_positions 183⟩
   ⟨Testspielplan 198⟩
 end test;
```

Root chunk (not used in this document).

Die Testumgebung zu start_gui

⟨*TEST_start_gui* 206⟩≡

```
program test;
  persistent constant system: "StdLib";
  visible simulate ← (argv(2) = "-s");
  visible constant SIMULATION_FILENAME ← "ScotlandYard.sim";
  visible constant SCOTLAND_PIPE       ← "ScotlandYard.pipe";
  visible constant SCOTLAND_GUI        ← "ScotlandYard.gui.nonblock";
begin
  start_gui();

  send_gui_message("I am the Server");
  repeat
    temp ← get_gui_message();
  until temp ≠ "" end repeat;

  put("Server empfaengt:", temp);

  if ¬ simulate
     then c_fct_call port_disconnect();
     else fclose(SIMULATION_FILENAME);
  end if;

  ⟨start_gui 135⟩
  ⟨send_gui_message 181b⟩
  ⟨get_gui_message 182a⟩
end test;
```

Root chunk (not used in this document).

Das folgende Programm realisiert einen Client für die Inter-Process-Communication (IPC) mit der GUI. Das heißt, es nimmt eine Nachricht am „anderen Ende“ der Unix-Pipe entgegen und sendet seinerseits eine Nachricht in die Pipe an den Server, der in dem obigen Programm realisiert ist. Das Client-Programm muß separat übersetzt und gestartet werden.

⟨*Client* 207⟩≡

```
program client;
  visible simulate ← false;
  visible constant SIMULATION_FILENAME ← "ScotlandYard.sim";
  visible constant SCOTLAND_PIPE       ← argv(3);
begin
  nonblocking ← 1;

  c_fct_call port_client_connect(SCOTLAND_PIPE : c_string);
  c_fct_call port_toggle_receive(nonblocking : c_integer);

  send_gui_message("I am the Client");

  temp ← "";
  repeat
    temp ← get_gui_message();
  until temp ≠ "" end repeat;

  put("Client empfaengt:", temp);

  ⟨send_gui_message 181b⟩

  procedure get_gui_message();
  begin
    message ← c_fct_call port_receive_string() c_string;
    return(message);
  end get_gui_message;

end client;
```

Root chunk (not used in this document).

Die Testumgebung zu load_board

Für diese Umgebung muß im aktuellen Verzeichnis eine Datei mit einer Spielbrettrepräsentation und einer Menge möglicher Startknoten unter dem Namen `ScotlandYard.plan` vorhanden sein.

⟨*TEST_load_board* 208⟩≡

```
program test;
  visible board;
  visible constant SCOTLAND_YARD ← "ScotlandYard";
begin
  start_positions ← load_board(SCOTLAND_YARD);

  put("Startpositionen:", start_positions);
  if board ≠ Ω
     then  put("Spielplan: OK");
     else  put("Spielplan: om");
  end if;
  put(board);
  put("Fertig!");

  ⟨load_board 144⟩
end test;
```

Root chunk (not used in this document).

Die Testumgebung zu `generate_X_position`

⟨*TEST_generate_X_position* 209⟩≡

```
program test;
  visible D_positions;
begin
  D_positions ← [8, 9, 10];
  put("1. Fall: D-Positionen:", D_positions);

  start_positionen ← –1, 2, 3, 4, 5, 6, 7″;
  temp ← generate_X_position(start_positionen);
  put("Startposition von Mr. X:", temp);

  D_positions ← [1, 5, 7];

  put("2. Fall: D-Positionen:", D_positions);
  for i ∈ [1..5] do
    put("Durchlauf", i);
    temp ← generate_X_position(start_positionen);
    put("Startposition von Mr. X:", temp);
  end for;

  ⟨generate_X_position 143b⟩
end test;
```

Root chunk (not used in this document).

Die Testumgebung zu generate_D_positions

⟨*TEST_generate_D_positions* 210a⟩≡

```
program test;
  visible D_count ← 5;
begin
  start_positionen ← −1, 2, 3, 4, 5, 6, 7″;
  for i ∈ [1..10] do
     temp ← generate_D_positions(start_positionen);
     put("Durchlauf, Startpositionen:", i, temp);
     if #temp = #−x: x ∈ temp″
        then put("OK");
        else put("Fehler!");
     end if;
  end for;

  ⟨generate_D_positions 143a⟩
end test;
```

Root chunk (not used in this document).

Die Testumgebung zu simulate_gui

Hierfür muß das aktuelle Verzeichnis eine Datei mit dem Namen `ScotlandYard.sim` enthalten.

⟨*TEST_simulate_gui* 210b⟩≡

```
program test;
  visible simulate ← false;
  visible SIMULATION_FILENAME ← "ScotlandYard.sim";
begin
put("Start");
  simulate_gui();
  fclose(SIMULATION_FILENAME);

  put("simulate:", simulate);

  ⟨simulate_gui (never defined)⟩
end test;
```

Root chunk (not used in this document).

Die Testumgebung zu start_strategy

⟨*TEST_start_strategy* 211⟩≡

```
program test;
  visible interface;
  visible board;
  visible D_count ← 3;
  visible level ← 1;
  visible last_public_X_position ← 58;
  visible used_X_tickets ← ["taxi"];
  visible all_used_X_tickets ← ["taxi"];
  visible number_of_X_moves;
  visible number_of_D_moves;
  visible D_positions ← [9, 20, 1];
  visible D_tickets ← [
                        [["taxi",1],["bus",1],["underground",1]],
                        [["taxi",2],["bus",2],["underground",2]],
                        [["taxi",3],["bus",3],["underground",3]]
                       ];
  visible X_tickets ← [["taxi",4],["bus",3],["underground",3],
                        ["black",D_count],["twice",2]];
begin
  set_board();
  start_strategy();

  procedure Start_Detective(rd nummer, rd TS1, rd TS2);
  begin
    put("Detektiv", nummer, "ist gestartet");
    if nummer = D_count
       then fetch_TS_data(TS2);
            loop
              fetch ( ? etwas )
                 ⇒ put("Ausserdem im Interface gefunden:");
                    put(etwas);
                 at TS1
               else put("Interface-TS ist leer");
                    quit;
              end fetch;
            end loop;
            loop
              fetch ( ? etwas )
                 ⇒ put("Ausserdem im Blackboard gefunden:");
```

```
                    put(etwas);
                 at TS2
               else put("Blackboard-TS ist leer");
                    quit;
              end fetch;
            end loop;
    end if;
    ⟨fetch_TS_data 202⟩
  end Start_Detective;

  ⟨start_strategy 146⟩
  ⟨X_moves_to_visible 182b⟩
  ⟨possible_X_positions 183⟩
  ⟨Testspielplan 198⟩
end test;
```

Root chunk (not used in this document).

Die Testumgebung zu serve_strategy_requests

⟨*TEST_serve_strategy_requests* 213⟩≡

```
program test;
  visible D_count ← 3;
  visible board;
  visible D_positions ← [1, 19, 20];
  visible D_tickets ← [[["taxi", 4],["bus", 4],["underground", 1]],
                  [["taxi", 4],["bus", 4],["underground", 1]],
                  [["taxi", 4],["bus", 4],["underground", 1]]];
  visible zuege ← [[46,"bus"],[9,"taxi"],[33,"taxi"]];
  visible interface;
begin
  set_board();
  interface ← CreateTS(Ω);
  put("1. Fall: Alle Detektive ziehen korrekt.");
  zuege ← [[46,"bus"],[9,"taxi"],[33,"taxi"]];
  for i ∈ [1..D_count] do
      || closure detective_process(i);
  end for;
  ergebnis ← serve_strategy_requests();
  put("Detektivzuege:", ergebnis);
  for i ∈ [1..D_count] do
      fetch ("fertig", ?) at interface end fetch;
  end for;
  put("2. Fall: D1 zieht zweimal falsch, dann richtig");
  zuege ← [[9,"bus"],[9,"taxi"],[33,"taxi"],
           [9,"taxi"],[9,"taxi"],[33,"taxi"],
           [46,"taxi"],[9,"taxi"],[33,"taxi"]];
  for i ∈ [1..D_count] do
      || closure detective_process(i);
  end for;
  ergebnis ← serve_strategy_requests();
  put("Detektivzuege:", ergebnis);

  procedure detective_process(rd nr);
  begin
    put("Detektiv", nr, "gestartet");
    versuch ← 0;
    loop
      put("Versuch", versuch + 1);
      id ← newat();
```

```
        deposit ["possible_moves", id, random(nr)]
             at interface end deposit;
        put("Detektiv", nr, "Anfrage: possible_moves");
        fetch ( id, ? antwort ) at interface end fetch;
        put("Detektiv", nr,
            "Anfrage: possible_moves, Antwort:", antwort);
        id ← newat();
        deposit ["shortest_path", id, random(nr),
                 random(nr), D_tickets(nr)]
             at interface end deposit;
        put("Detektiv", nr, "Anfrage: shortest_path");
        fetch ( id, ? antwort ) at interface end fetch;
        put("Detektiv", nr,
            "Anfrage: shortest_path, Antwort:", antwort);
        id ← newat();
        position ← versuch * D_count + nr;
        deposit ["D_position", id, nr,
                 zuege(position)(1), zuege(position)(2)]
             at interface end deposit;
        put("Detektiv", nr, "zieht von", D_positions(nr),
            "nach", zuege(position)(1));
        put("und benutzt dazu", zuege(position)(2));
        fetch ( id, ? ok) at interface end fetch;
        put("Der Zug von Detektiv", nr, "war");
        if ok
           then put("korrekt");
                quit;
           else put("nicht korrekt");
                versuch ← versuch + 1;
        end if;
      end loop;
      deposit ["fertig", nr] at interface end deposit;
    end detective_process;

    ⟨serve_strategy_requests 157⟩
    ⟨Testspielplan 198⟩
  end test;
```

Root chunk (not used in this document).

Die Testumgebung zu send_move_to_gui

⟨*TEST_send_move_to_gui* 215⟩≡

```
program test;
  visible D_count ← 3;
  visible D_tickets ← [];
  visible X_tickets;
  visible constant SCOTLAND_PIPE ← "ScotlandYard.pipe";
  visible constant SCOTLAND_GUI  ← "echo GUI starten"; -- UNIX-Befehl
begin
  for nr ∈ [1..D_count] do
      D_tickets(nr) ← [["taxi",10],["bus",8],["underground",4]];
  end for;
  X_tickets  ← [["taxi",4],["bus",3],["underground",3],
                ["black",D_count],["twice",2]];
  start_gui(); -- Verbindung zum extern gestarteten Server aufbauen
  zuege ← [[1,46,"bus"],[2,9,"taxi"],[3,33,"taxi"]];
  send_move_to_gui(zuege);

  ⟨start_gui 135⟩
  ⟨send_move_to_gui 180c⟩
  ⟨send_gui_message 181b⟩
  ⟨Ticketvorraete der GUI melden 131a⟩
end test;
```

Root chunk (not used in this document).

Die Testumgebung zu `wait_for_X`

Für den Test dieser Prozedur muß extern ein Client gestartet werden, der die Pipe für die Kommunikation mit dem folgenden Testprogramm aufbaut. Bei einem korrekten Durchlauf empfängt der Client zuerst die Nachrichten `"X_move ..."` und `"X_show ..."`. Ist das Argument der letzteren Nachricht 0, sendet der Client zusätzlich die Nachricht `"X_position ..."`. Nach einer Nachricht `"X_move twice"` müssen zwei weitere `"X_move ..."`-Nachrichten folgen, immer mit vorherigem Empfang der beiden Nachrichten `"X_move ..."` und `"X_show ..."` vom Testprogramm. (Siehe auch *Nachrichtenaustausch zwischen der grafischen Benutzeroberfläche und der Treiberkomponente*.)

⟨*TEST_wait_for_X* 216⟩≡

```
program test;
  visible simulate ← (argv(2) ≠ Ω);
  visible D_count ← 3;
  visible D_positions;
  visible X_positions;
  visible last_public_X_position;
  visible X_moves;
  visible number_of_X_moves;
  visible number_of_D_moves;
  visible used_X_tickets;
  visible all_used_X_tickets;
  visible X_tickets;
  visible constant SIMULATION_FILENAME ← argv(2);
  visible constant SCOTLAND_PIPE       ← "ScotlandYard.pipe";
  visible constant SCOTLAND_GUI        ← "echo PortTalk starten!";
begin
  start_gui(); -- Verbindung zum extern gestarteten Client aufbauen
  put("Verbindung steht");

  put("===> 1. FALL: Mr. X macht einen gewoehnlichen Zug");
  X_moves ← 0;
  number_of_D_moves ← Ω;
  number_of_X_moves ← Ω;
  used_X_tickets ← [];
  D_positions ← [4,5,6];
  X_positions ← –10, 11, 13″;
  all_used_X_tickets ← [];
  X_tickets ← [["taxi",4],["bus",3],["underground",3],
                 ["black",D_count],["twice",2]];
  put("Ausgangsdaten:");
  zeige_daten();
```

```
ergebnis ← wait_for_X(true) when game_over_message use game_over;
put("Ergebnisdaten:");
put("Zug von X:", ergebnis);
zeige_daten();
put("===> 2. FALL: Bei einem Doppelzug muss " +
    "sich X nach dem 1. Zug zeigen");
X_moves ← 7;
number_of_D_moves ← 4;
number_of_X_moves ← 4;
D_positions ← [4,5,6];
X_positions ← −10, 11, 13″;
used_X_tickets ← ["bus","taxi","black","taxi"];
all_used_X_tickets ←
  ["underground","taxi","bus","bus","taxi","black","taxi"];
X_tickets ← [["taxi",3],["bus",3],["underground",2],
              ["black",D_count],["twice",2]];
put("Ausgangsdaten:"); zeige_daten();
ergebnis ← wait_for_X(false) when game_over_message use game_over;
put("Ergebnisdaten:");
put("Zug von X:", ergebnis);
zeige_daten();
put("===> 3. FALL: Mr. X kann nur auf Feldern stehen, " +
    "die von Detektiven besetzt sind");
X_moves ← 0;
number_of_D_moves ← Ω;
number_of_X_moves ← Ω;
used_X_tickets ← [];
D_positions ← [4,5,6];
X_positions ← −4, 5″;
all_used_X_tickets ← [];
X_tickets ← [["taxi",4],["bus",3],["underground",3],
              ["black",D_count],["twice",2]];
put("Ausgangsdaten:");
zeige_daten();
ergebnis ← wait_for_X(true) when game_over_message use game_over;
put("Ergebnisdaten:");
put("Zug von X:", ergebnis);
zeige_daten();
put("===> 4. FALL: Mr. X zeigt sich auf von einem D besetzten Feld");
X_moves ← 2;
number_of_D_moves ← 2;
number_of_X_moves ← 2;
```

```
used_X_tickets ← [];
D_positions ← [4,5,6];
X_positions ← -10, 11, 25";
all_used_X_tickets ← [];
X_tickets ← [["taxi",4],["bus",3],["underground",3],
              ["black",D_count],["twice",2]];
put("Ausgangsdaten:");
zeige_daten();
ergebnis ← wait_for_X(true) when game_over_message use game_over;
put("Ergebnisdaten:");
put("Zug von X:", ergebnis);
zeige_daten();

if simulate
  then fclose(SIMULATION_FILENAME);
  else c_fct_call port_disconnect();
end if;

procedure zeige_daten();
begin
  put("D_positions:", D_positions);
  put("X_positions:", X_positions);
  put("last_public_X_position:", last_public_X_position);
  put("X_moves:", X_moves);
  put("number_of_X_moves:", number_of_X_moves);
  put("number_of_D_moves:", number_of_D_moves);
  put("used_X_tickets:", used_X_tickets);
  put("all_used_X_tickets:", all_used_X_tickets);
  put("X_tickets:", X_tickets);
  put("possible_X_positions", X_positions);
end zeige_daten;

procedure possible_X_positions(rd tickettuple);
begin
  return X_positions;
end possible_X_positions;

⟨wait_for_X 150⟩
⟨start_gui 135⟩
⟨send_gui_message 181b⟩
⟨get_gui_message 182a⟩
⟨X_moves_to_visible 182b⟩
```

```
    ⟨is_X_arrested 133a⟩
    ⟨Exceptionhandler: game_over 149b⟩

    ⟨string2integer 133b⟩
    ⟨newbreak 185b⟩
    ⟨newsubset 186⟩
  end test;
```

Root chunk (not used in this document).

Die Testumgebung zum Exceptionhandler game_over

⟨*TEST_game_over* 220⟩≡

```
  program test;
    visible D_count ← 3;
    visible interface;
  begin
    interface ← CreateTS(Ω);
    for i ∈ [1..D_count] do
        || closure detective_process(i);
    end for;

    ende ← run_handler() when game_over_message use game_over;

    put("Handler meldet:", ende);
    put("Tupelraum existiert", if ¬ ExistsTS(interface) then "nicht" end if);

    procedure run_handler();
    begin
      escape game_over_message("Escape durchgefuehrt");
      return(true);
    end run_handler;

    procedure detective_process(rd nr);
    begin
      put("Detektiv", nr, "gestartet");
      fetch ("commit_suicide", nr)
               ⇒ put("Detektiv", nr, "hat Nachricht erhalten");
         at interface end fetch;
      put("Detektiv", nr, "beendet");
    end detective_process;

    ⟨Exceptionhandler: game_over 149b⟩
  end test;
```

Root chunk (not used in this document).

Die Testumgebung zu `init`

⟨*TEST_init* 221⟩≡

```
program test;
  visible max_D_count;
  visible D_count;
  visible level;
  visible D_positions;
  visible D_tickets;
  visible X_tickets;
  visible D_moves;
  visible number_of_D_moves;
  visible X_moves;
  visible number_of_X_moves;
  visible last_public_X_position;
  visible used_X_tickets;
  visible all_used_X_tickets;
  visible simulate; -- Simulationsfile erstellen!
  visible board_name;
  visible board;

  visible constant SCOTLAND_YARD       ← "ScotlandYard";
  visible constant SIMULATION_FILENAME ← argv(2);
  -- Mögliche Testfiles sind:
  --    test1.sim (normal)
  --    test2.sim (mit Abbruch im ersten Empfangsblock)
  --    test3.sim (mit Abbruch im zweiten Empfangsblock)
  visible constant SCOTLAND_PIPE       ← SCOTLAND_YARD + ".pipe";
  visible constant SCOTLAND_GUI        ← if argv(2) = Ω
                                            then
                                              SCOTLAND_YARD + ".gui"
                                            else  -- UNIX-Befehl
                                              "echo GUI starten"
                                         end if;
  visible constant SCOTLAND_PLANS      ← SCOTLAND_YARD + ".planlist";
begin
  simulate ← (SIMULATION_FILENAME ≠ Ω);
  start_gui();
  weiter ← init() when game_over_message use game_over;
  if ¬ weiter
     then put("Programmabbruch!!!");
  end if;
```

```
    zeige_daten();
    if simulate
       then fclose(SIMULATION_FILENAME);
    end if;
    procedure zeige_daten();
    begin
      put("max_D_count:", max_D_count);
      put("D_count:", D_count);
      put("level:", level);
      put("D_positions:", D_positions);
      put("D_tickets:", D_tickets);
      put("X_tickets:", X_tickets);
      put("D_moves:", D_moves);
      put("number_of_D_moves:", number_of_D_moves);
      put("X_moves:", X_moves);
      put("number_of_X_moves:", number_of_X_moves);
      put("last_public_X_position:", last_public_X_position);
      put("used_X_tickets:", used_X_tickets);
      put("all_used_X_tickets:", all_used_X_tickets);
      put("simulate:", simulate);
      put("board_name:", board_name);
    end zeige_daten;

    ⟨start_gui 135⟩
    ⟨init 136⟩
    ⟨send_gui_message 181b⟩
    ⟨get_gui_message 182a⟩
    ⟨X_moves_to_visible 182b⟩
    ⟨Ticketvorraete der GUI melden 131a⟩
    ⟨Exceptionhandler: game_over 149b⟩

    ⟨string2integer 133b⟩
    ⟨newbreak 185b⟩
  end test;
```

Root chunk (not used in this document).

Die Testumgebung zum Hauptprogramm ScotlandYard

Die folgende Prozedur ist ein Detektivprozess, der zu Testzwecken den der Planungskomponente ersetzt. Wird der Prozess gestartet, so meldet er das und versucht dann, den aktuellen Zug aus dem Tupelraum abzufragen. Gelingt das, so werden die Anfragen "shortest_path" und "possible_moves" gemacht. Schließlich wird ein Zug des Detektivprozesses in den Tupelraum geschrieben und auf die Meldung gewartet, ob dieser Zug korrekt war oder nicht.

⟨*TEST_Start_Detective* 223⟩≡

```
procedure Start_Detective(rd nr, rd interface, rd blackboard);
  hidden myposition ← D_positions(nr);
  hidden mytickets  ← D_tickets(nr);
begin
  put("Detektiv", nr, "ist gestartet");
  repeat
    fetch ("current_X_move", ? move)
      xor ("commit_suicide", nr) ⇒ quit;
       at interface
    end fetch;
    put("Aktueller X-Zug", move, "von Detektiv",
        nr, "gelesen");
    fetch ("X_moves_to_visible", ? move)
      xor ("commit_suicide", nr) ⇒ quit;
       at interface
    end fetch;
    fetch ("number_of_D_moves", nr, ? move)
      xor ("commit_suicide", nr) ⇒ quit;
       at interface
    end fetch;
    put("Aktuelle number_of_D_moves", move, "von Detektiv",
        nr, "gelesen");
    fetch ("last_public_X_position", ? move)
      xor ("commit_suicide", nr) ⇒ quit;
       at interface
    end fetch;
    put("Aktuelle last_public_X_position", move, "von Detektiv",
        nr, "gelesen");
    id ← newat();
    deposit ["shortest_path", id, nr, nr+1, mytickets]
         at interface end deposit;
    put("Detektiv", nr, "stellt Anfrage:",
        ["shortest_path", id, nr, nr+1, mytickets]);
```

```
  fetch (id, ? antwort)
    xor ("commit_suicide", nr) ⇒ quit;
     at interface
  end fetch;
  put("Detektiv", nr, "erhaelt Antwort auf Anfrage",
      "shortest_path", ":", antwort);
  ende ← false;
  repeat
    id ← newat();
    deposit ["possible_moves", id, myposition]
         at interface end deposit;
    put("Detektiv", nr, "stellt Anfrage:",
        ["possible_moves", id, myposition]);
    fetch (id, ? antwort)
      xor ("commit_suicide", nr) ⇒ ende ← true;
                                   quit;
       at interface
    end fetch;
    put("Detektiv", nr, "erhaelt Antwort auf Anfrage",
        "possible_moves", ":", antwort);
    zug ← random(antwort);
    ziehe(id, nr, zug(1), zug(2));
    fetch (id, ? zug_ok)
      xor ("commit_suicide", nr) ⇒ ende ← true;
                                   quit;
       at interface
    end fetch;
    if zug_ok
       then put("Detektiv", nr, "hat korrekt gezogen");
            myposition ← zug(1);
            case zug(2)
               when "taxi"
                 ⇒ mytickets(1)(2) ← mytickets(1)(2) - 1;
               when "bus"
                 ⇒ mytickets(2)(2) ← mytickets(2)(2) - 1;
               when "underground"
                 ⇒ mytickets(3)(2) ← mytickets(3)(2) - 1;
            end case;
       else put("Detektiv", nr, "hat nicht korrekt gezogen");
    end if;
  until zug_ok end repeat;
until ende end repeat;
```

```
    return;

    procedure ziehe(rd kennung, rd D_nr, rd ziel, rd ticket);
    begin
      deposit ["D_position", kennung, D_nr, ziel, ticket]
           at interface end deposit;
      put("Detektiv zieht:",
          ["D_position", kennung, D_nr, ziel, ticket]);
    end ziehe;
  end Start_Detective;
```

Root chunk (not used in this document).

14. Implementation der Planung

14.1 Prototyping

Wie wir in den Seminarvorträgen gesehen haben, kann der Prototyping-Ansatz dazu führen, daß erstellter Quellcode wieder verworfen werden muß.

Das ursprünglich für die Implementierung vorgesehene Modulkonzept mußte wieder verworfen werden, weil der ProSet-Compiler einen Fehler aufwies: Wenn in verschiedenen Modulen, die in einen einzigen ProSet-Quellcode eingebunden werden, Prozeduren mit gleichen Namen benutzt werden – z.B. die Prozedur `Calculate_Move` in den Modulen `Random` und `Refer`, – so wird die Übersetzung von ProSet nach C nicht korrekt durchgeführt. Dieser Compiler-Fehler führte zu einer Änderung der in diesem Kapitel vorgestellten Implementierung, so daß wir zunächst die daraus resultierten Prototypen skizzieren und danach die aktuelle Implementation wiedergeben, was sich auch bei den Überschriften anhand von *Prototyp* bzw. *Implementation* erkennen läßt.

14.1.1 Prototyp: Der Start des Detektivprozesses

Das Haupt-Programm des Planungs-Prozesses ist für jeden Detektiv gleich, unabhängig von der Strategie, die er verfolgt. Die verschiedenen Strategien werden durch Module realisiert, von denen jeder Detektiv genau eines als sein Strategie–Modul instanziiert. Damit wird für eine größtmögliche Flexibilität gesorgt, da für neue Strategien nur ein neues Modul implementiert und dessen Instanziierung ermöglicht werden muß.

Der Ablauf entspricht dem des Struktogramms 9.1, Seite 99.

⟨*Prototyp: detect.pst* 226⟩≡

```
procedure Start_Detective (rd Number, rd Communication_TS,
                           rd Blackboard_TS);

visible  strategy        ← Ω;        -- Strategie-Modul-Instanz
hidden    x_move          ← Ω;        -- der letzte Zug von Mister X
visible  helper          ← Ω;
visible  number_of_detectives ← 0;
```

```
begin
   ⟨Initialisierung des Helper–Moduls 232a⟩

   ⟨Lese Anzahl der Detektive aus 232b⟩

   ⟨Prototyp: Initialisierung der Strategie 228a⟩

   ⟨Womit hat Mister X gezogen? 233a⟩

   while (x_move ≠ Ω) do
      repeat
         strategy.Calculate_Move (x_move);

      until Best_Move_Found (x_move )
      end repeat;
      strategy.Make_Move ();
      ⟨Womit hat Mister X gezogen? 233a⟩

   end while;

   ⟨Prozeduren der internen Schnittstelle 235⟩
end Start_Detective;
```

Root chunk (not used in this document).

14.1.2 Prototyp: Initialiserung des Strategie–Modules

Bei der Initialisierung des Strategie–Moduls sind zwei Dinge zu beachten. Zum einem muß eine Spielstärkenregulierung vorgenommen werden, zum anderen müssen einige Daten an das Modul übergeben werden.

Die Bestimmung der Spielstärke wird durch die Auswahl der Strategie geregelt. Dies geschieht durch ein `meet` der Spielstärke. Wenn sie größer als 5 ist, wird der Stuttgarter Algorithmus verwendet, ansonsten der triviale Algorithmus. D.h., die Spielstärke zeigt sich durch die Fähigkeiten der Gesamtheit der Detektive, weniger durch einen Detektiv selber. Welcher Detektiv welche Fähigkeit besitzen wird, ist durch den undeterministischen Zugriff auf dem Tupelraum festgelegt (oder besser: *nicht* festgelegt).

⟨*Prototyp: Initialisierung der Strategie* 228a⟩≡

```
⟨Lese Spielstaerke aus 239⟩

if (level > 5) then
    ⟨Prototyp: Initialisiere Stuttgarter Algorithmus 228c⟩
else
    ⟨Prototyp: Initialisiere trivialen Algorithmus 228b⟩
end if;
```

This code is used in chunk 226.

Um den trivialen Algorithmus zu instantiieren, wird das Blackboard und der Kommunikationstupelraum exportiert.

⟨*Prototyp: Initialisiere trivialen Algorithmus* 228b⟩≡

```
strategy ← instantiate closure Random
             export Blackboard_TS, Communication_TS, helper;
             import Calculate_Move, Make_Move;
end instantiate;
```

This code is used in chunk 228a.

Um den Stuttgarter Algorithmus zu instantiieren, wird das Blackboard und der Kommunikationstupelraum exportiert.

⟨*Prototyp: Initialisiere Stuttgarter Algorithmus* 228c⟩≡

```
strategy ← instantiate closure Refer
             export Blackboard_TS, Communication_TS;
             import Calculate_Move, Make_Move;
end instantiate;
pass;
```

This code is used in chunk 228a.

14.2 Implementation: Der Start des Detektivprozesses

Das Haupt-Programm des Planungs-Prozesses ist für jeden Detektiv gleich, unabhängig von der Strategie, die er verfolgt. Die verschiedenen Strategien werden durch Prozeduren realisiert. Damit wird eine große Flexibilität geschaffen, da jederzeit eine neue Strategie durch die Modifikation der Varibalen `strategy` aufgerufen werden kann. Dies geschieht mit der Anfangsstrategie der ersten zwei Züge, die unabhängig von den Strategien des 'Mittelspiels' ist.

Der Ablauf entspricht dem des Struktogramms 9.1, Seite 99.

⟨*detect.pst* 229⟩≡

```
procedure Start_Detective (rd Detective_Number, rd Communication_TS,
                            rd Blackboard_TS);

visible strategy ← 0;         -- Strategiekennzeichnung
visible number_of_detectives ← 0;
visible last_detective_position ← 0;
visible Kamikaze ← false;
visible Debug_Names ← ∅;
visible Debug_Level ← 0;
visible help ← Ω;

hidden  move ← Ω;             -- der letzte Zug von Mister X
hidden  doppelzug ← false;
hidden  iterations ← 0;
hidden  move_counter ← 0;
hidden  level ← 0;
hidden  constant Reference_Strategy    ← 1;
hidden  constant Random_Strategy       ← 2;
hidden  constant Short_Strategy        ← 3;
hidden  constant Schwachsinn_Strategy  ← 4;
hidden  constant Sum_Strategy          ← 5;
hidden  constant Mix_Strategy          ← 6;
hidden  constant Anti_Knubble_Strategy ← 7;
hidden  plan ← [];
hidden  best_plan ← [];
hidden  best_move_found ← false;
hidden  use_mix_strategy ← false;

begin
  Init_Trace();
```

```
put_debug
 (0, "detect",
  "Release: $Id: detect.pw,v 2.45 1995/03/01 13:06:50 jodeleit Exp $");

⟨Start–Position ermitteln 234c⟩
if Detective_Number = 1 then
  put_debug (2, "detect",
             ["Gebe Runde", move_counter + 1,"bekannt."]);
  deposit ["Runde", move_counter + 1]
    at Blackboard_TS
  end deposit;
end if;


⟨Lese Anzahl der Detektive aus 232b⟩

⟨Auswahl der Strategie 238⟩
if stratgy = Mix_Strategy then
  use_mix_strategy ← true;
else
  use_mix_strategy ← false;
end if;

⟨Initialisierung des Helper–Moduls 232a⟩

⟨Womit hat Mister X gezogen? 233a⟩

while (x_move ≠ Ω) do
  doppelzug ← (#x_move = 2);
  move_counter +← 1;
  if doppelzug then
    move_counter +← 1;
  end if;

  put_debug (2, "detect", ["Zug-Nr ", move_counter]);
  iterations ← 0;
  best_move_found ← false;

  if (use_mix_strategy) then
    tmp ← random 3;
    case tmp
      when 0 ⇒ strategy ← Reference_Strategy;
```

```
    when 1 ⇒ strategy ← Short_Strategy;
    when 2 ⇒ strategy ← Sum_Strategy;
  end case;
end if;

if (move_counter > 2) then
  repeat
    iterations +← 1;
    put_debug (2, "detect", ["Iteration:", iterations]);
    case strategy
      when Random_Strategy
        ⇒ best_move_found ← Random_Calculate_Move (x_move,
                                                   iterations);
      when Reference_Strategy
        ⇒ best_move_found ← Refer_Calculate_Move (x_move,
                                                  iterations);
      when Sum_Strategy
        ⇒ best_move_found ← Sum_Calculate_Move (x_move,
                                                iterations);
      when Short_Strategy
        ⇒ best_move_found ← Short_Calculate_Move (x_move,
                                                  iterations);
      when Schwachsinn_Strategy
        ⇒ best_move_found ← Schwachsinn_Calculate_Move (x_move,
                                                        iterations);
      when Anti_Knubble_Strategy
        ⇒ best_move_found ← Anti_Knubble_Calculate_Move (x_move,
                                                         iterations);

    end case;

    ⟨Check Kamikaze–Auftrag 233b⟩
  until best_move_found
  end repeat;
else
  if (move_counter = 1) then
  put_debug (2, "detect", "calculate First Round");
    calculate_first_round (Detective_Number);
  else
  put_debug (2, "detect", "calculate Second Round");
    calculate_second_round (Detective_Number);
  end if;
```

```
      ⟨Check Kamikaze–Auftrag 233b⟩
    end if;

    put_debug (1, "detect", "Setze Calculation Ready!");
    deposit ["calculation_ready", Detective_Number]
      at Blackboard_TS
    end deposit;

    if Detective_Number = 1 then
      Await_Other_Detectives (move_counter);
      help.Resolve_Conflicts (move_counter);
    end if;

    ⟨Womit hat Mister X gezogen? 233a⟩

  end while;

  ⟨Prozeduren der internen Schnittstelle 235⟩
⟨Strategie-Prozeduren 234b⟩
end Start_Detective;
```

Root chunk (not used in this document).

Bei der Initialisierung des Helper–Moduls müssen die Tupelräume als Export–Parameter übergeben werden.

⟨*Initialisierung des Helper–Moduls* 232a⟩≡

```
put_debug (2, "detect", "Instanziierung von help:");
help ← instantiate (closure strategy_help)
          export Communication_TS, Blackboard_TS, closure put_debug;
          import Read_Plans, Resolve_Conflicts, Best_Plan,
                 Get_Possible_Moves, Get_Shortest_Path, Get_Path_Sum,
                 Publish_Move, Get_Position_Of, Get_Tickets_Of,
                 Get_X_Positions;
        end instantiate;
put_debug (2, "detect", "Instanziierung von help ist abgeschlossen");
```

This code is used in chunks 226 and 229.

Um die Strategie zu initialisieren, muß die Anzahl der Detektive bekannt sein.

⟨*Lese Anzahl der Detektive aus* 232b⟩≡

```
meet ("D_count", ? number_of_detectives)
  at Communication_TS
end meet;
```

This code is used in chunks 226 and 229.

Desweiteren wollen wir natürlich alle wissen, wohin Mister X gezogen hat. Dies können wir nicht in Erfahrung bringen, dafür aber, mit welchem Verkehrsmittel er abgehauen ist. Da man während des Wartens auch noch anderen Aufgaben nachgehen soll, schauen wir parallel noch nach einem schönen Selbstmord–Kommando.

⟨*Womit hat Mister X gezogen?* 233a⟩≡
```
put_debug (2, "detect", ["Waiting for Runde", move_counter + 1]);
meet ("Runde", move_counter + 1)
  at Blackboard_TS
end meet;
put_debug (2, "detect", "Waiting for X-Move");
x_move ← "Damit hat Mister X bestimmt nicht gezogen!";
fetch ("current_X_move", Detective_Number, ? x_move)
  xor ("commit_suicide") ⇒ ⟨Harakiri 234a⟩
  at Communication_TS
end fetch;
put_debug (2, "detect", ["X hat gezogen: ", x_move]);
```
This code is used in chunks 226 and 229.

Wenn der Spieler das Spiel vorzeitig beenden möchte, bekommen die Detektive einen Kamikaze–Auftrag, um in letzter Sekunde und mit aller Gewalt Mister X zu fangen.

⟨*Check Kamikaze–Auftrag* 233b⟩≡
```
put_debug (2, "detect", "Checking Kamikaze-Order");
Kamikaze ← true;
fetch ("commit_suicide")
  at Communication_TS
else
  put_debug (1, "detect", "ELSE-Fall in Checking Kamikaze-Order");
  Kamikaze ← false;
end fetch;

if Kamikaze then
  ⟨Harakiri 234a⟩
end if;
put_debug (2, "detect", "still living");
```
This code is used in chunk 229.

Nun wird ernst ...

⟨*Harakiri* 234a⟩≡
```
  put_debug (0, "detect", "harakiri!");
  deposit ["died"]
    at Communication_TS
  end deposit;

  return Ω;
```
This code is used in chunk 233.

An dieser Stelle werden die Sourcen der Strategie–Prozeduren eingebunden.

⟨*Strategie-Prozeduren* 234b⟩≡
```

@include "anfang.pst"
@include "random.pst"
@include "reference.pst"
@include "short.pst"
@include "helper.pst"
@include "schwachsinn.pst"
@include "sum.pst"
@include "anti-knubble.pst"
```
This code is used in chunk 229.

Für die erste Runde wäre es nett, wenn die initiale Position als aktuelle Position in das Blackboard übertragen wird.

⟨*Start–Position ermitteln* 234c⟩≡
```
meet ("initial_D_position", Detective_Number, ?position)
  at Communication_TS
end meet;

deposit ["current_position",  Detective_Number, position]
 at Blackboard_TS
end deposit;

meet ("number_of_X_moves", ?move_counter)
  at Communication_TS
end meet;
```

This code is used in chunk 229.

14.2.1 Sind alle fertig?: Await_Other_Detectives

Jeder Detektiv setzt das Flag *Calculation_Ready* im `Blackboard_TS`, wenn er nicht weiter planen kann (oder will), da er einen optimalen Plan erreicht hat (oder warum auch immer...). Konflikte dürfen dann bereinigt werden, wenn alle Detektive ihre Berechnung abgeschlossen haben, also ihr `calculation_ready` Flag gesetzt haben. Dies wird durch blockierende `fetch`-Operationen erreicht.

⟨*Prozeduren der internen Schnittstelle* 235⟩≡

```
procedure Await_Other_Detectives (rd dummy);

visible  all_ready ← true;
visible  nr        ← 1;

begin
  put_debug (1, "detect", "ENTER Await_Other_Detectives");

  for nr ∈ [1 .. number_of_detectives] do
    put_debug (2, "detect", ["Waiting for Detective ", nr]);
    fetch ("calculation_ready", nr) at Blackboard_TS
    end fetch;
    put_debug (2, "detect", ["Detective ", nr, "is ready!"]);
  end for;

  put_debug (2, "detect", "Alle Detektive sind fertig!");

  put_debug (1, "detect", "LEAVE Await_Other_Detectives");

end Await_Other_Detectives;
```

This definition is continued in chunks 236 and 237.
This code is used in chunks 226 and 229.

14.2.2 Trace–Informationen

Als Spezialität wird eine Debug–Ausgabe kreiert, die dazu dient, daß auf jeden Fall die Detektiv–Nummer mitausgegeben wird, damit die einzelnen Ausgaben den laufenden Prozessen zugeordnet wird. Damit nicht mehrere Detektive gleichzeitig auf die Datei schreibend zugreifen, wird über den `Blackboard_TS` ein Semaphor gehalten, die die Ausgabe atomisiert.

⟨*Prozeduren der internen Schnittstelle* 235⟩+≡

```
procedure put_debug (rd Level, rd Name, rd OneLongTuple);
hidden  log_file ← "Scotty.log";
hidden  blanks ← "                                                   ";
begin
@ifdef DEBUG
  if (Level ≤ Debug_Level) ∧ (Name ∈ Debug_Names) then

    fopen (log_file, "a");
    put (blanks(1..2*Level),
         "Tracing Det", Detective_Number,
         "in ", Name,
          OneLongTuple);
    fput (log_file, blanks(1..2*Level),
         "Tracing Det", Detective_Number,
         "in ", Name,
          OneLongTuple);
    fclose (log_file);

    log_file ← "Detective." + capsulated_str(Detective_Number) + ".log";
    fopen (log_file, "a");
    fput (log_file, blanks(1..2*Level),
         "Tracing Det", Detective_Number,
         "in ", Name,
          OneLongTuple);
    fclose (log_file);
  end if;
@endif
  pass;
end put_debug;
```

Hier wird das Tracing initialisiert .

⟨*Prozeduren der internen Schnittstelle* 235⟩+≡

```
procedure Init_Trace ();
visible trace_file ← "Scotty.trace";
hidden ok          ← true;
begin
  put ("ENTER Init_Trace");
  fopen (trace_file, "r")
    when io_file_error use io_error;
  if ok then
    put ("Init_Trace: Fget Debug_Level");
    fget (trace_file, Debug_Level);
    put ("Init_Trace: Fget Debug_Names");
    fget (trace_file, Debug_Names);
    fclose (trace_file);
  end if;
  put ("Debug-Names: ", Debug_Names);
  put ("Debug-Level: ", Debug_Level);
  put ("LEAVING Init_Trace");

handler io_error ();
begin
  put ("The file", trace_file, "does not exist!");
  ok ← false;
end io_error;
end Init_Trace;
```

14.3 Implementation: Auswahl der Strategie

Bei der Auswahl der Strategie sollte eine Spielstärkenregulierung vorgenommen werden. Die Bestimmung der Spielstärke wird durch die Auswahl der Strategie geregelt. Dies geschieht durch ein `meet` der Spielstärke. In Abhängigkeit vom ausgelesenen Level `level` erhält der Detektiv eine Strategie zugewiesen.

```
⟨Auswahl der Strategie 238⟩≡
  put_debug (2, "detect", "Lese Spielstaerke aus");
  ⟨Lese Spielstaerke aus 239⟩

  put_debug (2, "detect", ["hat Level", level]);
  if (level = 7) then
    put_debug (0, "detect", " Anti-Knubble-Strategy");
    strategy ← Anti_Knubble_Strategy;
  elseif (level = 6) then
    put_debug (0, "detect", " Mix-Strategy");
    strategy ← Mix_Strategy;
  elseif (level = 5) then
    put_debug (0, "detect", " Reference-Strategy");
    strategy ← Reference_Strategy;
  elseif (level = 4) then
    put_debug (0, "detect", " Sum-Strategy");
    strategy ← Sum_Strategy;
  elseif (level = 3) then
    put_debug (0, "detect", " Short-Strategy");
    strategy ← Short_Strategy;
  elseif (level = 2) then
    put_debug (0, "detect", " Random-Strategy");
    strategy ← Random_Strategy;
  elseif (level = 1) then
    put_debug (0, "detect", " Schwachsinn-Strategy");
    strategy ← Schwachsinn_Strategy;
  end if;
```

This code is used in chunk 229.

Der Level der Spielstärke wird ausgelesen.

⟨*Lese Spielstaerke aus* 239⟩≡

```
put_debug (3, "detect", ["Level vor meet", level]);
meet ("level", ? level)
    at Blackboard_TS
end meet;
put_debug (3, "detect", ["Level nach meet", level]);
```

This code is used in chunks 228a and 238.

14.4 Die Strategie in der Anfangsspielphase

Weil sich Mister X erst in der dritten Runde zum ersten Mal zeigt, sind die Detektive in dieser Spielphase über den Aufenthaltsort von Mister X einzig und allein auf Vermutungen angewiesen. Um vernünftige Züge durchzuführen, ist es sinnvoll, auf Orte zu gehen, die viele Anschlußmöglichkeiten haben, zentral liegen und eine schnelle Fortbewegungsmöglichkeit bieten. Wenn der Ort auch für Mister X besonders wertvoll ist, weil er beispielsweise mit Hilfe eines Blacktickets über die Themse fahren kann, so sollte der Detektiv auch möglichst in die Richtung eines solchen Ortes ziehen, um mit dem zweiten Zug dahin zu gelangen.

Die möglichen Zielorte für zwei Runden werden ermittelt, so daß eine Menge mit einer Folge von Orten der Gestalt `[[Ort1, Ticket], [Ort2, Ticket]]` ermittelt wird. Aus dieser Menge wird dann der `Ort2` ausgewählt, der am höchsten bewertet ist.

Die Prozedur `calculate_first_round` berechnet `calculate_destination` den Zielort des Detektivs `Detective_Number` in der ersten Runde. Die zur Zugberechnung erforderlichen Daten, wie die Startposition und der Ticketvorrat des Detektivs, werden aus dem Kommunikationstupelraum ausgelesen.

⟨*anfang.pst* 240⟩≡

```
procedure calculate_first_round (rd Detective_Number);

hidden all_tickets ← ∅;
hidden plan        ← [];
hidden position    ← 0;

begin
  put_debug (1, "anfang", "Enter First Round!");
  meet ("initial_D_position", Detective_Number, ? position)
    at Communication_TS
  end meet;
  meet ("ticket_count", Detective_Number, ? all_tickets)
    at Communication_TS
  end meet;

  plan ← calculate_destination (position, – x : x ∈ all_tickets″, 2);
  put_debug (2, "anfang", "After calculate_destination");
  put_debug (2, "anfang", ["The Plan is ", plan]);

  deposit ["Plan", Detective_Number, plan]
    at Blackboard_TS
  end deposit;
```

```
    put_debug (2, "anfang", "Der Plan ist in den Tupelraum eingetragen.");

    put_debug (1, "anfang", "Leave First Round!");

  end calculate_first_round;
```

This definition is continued in chunks 241 and 242.
Root chunk (not used in this document).

Genau wie in der ersten Runde wird auch der zweite Zug berechnet, mit dem einzigen Unterschied, daß nicht von der Startposition ausgegangen werden muß, sondern von der Position, auf die der Detektiv in der ersten Runde gezogen hat.

⟨*anfang.pst* 240⟩+≡

```
  procedure calculate_second_round (rd Detective_Number);

  hidden all_tickets ← ∅;
  hidden plan        ← [];
  hidden position    ← 0;

  begin
    put_debug (1, "anfang", "Enter calculate_second_round");
    meet ("current_position", Detective_Number, ? position)
      at Blackboard_TS
    end meet;
    put_debug (2, "anfang", ["Current Position is: ", position]);

    meet ("ticket_count", Detective_Number, ? all_tickets)
      at Communication_TS
    end meet;
    put_debug (2, "anfang", ["ticket count is: ", all_tickets]);

    plan ← calculate_destination (position, - x : x ∈ all_tickets", 1);
    put_debug (2, "anfang", ["Der berechnete Plan lautet: ", plan]);

    deposit ["Plan", Detective_Number, plan]
      at Blackboard_TS
    end deposit;
    put_debug (2, "anfang", "Der Plan ist in den Tupelraum eingetragen.");

    put_debug (1, "anfang", "Leave Second Round!");

  end calculate_second_round;
```

Die Hilfsfunktion `calculate_destination` berechnet aus einer vorgegebenen Position `StartPosition` und einem bestimmten Ticketvorrat `all_tickets` ein Ziel, das für Mister X von besonderer Bedeutung ist.

⟨*anfang.pst* 240⟩+≡

```
procedure calculate_destination (rd StartPosition,
                                 rd all_tickets,
                                 rd level);

hidden Underground ← domain([x(2): x ∈ board | x(1) = "underground"](1));
-- 194 ist eine denkbar unguenstige Position fuer den Detektiv. Hoffentlich
-- weiss das Mr. X nicht :-)
hidden River ← − y : y ∈ domain([x(2): x ∈ board | x(1) = "black"](1))
                          | y ≠ 194″;
hidden moves ← ∅;
hidden ziele ← ∅;
hidden ziel  ← ∅;
hidden ort   ← Ω;
hidden ort1 ← Ω;
hidden ort2 ← Ω;
hidden plan ← ∅;
hidden path ← Ω;
hidden new_tickets ← ∅;
hidden wert ← 0.0;

begin
    put_debug (1, "anfang", ["ENTER calculate_destination, Startposition = ",
               StartPosition, "all_tickets = ", all_tickets,
               "Level = ", level]);
    put_debug (2, "anfang", ["River = ", River, "U-Bahn = ", Underground]);
    moves ← help.Get_Possible_Moves (StartPosition);
    put_debug (2, "anfang", ["Possible_Moves: ", moves]);

    moves ← −x : x ∈ moves | x(2) ∈ − y(1) : y ∈ all_tickets | y(2) > 0″ ″;
    put_debug (2, "anfang", ["Ticket verfuegbar fuer: ", moves]);

    if (level = 2) then
      ⟨Berechne die Ziele im zweiten Zug 243⟩
    elseif (level = 1) ∨ (#plan = 0) then
      ⟨Berechne die Ziele in einem Zug 244⟩
    end if;
```

```
  put_debug (1, "anfang", "LEAVE calculate_destination");

  return plan;
end calculate_destination;
```

⟨*Berechne die Ziele im zweiten Zug* 243⟩≡

```
for ort ∈ domain(moves) do
  ziele ← help.Get_Possible_Moves (ort);

  ort2 ← - i : i ∈ (Underground + River) | i ∈ domain(ziele) ";

  if #ort2 ≠ 0 then
     ort1 ← ort;
     for ziel2 ∈ ort2 do
       for ticket1 ∈ moves-ort" do
          put_debug (2, "anfang", ["2. Ebene Ticket: ", ticket1]);
          new_tickets ← all_tickets;

          x ← new_tickets(ticket1);

          new_tickets(ticket1) ← x - 1;

          if ort1 ∈ Underground then
              wert ← 0.0;
            elseif ort1 ∈ River then
              wert ← 25.0;
            else wert ← 50.0;
          end if;
          if ziel2 ∈ River then
            wert * ← 2.0;
          end if;

          plan with ← [Detective_Number,
                          ort1, ticket1, new_tickets, ziel2, wert, 10.0];
        end for;
     end for;
   end if;
  end for;
```

This code is used in chunk 242.

Wenn nur über eine Ebene geplant werden soll oder aber über 2 Ebenen hinweg kein Plan gefunden worden ist, wird einfach der kürzeste Weg zwischen der aktuellen Position und einem der gewünschten Zielorte gesucht. Wenn es möglich ist, eine Position mit U–Bahn oder Fluß zu erreichen, dann wird dies auf jedem Fall realisiert. Andere Positionen sind dann uninteressant, es sei denn, es ist wieder eine der gesuchten U–Bahn oder Fluß Positionen.

⟨*Berechne die Ziele in einem Zug* 244⟩≡

```
 for ort ∈ domain(moves) do
   if ort ∈ (Underground + River) then
     put_debug (2, "anfang", "Kann nach U-Bahn oder Fluss ziehen!");
     sum ← 0;
   else
     put_debug (2, "anfang", "Berechne Abstandssumme zur U-Bahn / River");
     sum ← help.Get_Path_Sum (ort, Underground + River);
     put_debug (2, "anfang", ["Abstandssumme von ", ort,
                    "zur U-Bahn/River=", sum]);
   end if;

   for ticket ∈ moves-ort″ do
     put_debug (2, "anfang", ["1. Ebene Ticket: ", ticket]);
     new_tickets ← all_tickets;

     x ← new_tickets(ticket);

     new_tickets(ticket) ← x - 1;

     plan with ← [Detective_Number,
                    ort, ticket, new_ticket, 0
                  , if sum = 0 then
                          100.0
                    elseif sum < 2 then
                          80.0
                    elseif sum < 5 then
                          50.0
                    else
                          10.0
                    end if
                 , 10.0];
   end for;

   put_debug (2, "anfang", ["Neuer Plan lautet: ", plan]);
```

```
  end for;
```

This code is used in chunk 242.

14.5 Der Referenzalgorithmus

Der Referenzalgorithmus basiert auf der Ausarbeitung von Becker und Zell an der Uni Stuttgart [BZ90].

14.5.1 Prototyp der Moduldefinition

Aus den in Kapitel 14.1 genannten Gründen, waren wir gezwungen die Modulinstantiierung als einen Prototyp anzusehen, der verworfen werden muß.

⟨*Prototyp: reference.pst* 246⟩≡

```
module Refer
    import blackboard,
           communication,
           my_detective_nr,
           all_detectives,  -- Menge aller Detektiv-Nummern
           helper_module;   -- Modul einiger Hilfsfunktionen
    export Calculate_Move;

    visible blackboard_ts       ← Ω;
    visible communication_ts    ← Ω;
    visible my_current_position ← Ω;
    visible iterations ← 0; -- Anzahl der Aufrufe in einer Spielrunde
    visible max_iter   ← 3; -- max. Anzahl
    visible help       ← Ω;

  begin
    blackboard_ts    ← blackboard;
    communication_ts ← communication;
    help             ← helper_module;

   ⟨reference.pst 247⟩
   ⟨Refer-Hilfsprozeduren 256⟩
end Refer;
```

Root chunk (not used in this document).

14.6 Implementation des Referenzalgorithmus

Bei diesem Algorithmus berechnen die Detektive alle möglichen Züge und Folgezüge. Eine Kombination aus Zug und Folgezug wird als Plan betrachtet. Alle Pläne werden bewertet. Die Bewertung richtet sich nach dem Abstand zu möglichen Orten von Mister X. Als Positionen der anderen Detektive werden deren jeweils bester Plan angenommen. Jeder Detektiv schreibt seine Pläne danach in den Blackboard–Tupelraum.

⟨*reference.pst* 247⟩≡

```
procedure Refer_Calculate_Move(rd x_move, rd iterations);
    hidden all_detectives ← -1..number_of_detectives";
    hidden max_iter   ← 2;                 -- max. Anzahl von Iterationen

    hidden my_plans ← ∅;                   -- eigene Plaene
    hidden my_tickets ← ∅;                 -- eigener Ticketvorrat
    hidden my_position ← 0;                -- eigene Position
    hidden my_possible_moves ← ∅;          -- moegliche eigene Zuege
    hidden my_next_moves ← ∅;              -- Zukunftsziele
    hidden my_next_moves_2 ← ∅;            -- Zukunftsziele
    hidden d_positions_0 ← ∅;              -- Detektiv Positionen
    hidden d_positions_1 ← ∅;              -- Detektiv Positionen
    hidden d_best_plans ← ∅;               -- Plaene der anderen Detektive
    hidden d_plan ← [];                    -- einzelner Plan
    hidden x_tickets ← ∅;                  -- Tickets von Mister X
    hidden x_moves ← ∅;                    -- moegliche Zuege von Mister X
    hidden x_positions_0 ← ∅;              -- moegliche Orte von Mister X
    hidden x_positions_1 ← ∅;              -- moegliche Orte von Mister X

    hidden position ← 0;                   -- Hilfsvariable
    hidden path_value ← 0;                 -- Hilfsvariable
    hidden next ← 0;                       -- Hilfsvariable
    hidden current_sum ← 0;                -- Hilfsvariable
    hidden move ← [];                      -- Hilfsvariable
    hidden ticket ← [];                    -- Hilfsvariable
    hidden new_tickets ← ∅;                -- Hilfsvariable
    hidden best_value   ← 0.0;             -- Hilfsvariable
    hidden best_target  ← 0;               -- Hilfsvariable
    hidden second_value ← 0.0;             -- Hilfsvariable
    hidden second_target← 0;               -- Hilfsvariable
    hidden current_plan ← [];              -- Hilfsvariable
    hidden ready_flag ← false;             -- Rueckgabewert;
```

```
    -- Konstanten
    persistent constant float:"StdLib";

begin
    put_debug (1, "reference", "ENTER Refer_Calculate_Move !!!");

    if iterations = max_iter then
        put_debug (1, "reference", "NOTHING TO DO!");
        return true;
    end if;

    -- Eigene Position und Ticketvorrat lesen:
    my_tickets  ← help.Get_Tickets_Of (Detective_Number);

    ⟨Lese Blackboard Daten der Detektive 249⟩

    --  Positionen von Mr. X berechnen:
    x_positions_0 ← help.Get_X_Positions ();

    ⟨Erstelle Planliste 250⟩
    ⟨Berechne potentielle X-Zukunftsziele 251a⟩

    -- Berechne fuer jeden Plan Zukunftsziele und bewerte Endplan:
    for d_plan ∈ my_plans do
        put_debug (3, "reference",
                      ["Zukunftsplanung von plan: ", single_plan]);
        ⟨Berechne moegliche Zukunftsziele 251b⟩
        ⟨Bewerte Zukunftsziele 252⟩
        ⟨Suche bestes Zukunftsziel 253⟩
    end for;

    ⟨Bewerte Plaene 254a⟩
    ⟨Planliste aktualisieren 254b⟩
    ⟨Abbruchkriterium abtesten 255⟩

    put_debug (1, "reference", "LEAVING Refer_Calculate_Move !!!");
    return ready_flag;

    ⟨Refer-Hilfsprozeduren 256⟩

end Refer_Calculate_Move;
```

This code is used in chunk 246.

Zur Generierung eines Zuges benötigt ein Detektiv Daten der anderen Detektive. Die aktuellen Ticketvorräte, die aktuellen Positionen und die Pläne der anderen Detektive werden aus dem Blackboard–Tupelraum ausgelesen. Bei den eingelesenen Plänen wird der Ort des jeweils besten Plans in der weiteren Planung als Zielort des entsprechenden Plans berücksichtigt.

⟨*Lese Blackboard Daten der Detektive* 249⟩≡

```
put_debug (2, "reference", "Blackboard-Daten lesen:");

-- Positionen lesen:
d_positions_0 ← ∅;
for d ∈ all_detectives do
    position ← help.Get_Position_Of (d);
    d_positions_0 with← [d, position];
end for;
put_debug(3, "reference", ["Positionen: ", d_positions_0]);

-- Plaene lesen:
d_best_plans ← help.Read_Plans();
for d_plan ∈ d_best_plans | d_plan(1) ≠ Detective_Number do
    if d_plan(2) = 0 then
      position ← d_position_0(d_plan(1));
      d_positions_1 with← [d_plan(1), position];
    else
      d_positions_1 with← [d_plan(1), d_plan(2)];
    end if;
end for;
put_debug(3, "reference", ["Ziele: ", d_positions_1]);
```

This code is used in chunk 247.

Um eine Liste mit Plänen zu erstellen, werden aus der Menge aller durch Verbindungen erreichbaren Orte diejenigen entfernt, für die der Detektiv kein gültiges Ticket mehr hat. Zusätzlich müssen die Orte entfernt werden,auf denen Detektive stehen, die später am Zug sind.

⟨*Erstelle Planliste* 250⟩≡

```
position ← d_positions_0(Detective_Number);
my_possible_moves ← help.Get_Possible_Moves (position);
my_plans ← ∅;

for move ∈ my_possible_moves, ticket ∈ my_tickets |
          ((move(2) = ticket(1))
           ∧ (ticket(2) ≠ 0)) do
     new_tickets ← ∅;
     for new ∈ my_tickets | new ≠ ticket do
          new_tickets with← new;
     end for;

     new_tickets with← [ticket(1), ticket(2) - 1];
     d_plan ← [Detective_Number, move(1),
                 ticket(1), new_tickets, 0, 0.0, 0.0];
     my_plans with← d_plan;
end for;

for d_plan ∈ my_plans, d_pos ∈ d_positions_0 |
          ((d_plan(2) = d_pos(2))
           ∧ (d_pos(1) > Detective_Number)) do
     my_plans less← d_plan;
end for;
```

This code is used in chunk 247.

Um die Orte zu berechnen, an denen sich Mister X nach dem nächsten Zug aufhalten kann, werden für jeden im Augenblick möglichen Aufenthaltsort alle mit dem Ticketvorrat von Mister X erreichbaren Orte berechnet. Alle Orte, auf denen sich im Augenblick ein Detektiv befindet, werden aus der Liste entfernt. Zur Vereinfachung der Zukunftsplanung werden `2xTicket` hier nicht berücksichtigt.

⟨*Berechne potentielle X-Zukunftsziele* 251a⟩≡

```
x_tickets ← help.Get_Tickets_Of(0);
x_positions_1 ← ∅;

for position ∈ x_positions_0 do
    x_moves ← help.Get_Possible_Moves (position);

    for move ∈ x_moves, ticket ∈ x_tickets |
                ((move(2) = ticket(1)) ∧ (ticket(2) ≠ 0)) do
        x_positions_1 with← move(1);
    end for;
end for;

put_debug (3, "reference", ["Moegliche X-Ziele:", x_positions_1]);
```

This code is used in chunk 247.

Um alle Orte zu berechnen, zu denen ein Detektiv im übernächsten Zug gelangen kann, wird die Anfrage `possible_moves` in den Kommunikations–Tupelraum geschrieben und auf die Antwort gewartet. Aus der erhaltenen Liste müssen die Orte entfernt werden, die der Detektiv mit seinem Ticketvorrat nicht mehr erreichen kann.

⟨*Berechne moegliche Zukunftsziele* 251b⟩≡

```
my_next_moves ← help.Get_Possible_Moves (d_plan(2));

for move ∈ my_next_moves, ticket ∈ d_plan(4) |
                        ((move(2) = ticket(1)) ∧ (ticket(2) = 0)) do
    my_next_moves less← move;
end for;

put_debug (3, "reference",
          ["Zukunftziele fuer:", d_plan(2), my_next_moves]);
```

This code is used in chunk 247.

Um mögliche Zukunftsziele zu bewerten, wird die Abstandssumme des Detektives (vom Zukunftsziel) zu den möglichen Zukunftsorten von Mister X berechnet.

⟨*Bewerte Zukunftsziele* 252⟩≡

```
my_next_moves_2 ← ∅;

for next ∈ my_next_moves do
    path_value  ← help.Get_Path_Sum (next(1), x_positions_1);
    current_sum ← float(path_value);

    put_debug(3, "reference",
              ["Abstandssumme von", next(1), ":", current_sum]);
    my_next_moves_2 with← [next(1), current_sum];
end for;

my_next_moves_2 ← Refer_Scale_Future(my_next_moves_2);
put_debug (3, "reference",
           ["Bewertete Zukunftsziele:", my_next_moves_2]);
```

This code is used in chunk 247.

Aus der Liste der möglichen Zukunftsziele wird das Beste ausgewählt und in den Plan als Folgeziel eingetragen. Der Wertverlust, der entsteht, wenn ein Detektiv nicht auf das gewünschte Ziel kann, errechnet sich aus der Differenz zwischen den Werten des besten und des zweitbesten Folgeziels.

⟨*Suche bestes Zukunftsziel* 253⟩≡

```
best_value   ← 0.0;
best_target  ← 0;
second_value ← 0.0;
second_target← 0;

for p ∈ my_next_moves_2 do
    if p(2) ≥ second_value then
        second_target ← p(1);
        second_value  ← p(2);

        if second_value > best_value then
            second_value  ← best_value;
            second_target ← best_target;
            best_value  ← p(2);
            best_target ← p(1);
        end if;
    end if;
end for;
put_debug (3, "reference", ["Bestes Zukunftsziel ist", best_target]);

-- Bestes Zukunftsziel und Bewertung in Plan eintragen:
current_plan ← d_plan;
my_plans less← d_plan;

current_plan(5)← best_target;
current_plan(7)← best_value - second_value;
my_plans with← current_plan;
```

This code is used in chunk 247.

Um die Pläne zu bewerten, werden wird die Abstandssumme des Wunschortes zu allen möglichen Mister X-Orten berechnet.

⟨*Bewerte Plaene* 254a⟩≡
```
put_debug(2, "reference", "> Bewerte Plaene:");

for d_plan ∈ my_plans do
  path_value ← help.Get_Path_Sum (d_plan(2), x_positions_0);
  current_sum ← float(path_value);
  put_debug(3, "reference",
            ["Wert von Ziel", d_plan(2), "ist :", current_sum]);
  current_plan   ← d_plan;
  my_plans less  ← d_plan;
  current_plan(6)← current_sum;
  my_plans with  ← current_plan;
end for;

my_plans ← Refer_Scale_Plans(my_plans);

put_debug (2, "reference", "CALCULATED PLANS:");
Refer_Plan_Display(my_plans);
```
This code is used in chunk 247.

Die berechneten Pläne eines Detektives werden im Blackboard veröffentlicht, damit andere Detektive diese Daten mit in ihre Planung einbeziehen können.

⟨*Planliste aktualisieren* 254b⟩≡
```
fetch ("Plan", Detective_Number, ?)
    at Blackboard_TS
else
    put_debug (3, "reference", "> Keine eigenen alten Plaene vorhanden!");
end fetch;

deposit ["Plan",  Detective_Number, my_plans]
    at Blackboard_TS
end deposit;
```
This code is used in chunk 247.

Das Abruchkriterium ist eine Anzahl von Planungsschritten, die der Detektiv durchgeführt habe muß, damit der Algorithmus terminiert.

⟨*Abbruchkriterium abtesten* 255⟩≡

```
if iterations = max_iter then
  put_debug (2, "reference",
             ["Calculation ready after iteration", iterations]);
  ready_flag ← true;
  iterations ← 0;
else
  put_debug (2, "reference",
             ["Finished iteration", iterations]);
end if;
```

This code is used in chunk 247.

14.6.1 Hilfsprozeduren

Hier werden eine Reihe von Prozeduren und Funktionen definiert, die an verschiedenen Stellen benutzt werden und deshalb hier zusammengefaßt sind.

Die Prozedur `Refer_Scale_Plans` skaliert die Werte der erstellten Pläne, so daß die Summe der Werte eins ergibt. Die errechneten Werte werden von eins abgezogen, damit aus einem schlechten Ergebnis von Shortest-Path auch einen schlechter Plan erzeugt wird.

⟨*Refer-Hilfsprozeduren* 256⟩≡

```
procedure Refer_Scale_Plans(rd in_plans);
  hidden sum    ← 0.0;
  hidden plan   ← [];
  hidden new    ← [];
  hidden skaled ← ∅;
begin
  -- calculate sum of all plan values
  for plan ∈ in_plans do
    sum +← plan(6);
  end for;

  -- scale values
  if sum ≠ 0.0 then
    for plan ∈ in_plans do
      new ← plan;
      new(6) ← 2.0 - (new(6) / sum);
      skaled with← new;
    end for;
  end if;

  return skaled;
end Refer_Scale_Plans;
```

This definition is continued in chunks 257 and 258.
This code is used in chunks 246 and 247.

Die Prozedur `Refer_Scale_Future` skaliert die Werte der möglichen Zukunftsziele eines Detektives, so daß die Summe der Werte eins ergibt. Die errechneten Werte werden von eins abgezogen, damit aus einem schlechten Ergebnis von `Path-Sum` auch einen schlechter Plan erzeugt wird.

⟨*Refer-Hilfsprozeduren* 256⟩+≡

```
procedure Refer_Scale_Future(rd in_targets);
  hidden sum    ← 0.0;
  hidden target ← [];
  hidden new    ← [];
  hidden skaled ← ∅;
begin
  put_debug(2, "reference", ["Entered Scale_Future with:", in_targets]);

  -- calculate sum of all target values
  for target ∈ in_targets do
    sum +← target(2);
  end for;

  put_debug(3, "reference", ["Summe aller plaene:", sum]);

  -- scale values
  if sum ≠ 0.0 then
    for target ∈ in_targets do
      new ← target;
      new(2) ← 2.0 - (new(2) / sum);
      skaled with← new;
    end for;
  end if;

  put_debug(2, "reference", "Leaving Scale_Future");
  return skaled;
end Refer_Scale_Future;
```

Die Prozedur `Refer_Plan_Display` gibt die erstellten Pläne als Debug Meldung aus.

⟨*Refer-Hilfsprozeduren* 256⟩+≡
```
procedure Refer_Plan_Display(rd in_plans);
    hidden counter ← 1;
    hidden plan    ← [];
begin
    if in_plans = ∅ then
        put_debug(2, "reference",
                     "Keine ausfuehrbaren Plaene berechnet !");
    end if;

    for plan ∈ in_plans do
        put_debug(2, "reference",
                     ["Berechneter Plan Nr.",
                      counter, ":", plan(2), plan(3), plan(6)]);
        counter ← counter + 1;
    end for;
end Refer_Plan_Display;
```

14.7 Der Zufallsalgorithmus

Dieser Algorithmus berechnet Pläne für den nächsten Zug des Detektivs, der sich für diese Strategie entschieden hat, wobei als Kriterien für die Auswahl der Züge die Bewertung des Spielbrettes, die Position von Mister X und der Zufall berücksichtigt werden.

14.7.1 Prototyp der Moduldefinition

Aus den in Kapitel 14.1 genannten Gründen, waren wir gezwungen die Modulinstantiierung als einen Prototyp anzusehen, der verworfen werden muß.

⟨*Prototyp: random.pst* 259⟩≡

```
module Random
  import communication_TS,blackboard_TS,detektiv_nr;
  export Calculate_Move;
  visible last_pos ← 0;                  -- Detektiv-Position der letzten Runde
  visible det_moves ← [];                -- erlaubte Zuege des Detektivs
  visible calculate_move_counter ← 0; -- Anzahl Caculate_Move-Aufrufe
  visible max_count ← 3;
  visible com_TS ← communication_TS;
  visible bb_TS ← blackboard_TS;
  visible det_nr ← detektiv_nr;
  visible importmodul ← instantiate closure strategy_help
                          export com_TS, bb_TS;
                          import Resolve_Conflicts;
                        end instantiate;
begin
  pass;
  ⟨random.pst 260⟩
end Random;
```

Root chunk (not used in this document).

14.8 Implementation des Zufallsalgorithmus

Als taktische Feinheiten kommt der Realisierung des Zufallsalgorithmus hinzu, daß nur während der ersten drei Iterationen ein Plan berechnet wird. Am Ende der dritten Iteration schreibt der Detektiv seinen Plan fest.

⟨*random.pst* 260⟩≡

```
procedure Random_Calculate_Move(rd x_move, rd iterations);

  visible det_pos ← 0; -- Detektiv-Position
  hidden det_ticket_counts ← ∅;
  hidden det_possible_moves ← ∅;
  hidden possible_x_positions ← ∅;
  hidden best_det_moves ← []; -- beste Zuege des Detektivs
  hidden best_det_plans ← ∅; -- beste Plaene des Detektivs
  visible ready ← false;
  hidden constant max_iterations ← 3;

  -- Hilfsvariablen
  hidden max_move ← [];
  hidden random_move ← [];
  hidden buffer ← [];
  hidden method ← 0;
  hidden local_possible_moves ← ∅;
  hidden det_moves ← [];
  hidden local_det_moves ← [];
  hidden new_ticket_counts ← ∅;

begin
  put_debug (1, "random", "ENTER Random_Calculate_Move");
  if iterations ≤ max_iterations then
    ⟨Lese Daten aus Tupelraum 261a⟩
    ⟨Bestimmung aller nach Ticketvorrat moeglichen Zuege des Detektivs 261b⟩
    ⟨dynamische Zuggewichtung 262a⟩
    if det_moves ≠ [] then
      ⟨Sortieren der Zuege nach abnehmendem Gewicht 262b⟩
      ⟨Auswahl 3er zufaelliger oder der 3 besten Zuege 264a⟩
      ⟨Aufbau der Plan-Datenstruktur 264b⟩
      ⟨Plaene in Tupelraum eintragen 264c⟩
    end if;
    ⟨Abbruchbedingung 265⟩
  end if;
```

```
  put_debug (1, "random", "LEAVE Random_Calculate_Move");
  return ready;
  ⟨Hilfsprozeduren 263⟩
end Random_Calculate_Move;
```
This code is used in chunk 259.

⟨*Lese Daten aus Tupelraum* 261a⟩≡
```
put_debug (2, "random", "Lese Position des Detektivs");
det_pos ← help.Get_Position_Of (Detective_Number);
put_debug (2, "random", "Lese Ticketvorrat");
det_ticket_counts ← help.Get_Tickets_Of (Detective_Number);
put_debug (2, "random", "Lese moegliche Zuege");
det_possible_moves ← help.Get_Possible_Moves (det_pos);
put_debug (2, "random", "Lese moegliche X Positionen");
possible_x_positions ← help.Get_X_Positions ();
```
This code is used in chunk 260.

Ein Zug ist nur dann zulässig, wenn der Detektiv noch Tickets von der für diesen Zug erforderlichen Sorte besitzt.

⟨*Bestimmung aller nach Ticketvorrat moeglichen Zuege des Detektivs* 261b⟩≡
```
put_debug (2, "random", "Bestimmung der Zuege");
det_moves ← [pos_ticket : pos_ticket ∈ det_possible_moves,
                          ticket_count ∈ det_ticket_counts
                        | pos_ticket(2) = ticket_count(1) ∧
                          ticket_count(2) > 0];
put_debug (2, "random", ["det_moves: ", det_moves]);
```
This code is used in chunk 260.

Ein Zug wird mit der Anzahl möglicher Züge von der Zielposition bewertet. Der Bewertung liegt die Tatsache zugrunde, daß die Anzahl möglicher Züge auf U–Bahnstationen größer ist als auf Busstationen, von wo wiederum mehr Züge möglich sind als von reinen Taxistationen. Ist die Zielposition des Zuges allerdings gleichzeitig mögiche Position von Mister X, wird der Zug mit 100 maximal bewertet.

⟨*dynamische Zuggewichtung* 262a⟩≡
```
put_debug (2, "random", "Gewichtung der Zuege");
local_det_moves ← det_moves;
for move ∈ det_moves do
  local_possible_moves ← help.Get_Possible_Moves (move(1));
  local_det_moves ← [m : m ∈ local_det_moves | m ≠ move];
  local_det_moves with ← [move(1), move(2),
    if move(1) ∈ possible_x_positions then
      100 -- beliebiger Wert ¿ local˙possible˙moves
    else
      #local_possible_moves
    end if];
put_debug (2, "random", ["local_det_moves: ", local_det_moves]);
end for;
det_moves ← local_det_moves;
put_debug (2, "random", ["det_moves: ", det_moves]);
```
This code is used in chunk 260.

Das Tupel `det_moves` wird absteigend mit Hilfe der Funktion `Maximum` sortiert:

⟨*Sortieren der Zuege nach abnehmendem Gewicht* 262b⟩≡
```
put_debug (2, "random", "Sortierung der Zuege");
buffer ← [];
local_det_moves ← det_moves;
for i ∈ det_moves do
  max_move ← Maximum (local_det_moves);
  buffer with ← max_move;
  local_det_moves ← [move : move ∈ local_det_moves | move ≠ max_move];
end for;
det_moves ← buffer;
put_debug (2, "random", ["det_moves: ", det_moves]);
```
This code is used in chunk 260.

Mit der Funktion `Maximum` wird aus einer Menge von Zügen derjenige extrahiert, der die maximale Bewertung hat.

⟨*Hilfsprozeduren* 263⟩≡

```
procedure Maximum (rd moves);
hidden local_max_move ← [];
begin
  put_debug (3, "random", "Enter Maximum");
  if #moves ≠ 0 then
    local_max_move ← moves(1);
    for m ∈ moves do
      if local_max_move(3) < m(3) then
        local_max_move ← m;
      end if;
    end for;
  end if;
  put_debug (3, "random", ["Leaving Maximum returning", local_max_move]);
  return local_max_move;
end Maximum;
```

This code is used in chunk 260.

Per Zufall wird entschieden, ob die 3 ersten, d.h. besten, Züge oder 3 beliebige Züge in `det_moves` ausgewählt und als Pläne in `best_det_moves` eingetragen werden.

⟨*Auswahl 3er zufaelliger oder der 3 besten Zuege* 264a⟩≡
```
put_debug (2, "random", "Belege best_det_moves");
if #det_moves ≤ 3 then
  put_debug (2, "random", "#det_moves <= 3");
  best_det_moves ← [move(1..2) : move ∈ det_moves];
else
  method ← random(1.0);
  if method > 0.8 then
    put_debug (2, "random", "method < 0.2");
    local_det_moves ← det_moves;
    for i ∈ [1..3] do
      random_move ← random (local_det_moves);
      best_det_moves with ← random_move(1..2);
      local_det_moves ← [move : move ∈ local_det_moves
                                 | move ≠ random_move];
    end for;
  else
    best_det_moves ← [move(1..2): move ∈ det_moves(1..3)];
  end if;
end if;
```
This code is used in chunk 260.

⟨*Aufbau der Plan-Datenstruktur* 264b⟩≡
```
put_debug (2, "random", "Baue Planstruktur auf");
put_debug (2, "random", ["best_det_moves: ", best_det_moves]);
for move ∈ best_det_moves, ticket ∈ det_ticket_counts
  | move(2) = ticket(1) do
  new_ticket_counts ← det_ticket_counts;
  new_ticket_counts less ← ticket;
  new_ticket_counts with ← [ticket(1), ticket(2)-1];
  best_det_plans with ← [Detective_Number, move(1), move(2), new_ticket_counts, 0, 0.0, 0.0];
end for;
put_debug (2, "random", ["best det plans", best_det_plans]);
```
This code is used in chunk 260.

⟨*Plaene in Tupelraum eintragen* 264c⟩≡
```
put_debug (2, "random", "Plan eintragen in Blackboard_TS");
deposit ["Plan", Detective_Number, best_det_plans]
   at Blackboard_TS
end deposit;
```
This code is used in chunk 260.

Die Planung eines Zuges wird nach spätestens drei Durchläufen von `Caculate_Move` abgebrochen.

⟨*Abbruchbedingung* 265⟩≡
```
put_debug (2, "random", "Ueberpruefe Abbruchbedingung");
-- if iterations = max˙iterations then
iterations ← 0;
ready ← true;
-- end if;
```
This code is used in chunk 260.

14.9 Implementation des Shortest-Path Algorithmus

Der Shortest-Path Algorithmus basiert auf der Berechnung des kürzesten Weges zu den möglichen Positionen von Mister X. Die Detektive berechnen die kürzesten Wege zu allen möglichen Positionen von Mister X. Der jeweils erste Zug aus einem berechneten kürzesten Weg ist ein Plan. Der Wert des Plans richtet sich nach der Länge des berechneten Weges. Jeder Detektiv schreibt seine Pläne danach in das Blackboard.

⟨*short.pst* 266⟩≡

```
procedure Short_Calculate_Move(rd x_move, rd iterations);
  visible plans          ← ∅;            -- Menge der erstellten Plaene
  visible my_position    ← 0;            -- Eigene Position
  visible my_tickets     ← ∅;            -- Eigener Ticketvorrat;
  hidden x_positions     ← ∅;            -- Menge moeglicher X-Positionen
  hidden plan            ← [];           -- Plantupel
  visible ready_flag     ← false;        -- Rueckgabewert
begin
    put_debug (1, "short", "ENTER Short_Calculate_Move");

    if iterations > 1 then
      put_debug (1, "short", "Nothing to do!");
      return true;
    end if;

    my_position ← help.Get_Position_Of(Detective_Number);
    my_tickets  ← help.Get_Tickets_Of(Detective_Number);

    -- Fuer jede moegliche X-Position einen Plan erstellen:
    x_positions ← help.Get_X_Positions();
    for x ∈ x_positions do
        plan ← Make_Plan(x);
        if plan ≠ [] then
            plans with← plan;
        end if;
    end for;

    Remove_Invalid();

    Scale_Plans();

    Plan_Display();
```

```
    Finish_Short();

    put_debug (1, "short", "LEAVING Short_Calculate_Move");
    return ready_flag;

    ⟨Short-Hilfsprozeduren 267⟩

  end Short_Calculate_Move;
```

Root chunk (not used in this document).

14.9.1 Hilfsprozeduren

Hier werden eine Reihe von Prozeduren und Funktionen definiert:

Die Prozedur `Finish_Short` schreibt die neu erstellten Pläne des Detektives in den `Backboard-Tupelraum`, nachdem evetuell vorhandene alte Pläne entfernt worden sind und setzt dann ein ready-flag in den `Blackboard-Tupelraum`, damit der Algorithmus terminiert.

⟨*Short-Hilfsprozeduren* 267⟩≡

```
 procedure Finish_Short();
 begin
    put_debug (2, "short", "Planliste aktualisieren:");
    fetch ("Plan", Detective_Number, ?)
        at Blackboard_TS
    else
        put_debug (2, "short", ["Keine alten Plaene fuer Detektiv: ",
                                                  Detective_Number]);
    end fetch;

    deposit ["Plan",  Detective_Number, plans]
        at Blackboard_TS
    end deposit;

    put_debug (2, "short", ["Calculation ready after iteration",
                                                       iterations]);
    ready_flag ← true;

 end Finish_Short;
```

This definition is continued in chunks 268–71.
This code is used in chunk 266.

Die Prozedur `Remove_Invalid` enfernt aus der Menge der erstellten Plaene diejenigen, die einen Ort bercksichtigen auf dem ein Detektiv steht, der später zieht (das heißt, daß dieser eine größere Detektiv–Nummer hat).

⟨*Short-Hilfsprozeduren* 267⟩+≡

```
procedure Remove_Invalid();

  hidden all_detectives ← -1 .. number_of_detectives";
  hidden invalid        ← ∅;
  hidden position       ← 0;

begin
    put_debug (3, "short","Entferne ungueltige Plaene");

    for d ∈ all_detectives | d > Detective_Number do
        position ← help.Get_Position_Of(d);
        invalid with← position;
    end for;

    for p ∈ plans | p(2) ∈ invalid do
        put_debug (3, "short", ["Entferne:", p]);
        plans less← p;
    end for;

end Remove_Invalid;
```

Die Prozedur `Make_Plan` erstellt für den angegebenen Ort einen Plan. Dieser Plan besteht aus dem ersten Schritt des Pfades, den die Funktion Shortest–Path liefert.

⟨*Short-Hilfsprozeduren* 267⟩+≡

```
procedure Make_Plan(rd x_position);

    hidden new_tickets    ← ∅;
    hidden plan           ← [];
    hidden path           ← [];
    hidden path_value     ← 0;
    hidden float_value    ← 0.0;
    hidden moves          ← ∅;

    persistent constant float:"StdLib";

begin
    put_debug(3, "short", ["Berechne Plan fuer X-Ort:", x_position]);

    path ← help.Get_Shortest_Path(my_position, x_position, my_tickets);
    if (path = Ω) ∨ (path = []) then
        put_debug(3, "short", ["Ort unerreichbar:", x_position]);
        return [];
    else
        path_value ← #path;
        float_value ← float(path_value);
        put_debug(3, "short", ["Weglaenge zu", path, float_value]);

        for t ∈ my_tickets do
            if t(1) ≠ path(1)(2) then
                new_tickets with← t;
            else
                new_tickets with← [t(1), t(2) - 1];
            end if;
        end for;

        plan ← [Detective_Number, path(1)(1),
                 path(1)(2), new_tickets, 0, float_value, 0.0];
        return plan;
    end if;

end Make_Plan;
```

Die Prozedur `Plan_Display` gibt die erstellten Pläne als Debug Meldung aus.

⟨*Short-Hilfsprozeduren* 267⟩+≡

```
procedure Plan_Display();
  hidden counter ← 1;
  hidden plan    ← [];
begin
    put_debug(2, "short", "CALCULATED PLANS:");

    if plans = ∅ then
        put_debug(2, "short",
                     "Keine ausfuehrbaren Plaene berechnet !");
    end if;

    for plan ∈ plans do
        put_debug(2, "short",
                     ["Plan N.", counter, " :",
                      plan(2), plan(3), plan(6)]);
        counter ← counter + 1;
    end for;
end Plan_Display;
```

Die Prozedur `Scale_Plans` skaliert die Werte der erstellten Pläne, so daß die Summe der Werte eins ergibt. Die errechneten Werte werden von eins abgezogen, damit aus einem schlechten Ergebnis von Shortest–Path auch einen schlechter Plan erzeugt wird.

⟨*Short-Hilfsprozeduren* 267⟩+≡

```
procedure Scale_Plans();
  hidden sum    ← 0.0;
  hidden plan   ← [];
  hidden new    ← [];
  hidden skaled ← ∅;
begin
    -- calculate sum of all plan values
    for plan ∈ plans do
        sum +← plan(6);
    end for;

    -- scale values
    if sum ≠ 0.0 then
        for plan ∈ plans do
            new ← plan;
            new(6) ← 2.0 - (new(6) / sum);
            skaled with← new;
        end for;
    end if;

    plans ← skaled;
end Scale_Plans;
```

14.10 Implementation des Shortest–Path–Sum Algorithmus

Der Shortest–Path–Sum Algorithmus versucht einen Zug zu berechnen, mit dem der Detektiv allen möglichen Positionen von Mister Xmöglichst nah kommt. Das heißt, die Summe der Abstände zu diesen Positionen soll möglichst gering sein. Dieser Algorithmus bewertet alle Orte, die ein Detektiv erreichen kann. Der Wert richtet sich nach der Summe der Abstände zu allen möglichen Aufenthaltsorten von Mister X.

⟨*sum.pst* 272⟩≡

```
procedure Sum_Calculate_Move(rd x_move, rd iterations);
  hidden plans          ← ∅;           -- Menge der erstellten Plaene
  hidden skaled         ← ∅;
  hidden my_tickets     ← ∅;           -- Eigener Ticketvorrat
  hidden my_moves       ← ∅;           -- Menge der moeglichen Zuege
  hidden my_correct     ← ∅;
  hidden all_detectives← -1..number_of_detectives";
  hidden d_positions    ← ∅;           -- Menge der Detektiv-Positionen
  hidden x_positions    ← ∅;           -- Menge moeglicher X-Positionen

  hidden new_tickets    ← ∅;           -- Hilfsvariable
  hidden plan           ← [];          -- Hilfsvariable Plantupel
  hidden new            ← [];          -- Hilfsvariable Plantupel
  hidden ticket         ← [];          -- Hilfsvariable Tickettupel
  hidden d              ← 0;            -- Hilfsvariable
  hidden position       ← 0;            -- Hilfsvariable
  hidden move           ← [];          -- Hilfsvariable
  hidden det_pos        ← [];          -- Hilfsvariable
  hidden path_value     ← 0;            -- Hilfsvariable
  hidden sum            ← 0.0;          -- Hilfsvariable
  hidden all_sum        ← 0.0;          -- Hilfsvariable
  hidden counter        ← 1;            -- Hilfsvariable counter
  hidden constant C_DIRECT ← 0.1;

  persistent constant float:"StdLib";
begin
    put_debug (1, "sum", "ENTER Sum_Calculate_Move");

    -- Ueberpruefen, ob Berechnung beendet:
    if iterations > 1 then
        put_debug (1, "sum", "Nothing to do, leaving!");
        return true;
```

```
end if;

-- Alle Detektiv Positionen lesen:
for d ∈ all_detectives do
    position ← help.Get_Position_Of(d);
    d_positions with← [d,position];
end for;

-- Moegliche Positionen von Mister X:
x_positions ← help.Get_X_Positions ();

-- Eigenen Ticketvorrat lesen:
my_tickets ← help.Get_Tickets_Of(Detective_Number);

-- Moegliche eigene Zuege berechnen:
my_moves← help.Get_Possible_Moves(d_positions(Detective_Number));

for move ∈ my_moves, ticket ∈ my_tickets |
            ((move(2) = ticket(1)) ∧ (ticket(2) ≠ 0)) do
    my_correct with← move;
end for;

-- Plaene fuer X-Orte, die direkt erreicht werden koennen:
for move ∈ my_correct, ort ∈ x_positions |
            move(1) = ort do
    put_debug (3, "sum",
               ["Moeglicher X-Ort", move(1),
                "kann direkt erreicht werden!"]);
    new_tickets ← ∅;
    for ticket ∈ my_tickets do
        if ticket(1) = move(2) then
            new_tickets with← [ticket(1), ticket(2) - 1];
        else
            new_tickets with← ticket;
        end if;
    end for;

    plan ← [Detective_Number, move(1),
             move(2), new_tickets, 0, C_DIRECT, 0.0];
    plans with← plan;

    my_correct less← move;
```

```
end for;


for move ∈ my_correct, det_pos ∈ d_positions |
            ((move(1) ≠ det_pos(2))
             ∨ (det_pos(1) < Detective_Number)) do

    path_value ← help.Get_Path_Sum (move(1), x_positions);
    sum ← float(path_value);
    all_sum +← sum;
    put_debug (3, "sum", ["Abstand zu X von Ort",
               d_positions(Detective_Number), ":", sum]);

    new_tickets ← ∅;
    for ticket ∈ my_tickets do
        if ticket(1) = move(2) then
            new_tickets with← [ticket(1), ticket(2) - 1];
        else
            new_tickets with← ticket;
        end if;
    end for;

    plan ← [Detective_Number, move(1),
             move(2), new_tickets, 0, sum, 0.0];
    plans with← plan;
end for;

-- Werte der Plaene skalieren:
if all_sum ≠ 0.0 then
    for plan ∈ plans do
        new ← plan;
        new(6) ← 2.0 - (new(6) / sum);
        skaled with← new;
    end for;
    plans ← skaled;
end if;

-- Debug Ausgabe der berechneten Plaene:
put_debug (3, "sum", "CALCULATED PLANS:");
if plans = ∅ then
    put_debug(2, "sum", "NO PLANS !");
end if;
```

```
    for plan ∈ plans do
        put_debug(2, "sum", ["Plan ", counter, " :",
                             plan(2), plan(3), plan(6)]);
        counter ← counter + 1;
    end for;

    -- Plaene in Blackboard schreiben:
    put_debug (2, "sum", "Planliste aktualisieren:");
    fetch ("Plan", Detective_Number, ?)
        at Blackboard_TS
    else
        put_debug (2, "sum", "Keine alten Plaene!");
    end fetch;

    deposit ["Plan",  Detective_Number, plans]
        at Blackboard_TS
    end deposit;
    put_debug (2, "sum", ["Calculation ready after iteration",
                                                  iterations]);

    put_debug (1, "sum", "LEAVING Sum_Calculate_Move");
    return true;
  end Sum_Calculate_Move;
```

Root chunk (not used in this document).

14.11 Interaktion der Detektive

Damit die Detektive untereinander kommunzieren können, um beispielsweise Konflikte zu lösen, müssen einige Prozeduren zur Verfügung gestellt werden. Da diese Funktionen innerhalb von Modulen benötigt werden, werden sie ebenfalls in einem Modul verpackt, damit sie in die Strategie-Module importiert werden können.

⟨*helper.pst* 276⟩≡

```
module strategy_help
import  Comm_TS, BB_TS, PD;
export  Read_Plans, Resolve_Conflicts, Best_Plan,
         Get_Possible_Moves, Get_Shortest_Path, Get_Path_Sum, Publish_Move,
         Get_Tickets_Of, Get_Position_Of, Get_X_Positions;

visible Communication_TS ← Comm_TS;
visible Blackboard_TS← BB_TS;
visible number_of_detectives ← 0;
visible put_debug ← PD;

begin
  put_debug (1, "helper", "Initializing helper Module $Revision: 2.55 $");
  meet ("D_count", ? number_of_detectives)
    at Communication_TS
  end meet;

⟨Read_Plans 277⟩
⟨Resolve_Conflicts 279⟩
⟨Best_Plan 288⟩
⟨Possible_Plans 289⟩
⟨Strategie–Hilfen 290⟩

end strategy_help;
```

Root chunk (not used in this document).

14.11.1 Pläne auslesen: Read_Plans

Die Prozedur `Read_Plans` gibt eine Menge von Plänen aus. Diese Menge besteht aus den jeweils besten Plänen aller beteiligten Detektive. Liegen von einem Detektiv keine Pläne im Blackboard vor, wird für diesen Detektiv sein augenblicklicher Aufenthaltsort als Plan eingetragen.

⟨*Read_Plans* 277⟩≡

```
procedure Read_Plans ();
  hidden all_detectives ← -1..number_of_detectives";
  hidden d              ← 0;
  hidden best_plans     ← ∅;
  hidden all_plans      ← ∅;
  hidden plan           ← [];
  hidden position       ← 0;
  hidden tickets        ← ∅;
  hidden ticket         ← "";
begin
  put_debug (1, "helper", "Enter Help.ReadPlans");

  for d ∈ all_detectives do
     put_debug (2, "helper", ["Meeting Plans of ", d]);
     meet ("Plan", d, ? all_plans)
        at Blackboard_TS
     else
        all_plans ← ∅;
     end meet;
     put_debug (2, "helper", ["Meet returned with: ", all_plans]);

     plan ← Best_Plan(all_plans);

     if plan = [] then
       tickets  ← Get_Tickets_Of(d);
       position ← Get_Position_Of(d);

       plan ← [d, position, ticket, tickets, 0, 0.0, 0.0];
     end if;

     best_plans with← plan;
  end for;

  put_debug (1, "helper", "Leaving Help.ReadPlans");
  return best_plans;
```

```
end Read_Plans;
```

This code is used in chunk 276.

14.11.2 Konflikte lösen: Resolve_Conflicts

Nachdem alle Detektive die Planung für ihren nächsten Zug beendet haben, muß noch sichergestellt werden, daß es keine Konflikte mehr in den Plänen der Detektive mehr gibt.

⟨*Resolve_Conflicts* 279⟩≡

```
procedure Resolve_Conflicts (rd move_counter);
  visible all_positions    ← ∅;         -- aktuelle Detektivpositionen

  hidden all_plans          ← ∅;         -- Alle Plaene aller Detektive
  hidden resolved_plans     ← ∅;
  hidden all_detectives     ← [1..number_of_detectives];
  hidden possible_moves     ← ∅;
  hidden calculated_plans   ← ∅;
  hidden deleted_plans      ← ∅;
  hidden best_plan          ← [];
  hidden current_plans      ← [];
  hidden copy_of_current_plans ← [];
  hidden change_plans       ← [];
  hidden current_position   ← [];
  hidden position           ← 0;
  hidden delete_flag        ← false;
  hidden resolved_flag      ← false;
  hidden occupied_flag      ← false;

begin
  put_debug (1, "helper", "ENTER Help.ResolveConflicts");

  for detective ∈ all_detectives do
    position ← Get_Position_Of(detective);
    all_positions with← [detective, position];
  end for;

  all_plans ← Read_All_Plans();
  Display_All_Plans(all_plans);

  ⟨Konfliktlösungsalgorithmus 281⟩

  Publish_Move(resolved_plans, move_counter + 1);
  put_debug (1, "helper", "LEAVING Help.ResolveConflicts");

  ⟨Conflict-Hilfsprozeduren 285⟩
```

```
end Resolve_Conflicts;
```

This code is used in chunk 276.

Ein Konflikt tritt auf, wenn der beste eigene Plan den selben Zug enthält wie der beste Plan eines anderen Detektivs.

⟨*Konfliktlösungsalgorithmus* 281⟩≡

```
for current_detective ∈ all_detectives do
  put_debug (2, "helper", ["START Detektiv", current_detective]);

  own_plans ← all_plans(current_detective);
  resolved_flag ← false;

  while resolved_flag = false do
    best_plan ← Best_Plan(own_plans);
    put_debug (3, "helper", ["Eigener bester plan: ", best_plan]);

    delete_flag ← false;
    occupied_flag ← false;

    ⟨Ziele testen 284⟩

    if ¬ occupied_flag then
        put_debug(3, "helper", "Ziel ist nicht besetzt!");

        resolved_flag ← true;
        delete_flag ← false;

        put_debug(3, "helper",
                  "Ueberpruefe Plaene der anderen Detektive");

        for other_detective ∈ all_detectives |
                (other_detective > current_detective) do
            put_debug(3, "helper", ["Ueberpruefe Detektiv:",
                                          other_detective]);
            other_plans ← all_plans(other_detective);
            other_best_plan ← Best_Plan(other_plans);
            put_debug (3, "helper", ["Anderer bester plan:",
                                         other_best_plan]);
            if other_best_plan ≠ [] then
                if best_plan(2) = other_best_plan(2) then
                put_debug(2, "helper", ["Konflikt mit Detektiv",
                          other_detective, "wegen", other_best_plan(2)]);

                    if best_plan(6) < other_best_plan(6) then
```

```
                    put_debug(4, "helper",
                             "Eigener Plan ist schlechter!");
                    delete_flag ← true;
                    resolved_flag ← false;
                else
                    put_debug(4, "helper",
                             "Anderer Plan ist schlechter!");
                    if #other_plans > 1 then
                        put_debug(2, "helper",
                                 "Entferne Anderen Plan!");
                        all_plans less← [other_detective, other_plans];
                        other_plans less← other_best_plan;
                        all_plans with← [other_detective, other_plans];
                    else
                        put_debug(3, "helper",
                          "Anderer Plan kann nicht entfernt werden!");
                        delete_flag ← true;
                        resolve_flag ← false;
                    end if;
                end if;
            end if;
        else
            put_debug (3, "helper", "KEIN ANDERER PLAN!!");
        end if;
    end for;

    put_debug(3, "helper", "Alle anderen Plaene ueberprueft!");
end if;


if ((delete_flag = true) ∨ (occupied_flag = true)) then
    put_debug(2, "helper", "Entferne eigenen Plan!");
    all_plans less← [current_detective, own_plans];
    own_plans less← best_plan;
    all_plans with← [current_detective,own_plans];

    if delete_flag = true then
        deleted_plans with← best_plan;
    end if;

    -- best_plan darf nicht mehr verwertet werden!
    best_plan ← [];
```

```
      end if;

      if #own_plans = 0 then
          resolved_flag ← true;
      end if;

    end while;

    put_debug (3, "helper",
               ["Bester Plan nach Ueberpruefung:", best_plan]);
    if best_plan = [] then
        put_debug (3, "helper",
                   "Kein Plan vorhanden, suche Konflikt-Plan!");
        best_plan ← Best_Plan(deleted_plans);

        if best_plan = [] then
            put_debug(2, "helper",
                      ["Kein Zug moeglich fuer Detektiv", current_detective]);

            tickets ← Get_Tickets_Of(current_detective);

            best_plan ← [current_detective,
                          all_positions(current_detective),
                          "I am still standing!", tickets, 0, 1.0, 1.0];
        end if;
    end if;

    put_debug(2, "helper", ["Bereinigter Plan von Detektiv",
                             current_detective, best_plan]);
    resolved_plans with← best_plan;

    put_debug(2, "helper", ["ENDE Detektiv:", current_detective]);
  end for;

  put_debug(2, "helper", "CONFLICT FREE PLANS:");
  Display_D_Plans(resolved_plans);
```

This code is used in chunk 279.

Der Plan eines Detektives wird zunächst daraufhin überprüft, ob er überhaupt auf den gewünschten Ort ziehen darf. Dabei wird getestet, ob sich auf dem Ort ein Detektiv befindet, der erst später ziehen muß bzw. die Konfliktlösung schon berechnet hat, daß ein Detektiv der eher am Zug ist, auf diesen Ort zieht.

⟨*Ziele testen* 284⟩≡
```
put_debug (4, "helper", "Ueberpruefe Zielort:");

if best_plan ≠ [] then
    -- Orte ausschliessen, auf denen ein Detektiv steht, der spaeter am Zug ist
    for pos ∈ all_positions |
            ((pos(1) > current_detective)
             ∧ (pos(2) = best_plan(2))) do
        occupied_flag ← true;
        put_debug(3, "helper",
                  ["Ort", pos(2), "besetzt durch", pos(1)]);
    end for;

    -- Orte ausschliessen, auf die ein Detektiv zieht, der eher am Zug ist.
    for plan ∈ resolved_plans |
             ((plan(1) < current_detective)
              ∧ (plan(2) = best_plan(2))) do
        occupied_flag ← true;
        put_debug(3, "helper",
                  ["Ort", plan(2), "Ziel von", plan(1)]);
    end for;
end if;
```
This code is used in chunk 281.

Die Prozedur `Read_All_Plans` ließt alle Pläne aus dem `Blackboard-Tupelraum`. Zurückgeliefert wird eine Menge von Tupeln. Jeder Tupel besteht aus zwei Elementen: Eine Detektivnummer und eine Menge von Plänen des entsprechenden Detektives.

⟨*Conflict-Hilfsprozeduren* 285⟩≡
```
procedure Read_All_Plans ();
  hidden  all_detectives ← -1..number_of_detectives";
  hidden  all            ← ∅;
  hidden  possible_plans ← ∅;
  hidden  plans          ← ∅;
  hidden  d              ← 0;
begin
  put_debug(1, "helper", "ENTER Help.Read_All_Plans");

  for d ∈ all_detectives do
      put_debug(2, "helper", ["Fetching Plans of ", d]);
      fetch ("Plan", d, ? plans)
          at Blackboard_TS
      else
          put_debug(1, "helper", ["ELSE-Fall in Fetching Plans of ", d]);
          plans ← ∅;
      end fetch;

      possible_plans ← Possible_Plans(d, all_positions(d));
      plans +← possible_plans;

      all with← [d,plans];
  end for;

  put_debug(1, "helper", "LEAVING Help.All_Plans");
  return all;
end Read_All_Plans;
```
This definition is continued in chunks 286 and 287.
This code is used in chunk 279.

Die Prozedur `Display_D_Plans` gibt die Pläne der Detektive aus, die aus dem Blackboard ausgelesen werden.

⟨*Conflict-Hilfsprozeduren* 285⟩+≡

```
procedure Display_D_Plans (rd d_plans);
  hidden plan    ← [];
  hidden counter ← 1;
begin
  put_debug(1, "helper", "Enter Help.Display_D_Plans");

  if d_plans = ∅ then
    put_debug(1, "helper", "No Plans to display!");
  else
    for plan ∈ d_plans do
      put_debug(2, "helper", ["Plan Nr.", counter, " :",
                              plan(3), plan(2), plan(5), plan(6)]);
      counter ← counter + 1;
    end for;
  end if;

  put_debug(1, "helper", "Leaving Help.Display_D_Plans");
end Display_D_Plans;
```

Die Prozedur `Display_All_Plans` gibt die alle Pläne der Detektive aus. Dies beinhaltet die Pläne, die von den Detektiven selbst berechnet wurden und alle überhaupt möglichen Pläne.

⟨*Conflict-Hilfsprozeduren* 285⟩+≡

```
procedure Display_All_Plans (rd a_plans);
  hidden d_plans ← [];
  hidden plan    ← [];
  hidden counter ← 1;
begin
  put_debug(1, "helper", "Enter Help.Display_All_Plans");

  if a_plans = ∅ then
    put_debug(1, "helper", "No Plans to display!");
  else
    for d_plans ∈ a_plans do
      put_debug(2, "helper", ["Display plans of detective:", d_plans(1)]);
      counter ← 1;

      if d_plans(2) = ∅ then
        put_debug(1, "helper", "No Plans to display!");
      else
        for plan ∈ d_plans(2) do
          put_debug(2, "helper", ["Plan Nr.", counter, " :", plan(3), plan(2), plan(5), plan(6)]);
          counter ← counter + 1;
        end for;
      end if;
    end for;
  end if;

  put_debug(1, "helper", "Leaving Help.Display_All_Plans");
end Display_All_Plans;
```

14.11.3 Bestimmung des besten Plans

Mit der Funktion `Best_Plan` wird aus einer Menge von Plänen dejenigen extrahiert, der die maximale Bewertung hat.

⟨*Best_Plan* 288⟩≡

```
procedure Best_Plan(rd plans);
  hidden best_value ← 0.0;
  hidden best_plan  ← [];
  hidden plan       ← [];
begin
  put_debug (1, "helper", "Enter Help.Best_Plan");

  if #plans ≠ 0 then
    for plan ∈ plans do
      if plan = [] then
        put_debug (1, "helper", ["Best_PLan FOUND EMPTY PLAN in:", plans]);
      else
        if best_value ≤ plan(6) then
          best_value ← plan(6);
          best_plan  ← plan;
        end if;
      end if;
    end for;
  end if;

  put_debug (1, "helper", ["Leaving Help.Best_Plan returning", best_plan]);
  return best_plan;
end Best_Plan;
```

This code is used in chunk 276.

14.11.4 Bestimmung aller möglichen Pläne

Da ein Detektiv nur dann stehen bleiben darf, wenn er nicht gehen kann, sollen an dieser Stelle alle möglichen Pläne generiert werden, die mit dem aktuellen Ticketvorrat realisierbar sind. Diese bekommen dann Bewertung den Wert 1, so daß normalerweise immer die anderen, richtig berechneten Pläne für die Planung verwendet werden. `Resolve_Conflicts` kann sich aber auf diese Pläne abstützen, um kein fehlerhaftes Stehenbleiben zu erzeugen. Diese Pläne werden in den Tupelraum eingetragen.

⟨*Possible_Plans* 289⟩≡

```
procedure Possible_Plans (rd detective, rd position);
  hidden  plans        ← ∅;
  hidden  moves        ← ∅;
  hidden  tickets      ← ∅;
  hidden  new_tickets  ← ∅;
  hidden  ticket       ← "";
  hidden  ticket_count ← 0;
  hidden  x            ← "";
  hidden  target       ← 0;
begin
  put_debug (1, "helper",
            ["Enter Help.Possible_Plans with:  Detective",
                             detective, "Position", position]);

  tickets ← Get_Tickets_Of(detective);
  moves   ← Get_Possible_Moves (position);

  for target ∈ domain(moves) do
    for ticket  ∈ - x : x ∈ moves–target″ | x ∈ domain(tickets) ″ do
      new_tickets ← tickets;

      ticket_count ← new_tickets(ticket);

      if ticket_count > 0 then
        new_tickets(ticket) ← ticket_count - 1;

        plans with ← [detective,
                      target, ticket, new_tickets, 0, 0.0, 0.0];
      end if;
    end for;
  end for;

  if plans = ∅ then
```

```
      put_debug (1, "helper",
                 ["NO PLANS POSSIBLE FOR DETECTIVE:", detective]);
    end if;

    put_debug (1, "helper", "Leaving Help.Possible_Plans");
    return (plans);
  end Possible_Plans;
```
This code is used in chunk 276.

14.11.5 Kapselung der Anfragen zum Treiber

Die Anfragen zum Treiber unterscheiden sich von den anderen Tupelraum–Operationen dadurch, daß auf sie durch ein Paar von Operationen zusammengesetzt werden. Deshalb werden sie an dieser Stelle als Prozeduren angeboten. Zur Identifikation der Messages werden jeweils Atome generiert.

⟨*Strategie–Hilfen* 290⟩≡
  ⟨*Anfragen an den Treiber* 291⟩
  ⟨*Allgemeine Hilfsprozeduren* 296a⟩
This code is used in chunk 276.

Die Funktion `Get_Possible_Moves` liefert alle möglichen Orte, die von den Parameter `from_position` erreichbar sind.

⟨*Anfragen an den Treiber* 291⟩≡

```
procedure Get_Possible_Moves (rd from_position);

visible message_id ← newat();
hidden moves       ← ∅;

begin
  put_debug (1, "helper", ["ENTER Help.Get_Possible_Moves with:",
                                              from_position]);
  deposit ["possible_moves", message_id, from_position]
     at Communication_TS
  end deposit;

  put_debug (2, "helper", ["Help.Get_Possible_Moves: Waiting for",
                                                message_id]);
  fetch (message_id, ? moves) at Communication_TS
  end fetch;

  put_debug (1, "helper",
             ["LEAVE Help.Get_Possible_Moves with:", moves]);
  if type(moves) = set then
    return moves;
  else
    return - x : x ∈ moves ";
  end if;

end Get_Possible_Moves;
```

This definition is continued in chunks 292–94.
This code is used in chunk 290.

Die Funktion `Get_Shortest_Path` liefert den kürzesten Weg zwischen `from_pos` und `to_pos` für den Ticketvorrat `tickets`.

⟨*Anfragen an den Treiber* 291⟩+≡

```
procedure Get_Shortest_Path (rd from_pos, rd to_pos, rd tickets);

visible message_id ← newat();
hidden moves       ← [];

begin
  put_debug (1, "helper",
            ["ENTER Help.Get_Shortest_Path with ticket",
             tickets, "from", from_pos, "to", to_pos,
             "MessageId: ", message_id                ]);
  deposit ["shortest_path", message_id, from_pos, to_pos, tickets]
     at Communication_TS
  end deposit;

  put_debug (2, "helper",
            ["Help.Get_Shortest_Path: Waiting for:", message_id]);
  fetch (message_id, ? moves) at Communication_TS
  end fetch;

  put_debug (1, "helper",
            ["LEAVE Help.Get_Shortest_Path with:", moves]);
  return moves;
end Get_Shortest_Path;
```

Die Funktion `Get_Path_Sum` liefert die Summe der kürzesten Taxi-Wege zwischen den Knoten `from_pos` und `to_pos`, wobei `to_pos` die Menge aller möglichen Aufenthaltsorte von Mister X ist. Diese Summe dient zur Bewertung der Detektivpläne.

⟨*Anfragen an den Treiber* 291⟩+≡

```
procedure Get_Path_Sum (rd from_pos, rd
to_pos);

visible message_id ← newat();
hidden  sum        ← [];

begin
  put_debug (1, "helper",
             ["ENTER Help.Get_Path_Sum with:",
              from_pos, "to", to_pos, "MessageId:", message_id]);
  deposit ["path_sum", message_id, from_pos, to_pos]
     at Communication_TS
  end deposit;

  put_debug (2, "helper",
             ["Help.Get_Path_Sum: Waiting for:", message_id]);
  fetch (message_id, ? sum) at Communication_TS
  end fetch;

  put_debug (1, "helper",
             ["LEAVE Help.Get_Path_Sum with ", sum]);
  return sum;
end Get_Path_Sum;
```

Bei der Veröffentlichung des Planes muß (zumindest) an dieser Stelle berücksichtigt werden, daß die berechnetes Pläne falsch im Bezug auf die Spielregen sind. In diesem Fall wird eine Ausnahme ausgelöst. Im positiven Fall wird die aktuelle Position des Detektives in den Tupelraum `Blackboard_TS` eingetragen.

⟨*Anfragen an den Treiber* 291⟩+≡

```
procedure Publish_Move (rd plans, rd next_move);
hidden message_id ← −[d, newat()] : d ∈ −1..number_of_detectives″ ″;
hidden move_is_ok ← false;
hidden all_detectives ← [1..number_of_detectives];

begin
  put_debug (1, "helper",
             ["ENTER Help.Publish_Move with plans: ", plans]);

  for plan ∈ plans do
    put_debug (2, "helper",
               ["Gewuenschter Plan: ", "D_position",
                message_id(plan(1)), plan(1), plan(2), plan(3)]);
    deposit ["D_position",
             message_id(plan(1)), plan(1), plan(2), plan(3)]
      at Communication_TS
    end deposit;

  end for;

  for detective ∈ all_detectives, plan ∈ plans |
                               plan(1) = detective do
    put_debug (2, "helper", ["Help.Publish_Move: Waiting for:",
                             message_id(detective)             ]);

    fetch (message_id(detective), ? move_is_ok)
      at Communication_TS
    end fetch;

    if ¬ move_is_ok then
      put_debug (2, "helper", ["Incorrekter Plan", detective]);
      escape Incorrect_Move (detective, plan(2));
    else
      fetch ("current_position", detective, ?)
        at Blackboard_TS
      end fetch;
```

```
      put_debug (2, "helper", ["deposit current_position:",
                               plan(1), plan(2)           ]);
      deposit ["current_position", plan(1), plan(2)]
        at Blackboard_TS
      end deposit;
    end if;
  end for;


  fetch ("Runde", ?) at Blackboard_TS end fetch;
  put_debug (1, "helper", ["Gebe Runde ", next_move, " bekannt."]);
  deposit ["Runde", next_move]
    at Blackboard_TS
  end deposit;

  put_debug (1, "helper", "LEAVE Help.Publish_Move");

end Publish_Move;
```

14.11.6 Allgemeine Hilfsprozeduren

Die allgemeinen Hilfsprozeduren sind Tupleraumabfragen, die von allen Strategiefunktionen benutzt werden und deshalb als Prozedur gekapselt worden sind.

Die Funktion `Get_Tickets_Of` liefert den momentanen Ticketvorrat eines Detektives zurück. Der Ticketvorrat wird aus dem `Kommunikations-Tupelraum` gelesen.

⟨*Allgemeine Hilfsprozeduren* 296a⟩≡

```
procedure Get_Tickets_Of (rd detective);
  hidden tickets ← ∅;
begin
  put_debug (1, "helper", ["ENTER Get_Tickets_Of with: ", detective]);
  meet ("ticket_count", detective, ? tickets)
    at Communication_TS
  end meet;

  tickets ← –x: x ∈ tickets";

  put_debug (3, "helper", ["LEAVE Get_Tickets_Of with: ", tickets]);
  return tickets;
end Get_Tickets_Of;
```

This definition is continued in chunks 296b and 297.
This code is used in chunk 290.

Die Funktion `Get_Position_Of` liefert die momentane Position eines Detektives zurück. Die Position wird aus dem `Blackboard-Tupelraum` gelesen.

⟨*Allgemeine Hilfsprozeduren* 296a⟩+≡

```
procedure Get_Position_Of (rd detective);
  hidden position ← 0;
begin
  put_debug (3, "helper",
            ["ENTER Get_Position_Of with: ", detective]);

  meet("current_position", detective, ? position)
    at Blackboard_TS
  end meet;

  put_debug (3, "helper",
            ["LEAVE Get_Position_Of with: ", position]);
  return position;
end Get_Position_Of;
```

Um alle augenblicklich möglichen Aufenthaltsorte von Mister X zu erhalten, wird das Datum `possible_x_positions` aus dem `Kommunikations-Tupelraum` gelesen.

⟨*Allgemeine Hilfsprozeduren* 296a⟩+≡

```
procedure Get_X_Positions ();
  hidden x_positions ← ∅;
begin
  put_debug (3, "helper", "ENTER Get_X_Positions");
  meet("possible_X_positions", ? x_positions)
    at Communication_TS
  end meet;

  put_debug (3, "helper",
             ["LEAVE Get_X_Positions with:", x_positions]);
  return x_positions;
end Get_X_Positions;
```

14.12 Test der Planung

Während die Implementierung ein äußerst konstruktiver Vorgang in dem Software-Entwicklungsprozeß ist, hat das Testen destruktiven Charakter: Es ist die Tätigkeit, ein Programm nur deshalb auszuführen, um Fehler zu finden. Dennoch wird mit beiden Tätigkeiten das gleiche Ziel angestrebt, und zwar die Korrektheit des Gesamtsystems.

Es stellt sich natürlich die Frage, wie man ein Programm am besten testen soll. Diese Frage läßt sich wohl nicht allgemein beantworten. Die Struktur des Tests sollte so beschaffen sein, daß Fehler anhand von Kombinationen statischer Prüfungen, Analysen und Tests eliminiert werden können.

Ideal zur Fehlererkennung, -lokalisierung und -beseitigung wäre ein Debugger, um Schritt für Schritt die Ausführung des zu testenden Programms verfolgen zu können und mit Hilfe von Breakpoints die Inhalte von Variablen etc. anzuzeigen. Da wir solch ein Hilfsmittel für ProSet nicht zur Verfügung haben, müssen wir Testmengen mit typischen und untypischen Eingabewerten erstellen. Besondere Aufmerksamkeit ist dem Verhalten von Prozeduren und Funktionen bei der Eingabe von Grenzwerten zu widmen. Weil von der Treiberkomponente sichergestellt wird, daß die Funktionen der Planungskomponente nur mit gültigen Werten aufgerufen werden, haben wir uns entschlossen, die einzelnen Strategien in ihrer Gesamtfunktionalität und jede Hilfsfunktion gesondert zu testen.

Da während des Integrationstestes einschneidende Veränderungen an den Prozeduraufrufen und der Programmstruktur vorgenommen werden mußten, ist das Programm nicht funktionstüchtig, weil es nicht allzu gewinnbringend ist, diese Änderungen nach erfolgreichem Abschluß des Integrationstestes zu überprüfen. Vielmehr erscheint es sinnvoll, einen Testprogrammgenerator zu haben, der alle Prozedur- und Funktionsaufrufe aus dem Programm extrahiert, so daß nur noch zu testende Randbedingungen manuell erstellt werden müssen. Zu dieser Thematik sei auf den Abschnitt IV ab Seite 324 verwiesen.

Abschließend läßt sich zu dem Testkonzept sagen, daß es interessant ist, aber leider nicht ausreichend ist, was sich auch daran erkennen läßt, daß später entwickelte Strategien aufgrund der einschneidenden Änderungen im Konzept (vgl. Kap. 14 auf Seite 226) nicht in das Testprogramm integriert werden konnten.

14.12.1 Programmstruktur des Tests für die Planungskomponente

Das Programm besitzt die folgende Struktur:

⟨*testplan.pst* 298⟩≡

```
program testplan;

  ⟨Deklarationen 299⟩
```

```
begin

   ⟨Initialisierungen 300a⟩
   ⟨Test des Helper-Moduls 300c⟩
   ⟨Test der Anfangsspielphase 309b⟩
   ⟨Test der Zufallsstrategie 312⟩
   ⟨Test des Referenzalgorithmus 314⟩
   ⟨Hilfsfunktionen des Testprogrammes 304a⟩

@include "anfang.pst"
@include "random.pst"
@include "reference.pst"
@include "helper.pst"

end testplan;
```

Root chunk (not used in this document).

Zur Kommunikation der Detektive untereinander und der Detektive mit der Treiberkomponente müssen die zwei Tupelräume `Communication_TS` und `Blackboard_TS` deklariert werden. In diese Tupelräume müssen diverse Daten abgelegt werden, und zwar die Anzahl der Detektive `number_of_detectives` und die Spielstärke `level`. Für das Testprogramm werden zwei Detektive `detective1` und `detective2` benötigt. Weitere Deklarationen werden vor den Stellen angeführt, die diese benutzen.

⟨*Deklarationen* 299⟩≡

```
visible Blackboard_TS ← CreateTS (Ω);
visible Communication_TS ← CreateTS (Ω);
visible number_of_detectives ← 2;
visible detective1 ← 1;
visible detective2 ← 2;
visible level ← 7;
```

This definition is continued in chunks 300, 301b, 303a, 304b, 306b, 309a, 311, and 313.
This code is used in chunk 298.

⟨*Initialisierungen* 300a⟩≡

```
put ("Definitionen erfolgreich durchgefuehrt");

put ("Lege Detektivanzahl und Spielstaerke in Communication_TS ab");
deposit ["D_Count", number_of_detectives] at Communication_TS end deposit;
deposit ["level", level] at Communication_TS end deposit;
put ("Erfolgreich abgelegt");
```

This code is used in chunk 298.

⟨*Deklarationen* 299⟩+≡

```
visible help ← Ω;
```

⟨*Test des Helper-Moduls* 300c⟩≡

```
put ("Vor instantiate closure strategy_help");
help ← instantiate (closure strategy_help) export Communication_TS,
                                                   Blackboard_TS;
                                            import Read_Plans,
                                                   Resolve_Conflicts,
                                                   Get_Possible_Moves,
                                                   Get_Shortest_Path,
                                                   Publish_Move;
        end instantiate;

put ("Nach instantiate closure strategy_help");

⟨Test Read_Plans 301a⟩
⟨Test Conflict_Resolving 302⟩
⟨Test Get_Possible_Moves 303b⟩
⟨Test Get_Shortest_Path 305⟩
⟨Test Publish_Move 307⟩
```

This code is used in chunk 298.

Die Prozedur `Read_Plans` gibt die besten Pläne aller Detektive aus, und zwar ohne die Pläne des anfragenden Detektives. Es muß also eine Menge von Plänen in den Tupelraum `Blackboard_TS` gelegt werden, um danach den Prozedurtest durchführen zu können.

⟨*Deklarationen* 299⟩+≡

```
visible plan1 ← ∅;
visible plan2 ← ∅;
visible best_plans ← ∅;
```

⟨*Test Read_Plans* 301a⟩≡

```
put ("Lege Plaene im Tupelraum Blackboard_TS ab");
plan1 ← ‐[detective1, 50, "taxi",
            ‐["taxi", 5], ["bus", 1], ["underground", 0], ["black", 0]″,
             51, 80, 50],
          [detective1, 55, "bus",
            ‐["taxi", 6], ["bus", 0],["underground", 0], ["black", 0]″,
             56, 89, 10]″;
plan2 ← ‐[detective2, 150, "taxi",
            ‐["taxi", 5], ["bus", 1], ["underground", 0], ["black", 0]″,
             51, 80, 50],
          [detective2, 155, "bus",
            ‐["taxi", 6], ["bus", 0],["underground", 0], ["black", 0]″,
             56, 89, 10]″;
put ("Depositing [Plan", detective1, plan1,"]");
deposit ["Plan", detective1, plan1] at Blackboard_TS end deposit;
put ("Done");

put ("Depositing [Plan", detective2, plan2,"]");
deposit ["Plan", detective2, plan2] at Blackboard_TS end deposit;
put ("Done");

put ("Vor Read_Plans");
best_plans ← help.Read_Plans (detective1);
put ("Die besten Plaene lauten:"); put (best_plans);
put ("Testende von Read_Plans");
```

This code is used in chunk 300c.

Wenn die Pläne verschiedener Detektive miteinander kollidieren, muß mit der Funktion `Resolve_Conflicts` die Situation entschärft werden, indem die Pläne verglichen werden und der Plan mit der niedrigeren Bewertung gelöscht wird. Der Test muß zeigen, daß Konflikte erkannt und entsprechend beseitigt werden. Dazu müssen im Tupelraum `Blackboard_TS` verschiedene Pläne mehrer Detektive abgelegt sein, mit deren Hilfe die Funktion getestet wird.

⟨*Deklarationen* 299⟩+≡

```
visible plan3 ← ∅;
visible detective3 ← 3;
```

⟨*Test Conflict_Resolving* 302⟩≡

```
number_of_detectives +← 1;
put ("Lege Plaene von Detektiven in Blackboard_TS ab");
plan1 ←  ⌈[detective1, 55, "taxi",
               ⌈["taxi", 5], ["bus", 5], ["underground", 0], ["black", 0]⌉,
                56, 10, 5],
          [detective1, 50, "bus",
               ⌈["taxi", 5], ["bus", 5],["underground", 0], ["black", 0]⌉,
                51, 21, 10]⌉;
plan2 ←  ⌈[detective2, 150, "taxi",
               ⌈["taxi", 5], ["bus", 5], ["underground", 0], ["black", 0]⌉,
                151, 10, 5],
          [detective2, 155, "bus",
               ⌈["taxi", 5], ["bus", 5], ["underground", 0], ["black", 0]⌉,
                156, 20, 10]⌉;
plan3 ←  ⌈[detective3, 20, "taxi",
               ⌈["taxi", 5], ["bus", 5], ["underground", 0], ["black", 0]⌉,
                21, 10 , 5],
          [detective3, 50, "bus",
               ⌈["taxi", 5], ["bus", 5],["underground", 0], ["black", 0]⌉,
                41, 20, 10]⌉;
deposit ["Plan", detective1, plan1] at Blackboard_TS end deposit;
deposit ["Plan", detective2, plan2] at Blackboard_TS end deposit;
put ("Der Plan 1:"); put (plan1);
put ("Der Plan 2:"); put (plan2);

put ("Vor Resolve_Conflicts");
help.Resolve_Conflicts(plan3, detective3);
put ("Testende von Resolve_Conflicts");
```

This code is used in chunk 300c.

Test der Anfragen an den Treiber

Die aufgeführten Funktionen kommunizieren über den Tupelraum `Communication_TS` mit dem Treiber. Deshalb muß die Reaktion des Treibers auf eine solche Anfrage simuliert werden. Der Test der Funktionen besteht in einer Überprüfung der Reaktionen auf die simulierten Antworten und nicht in einem Test der dahinterliegenden Funktionalität, die vom Treiber zur Verfügung gestellt wird.

Die Funktion `Get_Possible_Moves` liefert alle die Orte, die von einem vorgegeben Ort aus erreichbar sind. Der Test wird so durchgeführt, daß die Funktion für verschiedene Orte aufgerufen wird und dann eine gewisse Antwortmenge auf die Anfrage hin zurückgegeben wird.

⟨*Deklarationen* 299⟩+≡

```
visible moves ← ∅;
visible zuege ← ∅;
```

⟨*Test Get_Possible_Moves* 303b⟩≡

```
put ("Starte Server fuer Get_Possible_Moves");
|| closure Get_Possible_Moves_Server();

put ("Vor Get_Possible_Moves");
moves ← help.Get_Possible_Moves (1);
put ("Von Ort 1 sind die folgenden Orte erreichbar:"); put (moves);

put ("Testende von Get_Possible_Moves");
```

This code is used in chunk 300c.

Der Server für die Anfrage `Get_Possible_Moves` simuliert die Antwort auf die Anfrage in der Form eines parallelen Prozesses.

⟨*Hilfsfunktionen des Testprogrammes* 304a⟩≡

```
procedure Get_Possible_Moves_Server();
hidden position ← 0;
begin
  fetch ("possible_move", ? message_id, ? position)
    at Communication_TS
  end fetch;

  zuege ← [ [9, "taxi"], [8, "taxi"], [46, "bus"], [46, "underground"]];
  deposit [message_id, zuege]
    at Communication_TS
  end deposit;
end Get_Possible_Moves_Server;
```

This definition is continued in chunks 306a, 308, and 310.
This code is used in chunk 298.

Die Funktion `Get_Shortest_Path` liefert den kürzesten Weg zwischen den Orten `von` und `nach` für den Ticketvorrat `tickets`. Der Test wird so durchgeführt, daß auf eine Anfrage hin, eine Antwort zurückgeliefert und das Verhalten der Funktion für diese Antwort getestet wird. Dabei ist das angegebene Ziel einmal erreichbar, einmal nicht.

⟨*Deklarationen* 299⟩+≡

```
visible von ← 0;
visible nach ← 0;
visible tickets ← ∅;
visible ticketvorrat ← ∅;
```

⟨*Test Get_Shortest_Path* 305⟩≡

```
put ("Starte Server fuer Get_Shortest_Path");
|| closure Get_Shortest_Path_Server (1);

von ← 19; nach ← 46;
tickets ← [["taxi", 5], ["bus", 4], ["underground", 2]];
put ("Vor Get_Shortest_Path");
moves ← help.Get_Shortest_Path (von, nach, tickets);
put ("Der kuerzeste Weg lautet:"); put (moves);

put ("Starte Server fuer Get_Shortest_Path");
|| closure Get_Shortest_Path_Server (2);

von ← 1; nach ← 199;
tickets ← [["taxi", 5], ["bus", 4], ["underground", 2]];
put ("Vor Get_Shortest_Path");
moves ← help.Get_Shortest_Path (von, nach, tickets);
put ("Der kuerzeste Weg lautet:"); put (moves);

put ("Testende von Get_Shortest_Path");
```

This code is used in chunk 300c.

Der Server für die Anfrage Get_Shortest_Path simuliert die Antwort auf die Anfrage in der Form eines parallelen Prozesses.

⟨*Hilfsfunktionen des Testprogrammes* 304a⟩+≡

```
procedure Get_Shortest_Path_Server(rd times);

hidden message_id ← newat();

begin
  put ("Server Get_Shortest_Path gestartet. Times:", times);
  fetch ("shortest_path", ? message_id, ? von, ? nach, ? ticketvorrat)
    at Communication_TS
  end fetch;
  put ("Server: Fetch ist erfolgreich!");

  if (times = 1) then
    put ("Times ist 1");
    zuege ← [ [9, "taxi"], [1, "taxi"], [8, "taxi"], [46, "bus"] ];
  else
    put ("Times ist nicht 1");
    zuege ← [Ω];
  end if;

  deposit [message_id, zuege]
    at Communication_TS
  end deposit;
  put ("Server Get_Shortest_Path shutdown.");
end Get_Shortest_Path_Server;
```

Mit Hilfe der Prozedur Publish_Move werden von einem Detektiv detective berechnete Pläne veröffentlicht. Ist der Zug nicht erlaubt, wird eine Ausnahme erzeugt, sonst wird der Plan im Tupelraum Blackboard_TS eingetragen. Um die Prozedur zu testen, muß folglich ein gültiger und ein ungültiger Plan veröffentlicht werden. Der ungültige Plan besteht in einem Zug auf den nicht vorhandenen Ort mit Nummer 777.

⟨*Deklarationen* 299⟩+≡

```
visible detective ← 0;
visible message_id ← 0;
```

⟨*Test Publish_Move* 307⟩≡

```
detective ← 1;

put ("Starte Server fuer Publish_Move");
|| closure Publish_Move_Server();

put ("Deposit current position 97");
deposit ["current_position", detective, 97]
  at Blackboard_TS
end deposit;

put ("Vor Publish_Move (1, 98)");
help.Publish_Move (detective, 98, "taxi")
    when Incorrect_Move use Publish_Move_Handler;

meet ("current_position", ? detective, ? position)
  at Blackboard_TS
end meet;

put ("Im Tupelraum Blackboard gefunden: D-Nr. und Position");
put (detective, position);

put ("Starte Server fuer Publish_Move");
|| closure Publish_Move_Server();

put ("Vor Publish_Move (1, 777)");
help.Publish_Move (detective, 777, "bus")
   when Incorrect_Move use Publish_Move_Handler;

put ("Testende von Publish_Move");
```
This code is used in chunk 300c.

Der Server für die Anfrage `Publish_Move` simuliert die Antwort auf die Anfrage in der Form eines parallelen Prozesses.

⟨*Hilfsfunktionen des Testprogrammes* 304a⟩+≡

```
procedure Publish_Move_Server();
  hidden message_id ← Ω;
  hidden detective  ← 0;
  hidden position   ← 0;
  hidden ok         ← false;
  hidden ticket     ← "";
begin
  put ("Server Publish_Move gestartet.");
  fetch ("D_position", ? message_id, ? detective, ? position, ?ticket)
    at Communication_TS
  end fetch;
  put ("Server: Fetch ist erfolgreich!");

  if position < 200 ∧ position > 0 then
    ok ← true;
    put ("Server: Position", position, "ist ok.");
  else
    ok ← false;
    put ("Server: Position", position, "ist NICHT ok.");
  end if;

  deposit [message_id, ok]
    at Communication_TS
  end deposit;
  put ("Server Publish_Move shutdown.");
end Publish_Move_Server;
```

Da Publish_Move eine Ausnahme generiert, wenn der gewünschte Zug ungültig ist, wird ein Handler definiert, der diese Exception abfängt und angemessen darauf reagiert.

⟨*Hilfsfunktionen des Testprogrammes* 304a⟩+≡

```
handler Publish_Move_Handler (detective, position);
begin
  put ("Publish_Move Exception Handler");
  put ("Falsche Position (", position, ") des Detektives ", detective") des Detektives );
  return;
end Publish_Move_Handler;
```

14.12.2 Test der Anfangsspielphase

Zum Testen der Anfangsspielphase muß für den Detektiv eine Startposition vergeben werden, und er muß einen Ticketvorrat zur Verfügung haben. Mit Hilfe dieser vorgegebenen Spielsituation wird getestet, ob die Funktion `calculate_destination` Fehler enthält. Die Funktionalität der Prozedur `calculate_first_round` und der Prozedur `calculate_second_round`, die `calculate_destination` aufrufen, wird hier im Testprogramm simuliert.

⟨*Deklarationen* 299⟩+≡

```
visible ticket_counts ← –["taxi", 10], ["bus", 8],
                          ["underground", 4], ["black", 0]";
visible Startposition ← 1;
```

⟨*Test der Anfangsspielphase* 309b⟩≡

```
put ("Teste Anfangsspielphase und starte den zugehoerigen Server");
|| closure Anfangsspiel_Server ();

put ("Lese Daten aus dem Kommunikationstupelraum aus");
meet ("initial_D_position", detective1, ? position)
  at Communication_TS
end meet;

meet ("ticket_count", detective1, ? tickets)
  at Communication_TS
end meet;

put ("Done... Berechne nun den Zug fuer die erste Runde");
plan ← calculate_destination (position, tickets, 2);
put ("Der Detektiv hat folgenden Plan:"); put (plan);

-- put ("Berechne Zug fuer die zweite Runde");
-- calculate˙destination (position, tickets, 1);

put ("Testende der Anfangsspielphase");
```

This code is used in chunk 298.

Der Server `Anfangsspiel_Server` bedient die Anfragen, die die Anfangsspielphase in Gestalt der Funktion `calculate_destination` an den Server stellt.

⟨*Hilfsfunktionen des Testprogrammes* 304a⟩+≡

```
procedure Anfangsspiel_Server ();

hidden first_possible_moves ← ⌐[8, "taxi"], [9, "taxi"],
                               [46, "bus"], [58, "bus"],
                               [46, "underground"]⌝;
hidden second_possible_moves ← ⌐[45, "taxi"], [74, "bus"]⌝;
hidden position ← 0;

begin
  put ("Server gestartet, lege Daten im Communication_TS ab");
  deposit ["initial_D_position", detective1, Startposition]
    at Communication_TS
  end deposit;

  deposit ["ticket_count", detective1, ticket_counts]
    at Communication_TS
  end deposit;
  put ("Daten sind abgelegt");

  fetch ("possible_move", ? message_id, ? position)
    at Communication_TS
  end fetch;
  deposit [message_id, first_possible_moves]
    at Communication_TS
  end deposit;

  -- Diese for-Schleife muss auf die einzelnen Orte abgestimmt werden
  for ort ∈ domain (first_possible_moves) do
    fetch ("possible_move", ? message_id, ? position)
      at Communication_TS
    end fetch;
    deposit [message_id, second_possible_moves]
      at Communication_TS
    end deposit;
  end for;

  put ("Server Anfangsspiel shutdown");
```

```
end Anfangsspiel_Server;
```

14.12.3 Test der Zufallsstrategie

Um die Zufallsstrategie testen zu können, müssen einige Werte festgelegt werden. Mit einer simulierten Spielsituation wird dann getestet, ob der Algorithmus entsprechend der Spezifikation arbeitet.

⟨*Deklarationen* 299⟩+≡

```
visible d_pos ← 0;
visible d_possible_moves ← ∅;
visible d_ticket_counts ← 0;
visible board_v ← ∅;
visible plan4 ← ∅;
visible detective4 ← 4;
visible Detective_Number ← detective4;
visible last_detective_position ← 0;
```

⟨*Test der Zufallsstrategie* 312⟩≡

```
put ("Testbeginn der Zufallsstrategie");
number_of_detectives +← 1;
d_pos ← 11;
d_possible_moves ← –[13, "taxi"], [16, "bus"], [23, "bus"], [10, "taxi"]";
d_ticket_counts ← –["taxi", 6], ["bus", 4], ["underground", 0], ["black", 0]";
board_v ← –[13,98,0,0], [16,100,0,0], [23,110,0,0], [10,70,0,0]";

put ("Ablegen von Testbeispielen im Communication_TS");

deposit ["current_position", detective4, d_pos]
     at Communication_TS
end deposit;
put ("Der Detektiv ",detective4," steht auf Feld ",d_pos);

deposit ["possible_moves", d_pos, d_possible_moves]
     at Communication_TS
end deposit;
put ("Er kann von da aus ziehen auf: "); put (d_possible_moves);

deposit ["ticket_counts", detective4, d_ticket_counts]
     at Communication_TS
end deposit;
put ("Er besitzt die folgende Tickets:"); put (d_ticket_counts);

deposit ["board", board_v]
     at Communication_TS
end deposit;
put ("Es wird mit folgenden Brettwerten der Zug ausgewaehlt:");
put (board_v);

put ("Vor Random_Calculate_Move");
Random_Calculate_Move (1,1);
put ("Nach Random_Calculate_Move");

fetch ("Plan", detective4, ? plan4) at Blackboard_TS end fetch;
put ("Der Detektiv zieht folgendermassen:"); put (plan4);
put ("Testende der Zufallsstrategie");
```

This code is used in chunk 298.

14.12.4 Test des Referenzalgorithmus

Wie der Zufallsalgorithmus wird der Referenzalgorithmus in seiner Gesamtfunktionalität getestet. Dafür sind zunächst einige Initalisierungen durchzuführen.

⟨*Deklarationen* 299⟩+≡

```
visible my_detective_nr ← 1;
visible all_detectives  ← −1,2,3″;
visible my_current_position ← 1;
visible iterations ← 0;
visible max_iter   ← 1;

hidden my_plans ← ∅;               -- eigene Plaene
hidden my_tickets ← ∅;             -- eigener Ticketvorrat
hidden my_possible_moves ← ∅;      -- moegliche eigene Zuege
hidden my_next_moves ← ∅;          -- moegliche eigene Zukunftszuege
hidden x_tickets ← ∅;              -- Tickets von Mister X
hidden possible_x_moves ← ∅;       -- moegliche Zuege von Mister X
hidden next_x_positions ← ∅;       -- moegliche Zukunftsorte von Mister X
hidden current_x_positions ← ∅; -- moegliche Orte von Mister X
hidden current_d_positions ← ∅; -- Positionen der anderen Detektive


-- Hilfsvariablen für den Referenzalgorithmus


hidden path_value ← 0;    -- Wert des Pfades
hidden double_moves ← ∅;        -- Doppelzuege
hidden plan ← [];                 -- Planliste
hidden current_plan ← [];         -- gerade bearbeiteter Plan
hidden path ← [];               -- kuerzester Weg
hidden current_sum ← 0.0;        -- Wertsumme zur Ortbewertung
hidden detect ← 0;               -- Detektiv-Nummer
hidden position ← 0;             -- Detektiv-Position
hidden new_tickets ← ∅;         -- Menge von Ticketvorraeten
hidden best_value ← 0.0;         -- beste Zukunftszielbewertung
hidden best_target ← 0;          -- bestes Zukunftsziel
hidden second_value ← 0.0;       -- zweitbeste Zukunftszielbewertung
hidden second_target ← 0;        -- zweitbestes Zukunftsziel
```

⟨*Test des Referenzalgorithmus* 314⟩≡

```
put ("Testbeginn des Referenzalgorithmus");

iterations +← 1;

put("Get positions and tickets");
my_tickets ← Test_Get_Tickets_Of (my_detective_nr);

my_current_position ← Test_Get_Position_Of (my_detective_nr);


-- Lese Positionen der anderen Detektive;

current_d_positions ← ∅;
for detect ∈ (all_detectives less my_detective_nr) do
  position ← Test_Get_Position_Of (detect);
  current_d_positions with← [detect, position];
end for;

put("Get my possible moves");
my_possible_moves ← Test_Get_Possible_Moves (my_current_position);

put("Get possible x positions");
current_x_positions ← Test_Get_X_Positions ();


-- Erstelle Planliste;

put("Create my plan set");
new_tickets ← my_tickets;
my_plans ← ∅;
for move ∈ my_possible_moves, ticket ∈ my_tickets |
            ((move(2) = ticket(1)) ∧ (ticket(2) ≠ 0)) do
  new_tickets less← ticket;
  new_tickets with← [ticket(1), ticket(2) - 1];

  plan ← [my_detective_nr, move(1), ticket(1), new_tickets, 0, 0.0, 0.0];
  my_plans with← plan;
end for;
put("Created my plan set successful");
```

```
for plan ∈ my_plans, position ∈ current_d_positions |
            plan(2) = position(2) do
  if position(1) > my_detective_nr then
    my_plans less← plan;
  end if;
end for;
put("Removed invalid plans");


-- Berechne potentielle Zukunftsziele von Mister X;

put("Compute possible next X positions");
x_tickets ← Test_Get_Tickets_Of (0);

for position ∈ current_x_positions do
  possible_x_moves ← Test_Get_Possible_Moves (position);

  new_tickets ← x_tickets;
  next_x_positions ← ∅;

  double_moves ← ∅;
  for ticket ∈ x_tickets | ((ticket(1) = "twice") ∧ (ticket(2) ≠ 0)) do
    for p ∈ possible_x_moves do
      double_moves ← Test_Get_Possible_Moves (p (1));
    end for;
  end for;
  possible_x_moves +← double_moves;

  for move ∈ possible_x_moves, ticket ∈ x_tickets |
              ((move(2) = ticket(1)) ∧ (ticket(2) ≠ 0)) do
    new_tickets less← ticket;
    new_tickets with← [ticket(1), ticket(2) - 1];

    next_x_positions with← [move(1), new_tickets]; -- Wozu Ticketvorrat?
  end for;

  for p ∈ next_x_positions, c ∈ current_d_positions |
           ((p(1) = c(2)) ∨ (p(1) = my_current_position)) do
    next_x_positions less← p;
  end for;
end for;
```

```
-- Berechne moegliche eigene Zukunftsziele;

put("Compute own plans");
for single_plan ∈ my_plans do
  my_next_moves ← Test_Get_Possible_Moves (single_plan (2));

  for m ∈ my_next_moves, t ∈ single_plan(4) |
                                ((m(2) = t(1)) ∧ (t(2) = 0)) do
    my_next_moves less← m;
  end for;


  -- Berechne Abstandssumme;

  put("Compute way");
  for p ∈ my_next_moves do
    current_sum ← 0.0;

    for x ∈ next_x_positions do
      path ← Test_Get_Shortest_Path (p(2), x(1), single_plan (4));

      if (path ≠ Ω) ∧ (path ≠ []) then
        path_value ← #path;
        current_sum +← path_value;
      elseif path = Ω then
        current_sum +← 30; -- Strafwert
      end if;
    end for;

    if current_sum ≠ 0.0 then
      current_sum ← 100 / current_sum;
    end if;

    my_next_moves less← p;
    my_next_moves with← p + [current_sum];
  end for;


  -- Suche bestes Zukunftsziel;
```

```
  put("Take best target");
  best_value    ← 0.0;
  second_value ← 0.0;
  second_target ← 0;
  best_target   ← 0;

  for p ∈ my_next_moves do
    if p(3) ≥ second_value then
      second_target ← p(1);
      second_value  ← p(3);
      if second_value > best_value then
        second_value  ← best_value;
        second_target ← best_target;
        best_value  ← p(3);
        best_target ← p(1);
      end if;
    end if;
  end for;

  current_plan ← single_plan;
  my_plans less← single_plan;

  current_plan(5)← best_target;
  current_plan(7)← best_value - second_value;

  my_plans with← current_plan;


end for;  -- for single_plans in my_plans;


-- Bewerte Plaene;

put("Compute plan value");
current_sum←0.0;
for single_plan ∈ my_plans do
  for x ∈ current_x_positions do
    path ← Test_Get_Shortest_Path (single_plan (2), x, single_plan (4));
    if (path ≠ Ω) ∧ (path ≠ []) then
      path_value ← #path;
      current_sum ← current_sum + path_value;
    elseif path = Ω then
```

```
      current_sum +← 30; -- Strafwert
    end if;
  end for;

  if current_sum ≠ 0.0 then
    current_sum ← 100 / current_sum;
  end if;

  current_plan ← single_plan;
  my_plans less← single_plan;

  current_plan(6) ← current_sum;

  my_plans with← current_plan;
end for;


Test_Resolve_Conflicts (my_plans, my_detective_nr);


-- Abbruchkriterium abtesten;

put("Teste Abbruchkriterium");
if iterations = max_iter then
  put("Calculation ready!");
  put("Plans:");
  for single_plan ∈ my_plans do
    put("Detektiv :", single_plan(1));
    put("Ziel     :", single_plan(2));
    put("Ticket   :", single_plan(3));
    for t ∈ single_plan(4) do
      put("T-Rest   :", t(1), t(2));
    end for;
    put("F-Ziel   :", single_plan(5));
    put("Wert     :", single_plan(6));
    put("Verlust  :", single_plan(7));
  end for;
  iterations ← 0;
end if;

put ("Testende des Referenzalgorithmus");
```
This definition is continued in chunk 320.

This code is used in chunk 298.

Um den Treiber zu simulieren, werden einige Funktionen benötigt; diese Funktionen werden mit speziellen Testdaten ausgeführt und deswegen hier gesondert aufgeführt, wobei die rellen Prozedurnamen mit dem Zusatz *Test_* versehen sind.

⟨*Test des Referenzalgorithmus* 314⟩+≡

```
-- local procedures;

procedure Test_Get_Tickets_Of (rd detective);

hidden ticket← ∅;

begin
  put("Test_Get_Tickets_Of : ", detective);
  ticket with← ["black", 1];
  ticket with← ["bus", 1];
  ticket with← ["taxi", 1];
  ticket with← ["underground", 1];
  ticket with← ["twice", 1];

  return ticket;
end Test_Get_Tickets_Of;


procedure Test_Get_Position_Of (rd detective);

hidden current_pos ← 1;

begin
  put("Test_Get_Position_Of : ", detective);

  return current_pos;
end Test_Get_Position_Of;


procedure Test_Get_X_Positions ();

hidden current_x_positions ← ∅;

begin
  put("Test_Get_X_Positions : ");
  current_x_positions with← 31;
```

```
  current_x_positions with← 18;

  return current_x_positions;
end Test_Get_X_Positions;


procedure Test_Get_Possible_Moves (rd from_position);

hidden moves       ← ∅;

begin
  put("Test_Get_Possible_Moves: ", from_position);
  moves with← [8, "bus"];
  moves with← [9, "bus"];

  return moves;
end Test_Get_Possible_Moves;


procedure Test_Resolve_Conflicts (rd my_plans, rd detective);
begin
  put("Resolving conflicts, doing nothing!");

  return;
end Test_Resolve_Conflicts;


procedure Test_Get_Shortest_Path (rd from_pos, rd to_pos, rd tickets);

hidden moves       ← [];

begin
  put("Test_Get_Possible_Moves: ", from_pos, to_pos);
  if from_pos ≠ 1 then
    put("invalid start for shortest path: ", from_pos);
  end if;

  if to_pos = 8 then
    moves with← [8,"bus"];
  end if;
  if to_pos = 9 then
    moves with← [9,"bus"];
```

```
  end if;

  return moves;
end Test_Get_Shortest_Path;
```

Teil IV

Evaluation und Verbesserungsvorschläge

15. Die zweite Seminarphase

Zu Beginn der zweiten PG-Hälfte im Wintersemester wurde eine weitere Seminarphase abgehalten, um Grundlagen für das Spezifizieren von Werkzeugen vorzustellen.

Die zwölf Seminarvorträge umfaßten folgende Themenbereiche:

- Transformationelles Programmieren
- Software-Entwicklungsumgebungen
- Formale Spezifikation

15.1 Transformationelles Programmieren (Stefan Hedtfeld)

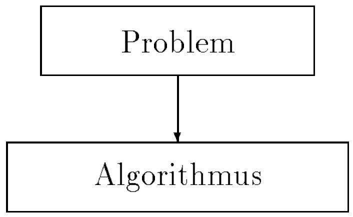
Verwendete Quellen

Als Quellen standen mir die Artikel von Partsch/Steinbrüggen [HP83] und Partsch/Möller [HP87] zur Verfügung.

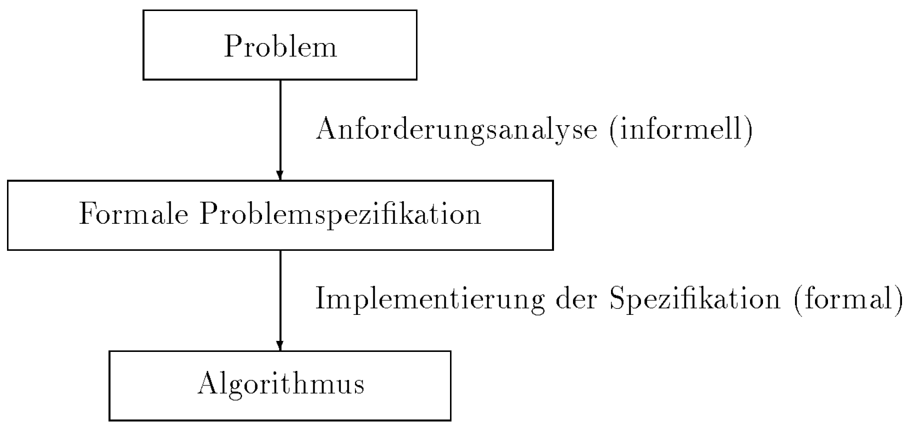
Motivation

Wie schwer es in der Praxis ist, korrekte Programme zu schreiben, zeigen die vielen Ansätze in der Theorie, Fehler in Programmen zu entdecken und zu eliminieren bzw. Fehler überhaupt zu vermeiden. Wie kommen Fehler überhaupt in Programme?

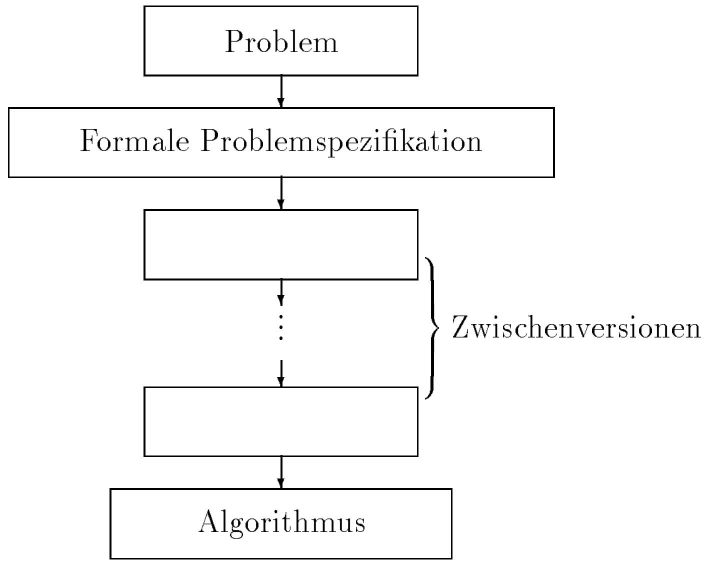
Zum einen sind dies sicher die *normalen* menschlichen Versehen, wie falsch verwendete Variablen, nicht präzise Bedingungen oder ähnliches. Daneben besteht das Problem, daß beim Programmieren ein Programm aus einer — vermutlich nicht vollständigen — informellen Spezifikation entstehen soll.



Dieser Schritt ist in der Regel zu groß, um fehlerfrei realisiert werden zu können. Also fügt man einen Zwischenschritt ein, so daß wir nun folgendes Vorgehen haben:



Aber auch der Sprung von der formalen Problemspezifikation zum Algorithmus ist noch zu groß, so daß man nun versucht sein könnte, weitere Zwischenschritte einzufügen.



Wenn diese Zwischenversionen nicht formal aus der jeweiligen vorherigen Version entstanden sind, dann spricht man von der *schrittweisen Verfeinerung*. Der Nachteil dieser Vorgehensweise liegt darin, daß für jedes Problem dieser Entscheidungsprozeß wieder neu entwickelt werden muß. Hier setzt das *transformationelle Programmieren* an:
Beim transformationellen Programmieren versucht man, formale Transformationsregeln für einzelne Übergänge anzugeben. Durch ein ganzes Regelwerk solcher Transformationen können die Zwischenversionen evaluiert werden. Damit ergeben sich folgende Vorteile (gegenüber „traditioneller Programmierung“):

- Korrektheit des Programms;
- Übergänge für ganze Problemklassen wiederverwendbar;
- Rechnerunterstützung möglich, da das Regelwerk formal ist.

Ausprägungen des transformationellen Programmierens

Beim transformationellen Programmieren können unterschiedliche Ziele verfolgt werden. Man unterscheidet daher folgende Ausprägungen des transformationellen Programmierens:

Programm-Modifikation beinhaltet zum Beispiel Optimierung, Effizienzsteigerung und Anpassung an Programmierstile.

Programm-Synthese meint die Erzeugung eines Programms aus einer Problembeschreibung.

Anpassung von Programmen an spezielle Umgebungen kann die Umformung von einer Sprache in eine andere bedeuten, aber auch die Anpassung von einem Ein-Prozessor-System an ein Mehr-Prozessor-System.

Verifikation als Nachweis der Korrektheit eines Programms gegenüber seiner Spezifikation.

Allen gemeinsam ist der evolutionäre Ansatz der Software-Entwicklung.

Transformationsregeln

Es gibt einige allgemeine Transformationsregeln, die durch sprachabhängige Regeln ergänzt werden.

- Expandieren (*unfold*): Ersetzen eines Funktions- oder Prozeduraufrufs durch den entsprechend modifizierten Rumpf;
- Komprimieren (*fold*): Ersetzung eines Ausdruck durch einen dazu gleichwertigen Funktions- oder Prozeduraufruf;
- Einführen und Löschen von Hilfsvariablen;
- Anwendung von Gesetzen der zu Grunde liegenden Datentypen (z.B. Auswerten von Klammerausdrücken beim Typ Integer).

Zwei Beispiele zu den sprachabhängigen Regeln:

- Distributivität von Funktionsaufrufen bzw. Operationen über die Fallunterscheidung:

  ```
  y + f(if B then x else x+1 endif) - 1
  ```

  ist gleichwertig mit

  ```
  if B
     then y + f(x) - 1
     else y + f(x+1) - 1
  endif
  ```

- Gleichwertigkeit von (endständiger) Rekursion und Iteration:

  ```
  function f(x: m) : n;
    if B(x) then f(K(x)) else T(x) endif
  ```

  ist gleichwertig mit

```
function f(x: m) : n;
  begin
    var y: m := x;
    while B(y) do y := K(y) enddo;
    T(y)
  end.
```

Das SETL-System

Das SETL-Projekt wurde Mitte der 70er Jahre am *Courent Institute of New York University* initiiert [HP83]. Es umfaßt zwei Bereiche des transformationellen Programmierens.

Der Optimizer

Der Optimizer verwendet einige transformationelle Techniken, zum Beispiel um Datenstrukturen auszuwählen oder effiziente Datentyp-Repräsentation nachzuweisen.

Durch Hinzufügen von Typdeklarationen zu einem Programm kann der Benutzer aus einer Menge von schon fertiger (korrekter) Implementationen für Tupel, Mengen und Abbildungen wählen. Wenn Deklarationen fehlen, werden konkrete Repräsentationen durch den Optimizer automatisch ausgewählt.

Semiautomatische Entwicklung

Der zweite Bereich, den das SETL-Projekt abdeckt, ist ein experimentelles interaktives System zur Unterstützung von halbautomatischer Entwicklung von wiederverwendbarer, effizienter Software. Das System beruht dabei auf schrittweiser Verfeinerung mittels korrektheitsbewahrenden Transformationen, die auf einer allgemeinen Form strenger Reduktion basieren. In das System einbezogen sind Kontroll- und Datenflußanalyse, geplant waren außerdem Typ- und Störungsanalyse (Stand: 1981).

Das CIP-Projekt

Das CIP-Projekt (Computer-aided, Intuition-guided Programming) wurde an der TU München entwickelt [HP87]. Nach einem in PASCAL geschriebenen Prototypen entstand das System CIP-S. Dieses besteht aus den beiden Teilen *Benutzungsoberfläche* und *Systemkern*.

Der Systemkern

Der Systemkern von CIP-S besteht aus zwei wesentlichen Teilen, dem *Transformator* und der *Wissensbasis*.

Transformator

Der Transformator stellt die Aufgaben

- Behandlung von Programmschemata und ihre Manipulation durch Transformationsausdrücke,
- Vereinfachung von Anwendbarkeitsbedingungen und
- Verwaltung verschiedener Datenbanken

zur Verfügung. Er ist die Schnittstelle zwischen den systemspezifischen Operationen und dem Benutzer.

CIP-S ist *sprachunabhängig* konzipiert, d.h. es kann mit jeder Sprache gearbeitet werden, die algebraisch spezifiziert wurde und für die entsprechende Werkzeuge vorliegen, die die Übersetzung zwischen interner und externer Repräsentation übernehmen.

Der Transformator gibt Unterstützung bei der Verifikation und Reduktion von Anwendbarkeitsbedingungen und die automatische Dokumentation von Entwicklungen.

Wissensbasis

Zur Wissensbasis gehören mehrere Datenbanken, in denen

- Transformationsregeln,
- abstrakte Datentypen, dargestellt durch den Gesetzen entsprechende Transformationsregeln,
- Programmschemata
- und Programmentwicklungen (in Baumform)

verwaltet werden.

Zu den verwalteten Datenbanken zählt insbesondere die *Entwicklungsgeschichte*, eine Datenbank zur Verwaltung verschiedener Programm(zwischen)-Versionen und ihren Beziehungen. Sie dient jedoch nicht nur der Möglichkeit, die zeitliche Entwicklung nachzuvollziehen, sondern soll auch Grundoperationen zur Manipulation der Entwicklung zur Verfügung stellen. Dazu gehören unter anderem auch das Komprimieren einer Entwicklung.

Weitere Entwicklung

Zur Zeit (1987) wird daran gearbeitet, aus dem Systemkern ein leistungsfähiges Software-Werkzeug zu entwickeln, das eine spezielle Sprache unterstützt. Dazu gehört nicht nur die Einbettung in eine Benutzungsoberfläche, sondern auch das Hinzufügen sprachspezifischer Komponenten und das Füllen der Wissensbasis mit abstrakten Datentypen und formal korrekt bewiesenen Transformationsregeln.

Interessant scheint auch, daß das System selbst auch mit Hilfe transformationeller Programmierung erstellt wurde und diese Erfahrungen gezeigt haben, daß auch große Projekte mit dieser Technik entwickelt werden können. Zusätzlich stellte sich sogar heraus, daß der Entwicklungsaufwand vergleichbar mit *konventioneller Entwicklung* ist, mit dem Vorteil, daß das System korrekt ist.

Bewertung der transformationellen Programmierung

Einige Vorteile wurden schon zu Beginn dieses Abschnitts aufgezählt und zeigten sich ja auch bei der Entwicklung des CIP-Systems. Zu diesen Vorteilen zählt

- Korrektheit des Programms;
- Übergänge für ganze Problemklassen wiederverwendbar;
- Rechnerunterstützung möglich, da das Regelwerk formal ist.

Die Korrektheit trägt in großem Maße zur *Produktqualität* bei, denn der Zwang zur formalen Spezifikation schafft Genauigkeit über Anforderungen usw. Außerdem können Fehlentwicklungen früher erkannt und damit besser vermieden werden.

Daneben führt das rechnergestützte transformationelle Programmieren auch zu einer großen *Wartungsfreundlichkeit*, da nicht nur das Programm vorliegt, sondern auch die formale Spezifikation des Problems und der vollständige Weg von der formalen Spezifikation zum Programm in der Entstehungsgeschichte nachvollzogen werden kann.

Werden später Mängel am Programm festgestellt, so kann

1. bei zu verändernder Spezifikation der Entwicklungsprozeß wiederholt werden (mit eventuell vorzunehmenden Variationen),
2. bei Portierung auf andere Systeme die Entwicklung mit dem neuen Zielsystem vorgenommen werden.

Weiter wird der *Herstellungsprozeß* inhaltlich nachvollziehbar und überprüfbar.

Neben der *Wiederverwendbarkeit* von Transformationsregeln wird auch die klassische Wiederverwenbarkeit im Sinne von Bibliotheken besser unterstützt, da diese leichter an geänderte Umgebungen angepaßt werden können.

15.2 Software-Entwicklungsumgebungen

15.2.1 Programmierumgebungen (Oliver Alsbach)

Einleitung

Zur effizienten Erstellung von Software werden Werkzeuge entwickelt, die alle Phasen der Erstellung unterstützen sollen. Die Möglichkeiten und Werkzeuge zur rechnerunterstützten Softwareerstellung sollen hier am Beispiel der SMALLTALK-Programmierumgebung

SMALLTALK/Objectworks gezeigt werden. Am Ende des Kapitels wird der Unterschied von Programmierumgebungen zu Softwareentwicklungsumgebungen erläutert.

SMALLTALK

SMALLTALK-80 ist eine objektorientierte Programmiersprache. Um zwischen der eigentlichen Programmiersprache und der Programmierumgebung zu unterscheiden, wird letztere im weiteren immer mit SMALLTALK/ Objectworks bezeichnet, die Sprache selbst mit SMALLTALK. Dieser Text bezieht sich auf die Version SMALLTALK/ Objectworks, Release 4, von Parc Place Systems. Die Programmierumgebung ist in SMALLTALK selbst geschrieben. Dadurch stellt sich die Umgebung als eine Menge von Objekten dar, wie Zahlen, Graphiken, Editoren, Compiler, usw. Alle Objekte sind für den Benutzer transparent, das heißt er kann die Eigenschaften des jeweiligen Objektes verändern, wodurch er -zumindest theoretisch- die Programmierumgebung selbst modifizieren kann.

Softwareentwicklung unter SMALLTALK/ Objectworks

Für das Entwickeln von Software existieren eine Reihe von Modellen, wie zum Beispiel das Wasserfallmodell, das Spiralmodell, etc. Um die Werkzeuge einzuordnen, die SMALLTALK/ Objectworks innerhalb der Entwicklung zur Verfügung stellt, wird diese hier in drei Phasen unterteilt:

1. `Analyse/ Entwurf`
2. `Implementierung`
3. `Systemintegration/ Test`

Analyse/ Entwurf bedeutet in SMALLTALK, die für das System notwendigen Objekte zu identifizieren, sie bezüglich ihrer statischen und dynamischen Eigenschaften zu klassifizieren und die Interaktion zwischen diesen Objekten zu entwerfen. In dieser Phase bietet SMALLTALK/ Objectworks keine speziellen Werkzeuge.

Implementierung heißt zuerst zu prüfen, wie die zu implementierenden Klassen in die bestehende Klassenhierarchie eingefügt werden sollen. Dabei können sowohl neue Klassen eingefügt werden, als auch — durch den Vererbungsmechanismus der objektorientierten Sprache — bereits vorhandene Klassen mitbenutzt werden. Diese Phase wird hauptsächlich durch das Werkzeug *System Browser* unterstützt.

Systemintegration/ Test bedeutet, Objekte der geänderten und neu hinzugefügten Klassen zu erzeugen und diesen Objekten Botschaften zu senden. Im Gegensatz zu konventionellen Systemen gibt es in SMALLTALK/ Objectworks keinen speziellen Vorgang, der einzelne Module zu einem Gesamtsystem zusammenfügt. Erzeugte Programme im konventionellen Sinn sind selbst Objekte und Bestandteile des Gesamtsystems. Die Integrations- und Testphase wird hauptsächlich durch die Werkzeuge *Workspace*, *Notifier*, *Debugger* und *Inspector* unterstützt.

Der System Browser

Der *System Browser* ist ein Werkzeug, das es ermöglicht sich interaktiv über die im System implementierten Klassen zu informieren, sie zu ändern und weitere hinzuzufügen. Zur besseren Übersichtlichkeit werden sowohl Klassen als auch deren Methoden in Kategorien unterteilt, die in seperaten Fenstern dargestellt werden. In einem weitern Fenster wird die Definition der gerade ausgewählten Klasse oder Methode angezeigt und kann editiert werden. Um das Hinzufügen neuer Klassen und Methoden zu vereinfachen, stellt der *System Browser* Muster zur Verfügung. Nach dem Einfügen bzw. Editieren einer Klasse oder deren Methoden kann diese Änderung compiliert werden und steht sofort im gesamten System zur Verfügung. Sind Klassen geändert worden, von denen im System noch aktuelle Objekte vorhanden sind, so erhalten diese existierenden Objekte nach Möglichkeit auch die geänderten Eigenschaften.

Der Finder

Der *Finder* ist ein Werkzeug, mit dem man auf einfache Art graphische Benutzerschnittstellen erzeugen und aufrufen kann. Interaktive Anwendungen bestehen in SMALLTALK aus den Teilen *Model*, *View* und *Controller*. Dabei bezeichnet *Model* die funktionierende Komponente, *View* die graphische Darstellung des Models und *Controller* die Steuerung der Benutzerinteraktion. Alle Komponenten sind wiederum als Objekte zu verstehen. Durch den Finder können nach dem *Baukastenprinzip* verschiedene graphische Darstellungen eines Objektes gewählt werden und die nötigen *Model*- und *Control*-Komponenten spezifiziert werden.

Workspace, Notifier, Inspector und Debugger

Um Objekte zu erzeugen und ihnen Nachrichten zu senden, benutzt man einen *Workspace*, eine Arbeitsumgebung, in die Ausdrücke geschrieben werden, die dann ausgeführt werden können. Dabei besteht ein Ausdruck aus einem Objekt und einer Nachricht, die an das Objekt gesand wird. Den aktuellen Zustand eines Objektes kann man sich mittels eines *Inspectors* ansehen. Treten bei der Ausführung von Ausdrücken Fehler auf, so wird ein *Notifier erzeugt. Dieser gibt an, welche Nachricht an welches Objekt den Fehler ausgelöst hat. Reicht diese Meldung noch nicht aus, um den Fehler sofort mittels eines* System Browsers *zu korrigieren, kann man zusätzlich noch einen* Debugger *aufrufen. Dieser zeigt sowhl die Fehlermeldung als auch die entsprechende fehlerhaft Methode an. Zusätzlich werden in zwei* Inspector-*Fenstern das entsprechende Objekt und die Parameter angezeigt. Mit dem* Debugger kann man den Ablauf eines Fehlerhaften Ausdrucks verfolgen und die Ergebnisse einzelner Nachrichten abfragen. Eine weitere Testmöglichkeit besteht darin, an gewünschten Stellen in die zu testende Methode direkt Aufrufe für einen *Notifier* oder *Debugger* einzufüegen, die dann die Ausführung an der entsprechenden Stelle unterbrechen.

Versionsverwaltung und Teamwork

Die Erstellung von großen Softwareprojekten ist es hilfreich, wenn die Aufgaben innerhalb von Gruppen verteilt werden können und unterschiedliche Versionen der erstellten Software verwaltet werden können. SMALLTALK/Objectworks selbst stellt hierfür keine speziellen Werkzeuge zur Verfügung. Alle am System vorgenommen Änderungen können zwar in einer *Change*-Datei eingesehen und schrittweise rückgängig werden, es gibt aber keine explizite Versionsverwaltung wie zum Beispiel das [[RCS]] unter [[UNIX]]. Die Möglichkeit zur Teamwork bietet sich nur sehr rudimentär dadurch, daß erstellte Klassen in Dateien abgespeichert bzw. aus Dateien ausgelesen werden können und somit auch anderen zur Verfügung stehen.

Abgrenzung gegenüber Softwareentwicklungsumbungen

Eine *Programmierumgebung* wie SMALLTALK/Objectworks unterstützt generell nur das Programmieren und Testen von Software. Die enthaltenen Werkzeuge sind oft sehr mächtig, aber auch sprachspezifisch, wodurch man auf eine Sprache festgelegt ist. Die Analyse und das Design des zu erstellenden Programms wird nicht unterstützt. In einer *Softwareentwicklungsumgebung* hingegen können auch diese Aspekte rechnerunterstützt werden. Als Beispiel sei hier kurz INCOME/ STAR vorgestellt. Es handelt sich hierbei um ein Projekt der UNI Karlsruhe, das auf dem komerziellen Produkt INCOME der Firma Oracle aufbaut. INCOME/ STAR unterstützt zum Beispiel folgende Aspekte:

- Systemanalyse durch High-Level-Petri-Netze
- Kontrollierter Nachrichtenfluß durch eingebundenes Mailsystem
- Konsistenz und Verwaltung von Dokumenten durch ein Data-Dictionary
- Arbeitsaufteilung und Teamwork innerhalb und zwischen einzelnen Gruppen
- Modellierung des Arbeitsflusses

15.2.2 Sprachunabhängige Entwicklungsumgebungen (Peter Neumann)

OPUS ist eine Entwicklung der STZ Gesellschaft für Software–Technologie mbH und des Lehrstuhls X der Universität Dortmund. Diese Entwicklungsumgebung soll Grundlage für den modularen Aufbau eines Software–Systems sein. Sprachunabhängig soll hierbei bedeuten, daß man theoretisch jede beliebige Programmiersprache in *OPUS* verwenden kann.

Grundlage für diesen Abschnitt ist [RF93].

Bestandteile von *OPUS*

Architektureditor: *OPUS* besteht aus einem Architektureditor, mit dem man Module anzeigen und hinzufügen kann. Außerdem werden im weiteren Verlauf der Programmiertätigkeit die Benutztbeziehungen zwischen den Modulen graphisch angezeigt, wenn über den Interfaceeditor solche Beziehungen eingegeben werden.

Interfaceeditor: Der Interfaceeditor dient zur textuellen Erfassung der Schnittstellen von Modulen. Hier werden die ex– und importierten Typen und Operationen, sowie Kommentare ediert.

Body–Editor: Im Body–Editor werden die internen Strukturen der jeweiligen Module abgelegt. Es können auch weitere nach außen nicht sichtbare Operationen definiert werden.

Außerdem ist es vorgesehen, Vorüberlegungen zur Realisierung von Typen und Operationen in Form von Pseudocode eingeben zu können, was aber z.Zt. noch als Kommentar betrachtet wird.

Code–Rahmen–Generator: Dieser Generator setzt die Schnittstellendefinition in eine Programmiersprache um, wobei man zwischen Modula–2 und C auswählen kann.

„C"–Syntaxeditor: Mit diesem Editor kann man die spezifizierten Module direkt Kodieren und alternativ zur Code–Rahmen–Generierung die Konsistenz zwischen Code und Entwurf sicherstellen.

Konzepte

OPUS unterstützt im Wesentlichen zwei Konzepte.

- Das Modularisierungskonzept
- Die Teilsystembildung

Das Modulkonzept

Um eine bessere Wartungsfreundlichkeit oder Wiederverwendbarkeit zu erreichen, müssen die Module eine vollständige Behandlung genau festgelegter Aufgaben verkapseln, sowie eine klare und eindeutige Schnittstelle besitzen.

Module in *OPUS*

In *OPUS* gibt es vier verschiedene Arten von Modulen:

ADT-Modul: In dieser Art von Modul wird eine Datenstruktur mit ihren zugehörigen Funktionen verkapselt, d.h. man kann einen oder mehrere Datentypen, sowie die dazu gehörigen Operationen exportieren.

ADO-Modul: Hier wird ein konkretes Objekt verkapselt, d.h. es können nur die zugehörigen Operationen exportiert werden und nicht der Datentyp selbst.

Funktionsmodul: Hier werden Operationen abgelegt, die zwar logisch zusammengehören aber nicht einem bestimmten Datentyp zugeordnet werden können, wie z.B. Steuerungsaufgaben, Transformationen oder Auswertungen.

Typkollektionsmodul: So ein Modul exportiert nur einfache Datentypen, wie INTEGER, BOOLEAN oder CHAR, auf die ohne spezielle Operationen zugegriffen werden kann.

Teilsysteme

Abstrahiert man von den Modulen auf „Black Boxes" erhält man daraus die im Architektureditor graphisch dargestellte Systemarchitektur.

Sollte einem diese Struktur dann immer noch zu undurchsichtig sein, kann man eine Art Supermodul, ein Teilsystem, definieren.

So kann eine hierarchische Struktur aufgebaut werden, um das System übersichtlicher zu machen, sowie die Mehrbenutzerfähigkeit zu unterstützen. [1]

Benutzung

Dem Benutzer präsentiert sich *OPUS* unter einer gemeinsamen fensterorientierten Oberfläche (X–Window, OSF–Motif). Jeder Editor hat sein eigenes Fenster, die alle gleichzeitig geöffnet sein können. Durch ein „Zoom–In" kann man sich die Struktur eines Teilsystems auf dem Architektureditor anzeigen lassen.

Die inkrementabhängigen, kontextsensitiven Pop–Up–Menüs lassen keine Verletzung des Aufbaus der Dokumente, sowie der Syntax des Entwurfs zu, da sie nur zulässige und syntaktisch richtige Operationen zur Modifikation erlauben.

Modi-freie Arbeitsweise

Zu jedem Zeitpunkt des Entwurfs hat man eine korrekte und konsistente Darstellung, egal in welchem Dokument man sich gerade befindet. Wenn man z.B. in einer Export–Schnittstelle eine Operation hinzufügt, kann man in dem dazugehörigen Body–Editor die Änderung ebenfalls beobachten.

Die Analysefunktionen dienen zum Auffinden von Funktionen importierender Module oder der Auflistung von Stellen an denen eine Operation gemacht wird.

[1] Näheres zur Mehrbenutzerfähigkeit von *OPUS*, siehe unter: **Verteiltes Arbeiten mit** *OPUS*

Verteiltes Arbeiten mit *OPUS*

Wie bereits angesprochen wird in *OPUS* die Mehrbenutzerfähigkeit oder die nebenläufige Entwicklung durch Teilsysteme unterstützt.

Dazu sollte die Schnittstelle eines Teilsystems

- explizit
- schmal und
- fix

sein.

Explizit heißt, daß jeder Benutzer weiß, welche Operationen er nicht ausführen darf oder kann, bzw. andersherum, welche Aktionen einen anderen Benutzer betreffen.

Schmal sollen die Schnittstellen sein, um die Auswirkungen von zugelassenen Änderungen so weit wie möglich einschränken zu können.

Fix bedeutet hierbei, daß die Schnittstellen so wenig wie möglich geändert werden, um Inkonsistenzen und Störungen paralleler Arbeiten zu vermeiden.

Durch ein *„Check Out“* werden die Teilsysteme aus dem Gesamtsystem herausgetrennt. So können die verschiedenen Benutzer das Teilsystem nebenläufig und beliebig verändern, ohne daß das Auswirkungen auf das Gesamtsystem hat.

Unter gewissen Einschränkung, siehe [RF93] S.14-17, ist es möglich ein Softwaresystem auf relativ hoher Ebene *„top down“* zu entwickeln, da die Teilsysteme voneinander losgelöst entwickelt werden können.
Beim anschließenden *„Check In“* kann dann der Grobentwurf eventuell angepaßt werden.

15.2.3 Integrierte Software-Entwicklungsumgebungen (Horst Sdun)

Einleitung

Grundlage dieses Vortrags war der Artikel [Cag].
Definition einer Enwicklungsumgebung:

Eine Softwareentwicklungsumgebung ist im Prinzip eine Zusammenstellung von Werkzeugen, die Zusammenarbeiten um den Softwareentwicklungsprozess zu unterstützen.

Ziele, Erwartungen und Probleme

Zielsetzung

Die Umgebung soll zum einen eine einheitliche, leicht erlernbare Benutzungsoberfläche bieten, und zum anderen die Arbeitsabläufe bei der Softwareentwicklung unterstützen und vereinfachen.

- Einheitliche, leicht erlernbare Benutzungsoberfläche
- Unterstützung und Vereinfachung von Arbeitsabläufen

Erwartungen

Die Erwartungen, die an die Benutzung einer solchen Umgebung geknüpft werden sind:

- Steigerung der Produktivität
- Steigerung der Qualität
- Langfristige Kostensenkung

  Investition,Schulungen,Einarbeitung erhöhen zunächst die Kosten bei dem Einsatz einer Eintwicklungsumgebung.

Das eigentliche Problem ergibt sich nun aus dem Wunsch, die verschiedenen Werkzeuge des Softwareentwicklungsprozesses miteinander zu verknüpfen.
Typischerweise werden Werkzeuge von verschiedenen Herstellern angeboten bzw. verwendet, wobei sich diese Werkzeuge in der Regel auf ganz unterschiedliche Aspekte der Softwareentwicklung konzentrieren.
Diese verschiedenen Werkzeuge besitzen in der Regel jedoch keine einheitlichen Schnittstellen (GUI, Daten etc).
Momentan existiert auch kein wirklicher Standard, der von mehreren Herstellen eingehalten/akzeptiert wird.
Das Ziel der Werkzeugintegration ist es nun, eine vollständige Entwicklungsumgebung unter/durch Einbindung dieser Werkzeuge zu kreieren, die den gesamten Software- lifecycle abdeckt.

> **Wunsch:**
>
> Integration und Verknüpfung aller bei der Softareentwicklung beteiligten Werkzeuge in eine Entwicklungsumgebung.
>
> daraus ergeben sich jetzt unmittelbar die
>
> **Probleme:**
>
> - Verschiedene, vernetzte Betriebssysteme innerhalb der Entwicklungsumgebung (Stichwort: Virtual Operating Environment)

- Die Vielfalt der Werkzeugschnittstellen
  Datenrepräsentation (Stichwort: Datenintegration)
- Die unterschiedlichen Benutzungsschnittstellen
  Stichwort: Präsentationsintegration

Die HP SoftBench

Die vollständige Integration der verschieden Werkzeuge in eine Entwicklungsumgebung besteht aus der Integration 5 verschiedener Teilaspekte. Die folgenden Typen sich definieren:

Integrationstypen

1. **Plattformintegration**
   (Verschiedene Tools müssen mit/nebeneinander verwendet werden.) Der wichtigste Punkt für eine integrierte Lösung ist die Interoperabiliät der Einzelwerkzeuge. In der Vergangenheit bedeutete dies, daß die Software auf einem Rechner unter einem Betriebsystem lief. Heute ermöglicht eine verteilte Umgebung durch die Verwendung netzwerkfähiger Dateisysteme und netzwerkfähiger Interprozeßkommunikation den Austausch von Daten zwischen verschiedenen Systemen. (← Virtual Operating Environment)

2. **Präsentationsintegration**
   Alle Tools sollten ein gemeinsames look & feel aufweisen. Dies vereinfacht nicht nur die Erlernbarkeit eines neuen Tools, sondern erhöht auch die Akzeptanz. Dies wurde bereits durch den Apple Macintosch vorgeführt.

3. **Datenintegration**
   Die Werkzeugintegration benötigt zwei Dinge:

   a) Die Tools müssen in der Lage sein, sich Daten zu teilen.
   b) Management der Relationen, die zwischen den durch die verschiedenen Tools erzeugten Daten. bestehen.

   Ursprünglich wurden dazu Dateien und Interprozeßkommunikation benutzt. In der letzten Dekade hat sich der Gedanke gefestigt, daß dies durch ein(e) Repository (Datenbank) erfolgen sollte. Beispiele: PCTE (Portable Common Tool Environment) mit OMS (Object Managemet System), oder IBM/AD Cycle.

4. **Kontrollintegration**
   Die Werkzeuge müssen einerseits in der Lage sein, Ereignisse zu erzeugen und andererseits natürlich auch auf solche zu reagieren. Beispiel: Idealfall wäre, das durch die Änderung eines Designs auch die dazugehörige Dokumentation automatisch auf diesen neuen Stand gebracht wird. Dies ermöglicht z.B. die HP-Softbench durch

den BMS (Broadcast Message Server), bei dem Tools ein Event absetzen oder auf gewisse Nachrichtenklassen warten können.

5. **Prozeßintegration**
   Eine Entwicklungsumgebung sollte nicht eine Menge von Werkzeugen, die miteinander kommunizieren, zur Verfügung stellen, sondern auch den Entwicklungsprozeß selbst unterstützen. Dies kann durch die Bereitstellung geeigneter Werkzeuge geschehen, welche eine Steuerung des (Ablaufs der) Entwicklungsprozesses zulassen.

Anforderungen HP formulierte für die SoftBench die folgenden 7 Anforderungen:

Anforderungen

1. **Unterstützung verschiedener, austauschbarer Entwicklungswerkzeuge.** HP wollte keine universelle Lösung für alle Entwickler, Aufgaben, und Anforderungen an die Werkzeuge. Sondern die flexible Berücksichtigung individueller Vorlieben und arbeitsspezifische Notwendigkeiten (Schwerpunktverteilung, Fähigkeiten der Werkzeuge).

2. **Unterstützung von Softwareentwicklung in einer verteilten Computerumgebung.**

3. **Einbindung bereits vorher vorhandener und auch selbstentwickelter Werkzeuge in die Umgebung.** Notwendigerweise muß die Einbindung ohne Eingriffe in die Quellen möglich sein. (Was bei der Einbindung kommerzieller Software ja auch gar nicht möglich wäre.)

4. **Unterstützung von Entwicklerteams in einer verteilten Entwicklungsumgebung.** Also nicht nur die Unterstützung der "reinen" Softwareentwicklung sondern auch von Team-Koordination und Projekt-Managment.

5. **Unterstützung von verschiedenen Vorgehens- und Arbeitsweisen.** Die Umgebung soll die Arbeitsabläufe auf gar keinen Fall diktieren, sondern gleichzeitig verschiedene Abläufe wie z.B. Wartung, Rapid Prototying auf verschiedene Arten unterstützen.

6. **Einbindbarkeit von beliebigen anderen Life-Cycle Werkzeugen.** Analyse, Designwerkzeuge (Dokumentations Retrieval).

7. **Die Umgebung soll auf bestehende Standards aufsetzen.** HP sieht in diesen "bestehenden Standards" das Betriebssystem Unix (System V) mit NFS (Network File System), ARPA (Advanced Research Projects Agency) und das MIT X Window System mit dem Erscheinungsbild und Verhalten von OSF/Motif. (gemeint ist hiermit das look&feel des OSF/Motif GUI Style Guides).

Integration Die HP-SoftBench ist in erster Linie eine **Integrationssoftware**, die durch ihre Architektur einen Mechanismus für die Kooperation verschiedener Werkzeuge in einer verteilten Umgebung zur Verfügung stellt. Dieser Integration stützt sich hauptsächlich auf die **Kontroll- und Prozessintegration** (vgl. 5 und 4).
Im Gegensatz dazu stehen Integrationsarchitekturen, die sich auf die Organisation von Zugriffen auf Daten oder Management von Relationen zwischen Daten (Datenintegration vgl. 3) konzentrieren.

Die HP SoftBench unterstützt für die Werkzeugintegration drei Mechanismen.

HP SoftBench Werkzeugintegration

- **Werkzeugkommunikation** Teil der Daten/Kontrollintegration.
- **Unterstützung im Netzwerk** Teil der Plattformintegration.
- **Benutzungsoberflächen Management** Entspricht der Präsentationsintegration.

Kommunikation Die (Werkzeug-) Kommunikation wird durch den HP eigenen BMS (Broadcast Message Server) erbracht (*etwa wie durch einen Kommunikations-Satelliten*). Jedes Werkzug meldet sich zunächst bei Programmstart bei dem BMS an.

BMS-Services

- **(Event-) Notification**
  teilt diesem mit, welche Nachrichtentypen vom BMS an dieses Werzeug durchgereicht werden. Es handelt sich also nicht um einen echten Broadcast ("Multicast Message Server" träfe besser). Dies verringert die Netzlast, und vereinfacht Werkzeuge, die somit nicht alle Nachrichtentypen erkennen müssen. Was zweifelsohne auch eine unproblematische Erweiterung des Protokolls ohne Eingriffe in bestehende Werkzeuge ermöglicht. Erhält der BMS nun eine Nachricht von einem Werkzeug, so sendet er sie an alle anderen Werkzeuge, die ihr Interesse an diesem Nachrichtentyp bekundet haben. Somit kann z.B. ein Filemanager zu einem Refresh veranlaßt werden (sozusagen ein Trigger).
- **(Service-) Request**
  weiterhin kann jedes Tool auch einen Request an den BMS stellen. Läuft das benötigte Werkzeug bereits, so wird der Request entsprechend weitergeleitet, andernfalls wird das vom Benutzer für diesen Zweck — der Benutzer kann individuelle Werkzeuge für Requests definieren — definierte Werkzeug gestartet und der Request dann weitergeleitet. Die Ausführung der Werkzeuge wird von dem *Execution Manager* durchgeführt und überwacht.

Verteilte Umgebung Die Ausführung von Werkzeugen kann auch remote auf anderen Rechnern im Netzwerk erfolgen.

HP SoftBench im Netzwerk

- Remote Execution via HP SPC (HP-SoftBench): Subprocess Control
  Ist ein Werkzeug für einen bestimmten Host konfiguriert, so wird dieses Werkzeug auch immer dort automatisch gestartet. Alle nötigen Einstellungen (z.B. Verwaltung des Displays) geschehen automatisch.
  - Unix NFS
    Die Pfade für Datenzugriffe können entsprechend konfiguriert werden.
  - X Windows
    Display Management

Präsentation Für die Benutzungsoberfläche der HP SoftBench wird OSF/Motif unter X Windows benutzt. Da OSF/Motif über einen GUI-Styleguide verfügt, wird einheitliches Aussehen und Bedienung der Applikationen garantiert.

Die HP SoftBench bietet ein Werkzeug an, den HP Encapsulator, mit dem beliebige andere (ASCII-UI) Werkzeuge in ein graphisches look&feel gekapselt werden können. Der Encapsulator ist einer Art Motif Widget Set, der u.a. auch ein Edit Widget für interaktive Benutzereingaben zur Verfügung stellt. (Z.B. für RCS/SCCS Kommentare etc.)

Desweiteren werden auch Fremdsprachen und deren Zeichensätze in allen Dokumenten und Quelltexten unterstützt. Benutzer die die Tastatur der Maus vorziehen, können für alle betreffenden Operationen Tastaturmakros definieren.

Die HP SoftBench Präsentationsintegration

- HP Encapsulator. Ein Tool zur Einkapselung von (fast) beliebigen anderen Werkzeugen. Der HP-Encapsulator ist im Prinzip eine X11-Widget Bibliothek, mit welcher die Werkzeuge dann auf C-Ebene eingekapselt werden können. HP liefert zur HP SoftBench u.a. eine Einkapselung von "mailx".

  Der "Encapsulator" unterstützt auch alle Möglichkeiten der HP SoftBench. D.h. Fremdwerkzeuge haben in der Umgebung dieselbe Wertigkeit wie die HP-Werkzeuge.
  - Events und Services
  - Distributed execution
  - Network-wide comunication
  - OSF/Motif look&feel

- **Fremdsprachenunterstützung**
  durch Unterstützung von 8/16 Bit Zeichensätze (Folie Japanese) in den zur WorkBench mitgelieferten Werkzeugen.

- **Benutzerdefinierbare Tastaturmakros**
  zur individuellen Konfiguration der zur WorkBench mitgelieferten Werkzeuge.

15.2.4 Prozesszentrierte Software-Entwicklungsumgebungen (Klaus Alfert)

Merlin: Eine prozessorientierte Entwicklungsumgebung

Merlin (siehe auch [JPSW93], [JPSW94], [PS92],[PW93] sowie [SW94]) ist eine regelbasierte prozess–zentrierte Software–Enwicklungsumgebung (PSDE), die die Kooperation zwischen mehreren Anwendern unterstützt. Merlin fußt auf dem Begriff des Software–Prozesses. Damit ist nicht ein Prozeß wie in Betriebssystemkontexten gemeint, sondern eine Beschreibung, in welcher Art und Weise Software entwickelt wird. Genauer müßte es also Prozeß der Software–Entwicklung heißen. Zur Beschreibung eines Software–Prozesses und der damit verbundenen Aktivitäten dient eine regelbasierte PROLOG-ähnliche Sprache. Man kann sich somit Merlin als eine Art Expertensystem ohne Erklärungskomponente für die Software–Enwicklung vorstellen.

Die Elemente eines Prozesses Ein Prozeß in Merlin besteht aus einer Ansammlung von Objekten aus vier verschiedenen Klassen. Durch die Definition der möglichen Instanzen dieser Klassen sowie ihre Abhängigkeiten untereinander, kann man den Prozeß vollständig beschreiben. Im folgenden sollen die Klassen erläutert werden.

Aktivitäten Aktivitäten sind Aufgaben, die erfüllt werden müssen, um ein bestimmtes Ziel im Rahmen der Entwicklung zu erreichen. Dies sind zumeist Teilaufgaben im Gesamtprozeß wie etwa Spezifizieren, Editieren, Compilieren etc.

Rollen Rollen fassen Aktivitäten zusammen, die logisch zu einander passen. Sie beschreiben den Aufgaben- bzw. Tätigkeitsbereich eines Entwicklers, wie z.B. Projektmanagement, Design, Qualitiätssicherung, Implementation.

Dokumente Mit Dokumenten werden alle Objekte gemeint, die im Laufe des Software–Entwicklungs–Prozesses entstehen. Das sind sowohl maschinell als auch manuell erzeugte Objekte wie Programme, Dokumentation, Testpläne, Module usw.

Resourcen Resourcen umfassen sowohl die Menschen, die an der Entwicklung beteiligt sind, als auch die Programme und Tools, die zur Entwicklung benötigt werden (z.B. Editoren, Debugger).

Zwischen den verschiedenen Klassen bestehen Zusammenhänge der Art, daß auf Dokumenten bestimmte Aktivitäten durchgeführt werden können, die wiederum spezifische Resourcen benötigen. Die Benutzer sind mit entsprechenden Rollen verknüpft, die den Tätigkeitsbereich definieren.

Dokumente besitzen die besondere Eigenschaft, daß sie mit einem Status versehen sind. Dieser Status kann durch Aktivitäten modifiziert werden. Gleichzeitig dient dieser Status dazu, zu bestimmen, welche Aktivitäten auf einem Dokument ausgeführt werden dürfen. Näheres dazu weiter unten (Seite 344).

Der Working Context Die Schnittstelle zum Benutzer wird Working Context genannt. Nach dem Anmelden des Benutzers und der damit verbundenen Festlegung der Rolle des Benutzers, werden sämtiche für die Rolle verfügbaren Dokumente als Icons und ihre Relationen untereinander (z.B. benutzt, importiert) angezeigt. Zu jedem Dokument kann ein individuelles kontextsensitives Menü mit den möglichen Aktivitäten des spezifischen Dokumentes angezeigt werden.

Wenn sich durch die Ausführung einer Aktivität der Status einiger Dokumente geändert hat, so wird dies dem Benutzer durch ein Update–Flag angezeigt. Nach dem Klicken auf das Update–Flag wird der Working Context von Merlin neu berechnet. Diese Neuberechnung kann zur Folge haben, daß neue Dokumente erscheinen, andere entfernt werden, und daß sich die möglichen Aktiviäten auf einem Dokument verändert haben. Diese Veränderungen des Working–Contexts haben Auswirkungen auf alle Benutzer, die an dem selbem Projekt arbeiten.

Ein Beispiel–Szenario Die Arbeitsweise soll an einem einfachem Beispiel gezeigt werden. Gegeben seien drei Dokumentenklassen: Spezifikation, Programm–Modul sowie Programm. Das Programm soll aus zwei Modulen bestehen, die wiederum in 2 Spezifikationen definiert werden. Es bestehen die folgenden Relationen zwischen den Dokumenten:

m1-spec	uses	m2-spec
m1-spec	is-implemented-in	m1-c
m2-spec	is-implemented-in	m2-c
m1-c	imports	m2-c
m1-c	is-executable-in	m1-prg

Es soll nun der Working Context eines Benutzers mit der Rolle *Programmierer* betrachtet werden. Ein Programmierer darf die Spezifiktion nicht ändern, also darf er die beiden Spezifikationen nur lesen und ausdrucken. Anders dagegen die beiden Programm–Module. Diese dürfen zusätzlich editiert werden. Das Programm darf nur ausgeführt werden. Dieses ist aber nur dann möglich, wenn die beiden Programm–Module den Status *compiled* oder höher besitzen, da sie erst dann dem Programm als Bestandteile zur Verfügung stehen. Nach dem Editieren eines Moduls legt der Benutzer fest, ob das Modul den Status *implemented* erreicht hat. Falls dies der Fall ist, kann Merlin selbständig die

Compilierung einleiten. Falls dies erfolgreich war, wird dem Modul der Status *compiled* vergeben, anderfalls wird es wieder zurückgesetzt auf den Stand *not-implemented*. Um das Modul m1-c übersetzen zu können, muß aber zuerst das Module m2-c im Status *compiled* vorliegen, da sonst der Import nicht funktionieren würde.

Bis hierhin stellt sich Merlin als eine interaktive Erweiterung von Makefiles dar. Deshalb soll an dieser Stelle ein Beispiel der Kooperation zwischen mehreren Benutzern gezeigt werden. Dafür wird eine neue Rolle — die Qualitätssicherung — eingeführt. Diese benutzt zusätzlich den Dokumenttyp Testplan. Die neuen Relationen sehen wie folgt aus:

m1-spec	implementation-is-reviewed in	m1-testplan
m2-spec	implementation-is-reviewed in	m2-testplan
m1-c	is-reviewed-in	m1-testplan
m2-c	is-reviewed-in	m2-testplan

Der Testplan darf von der Qualitätssicherung natürlich editiert werden, aber nicht die Programm–Module. Wenn der Programmierer sein Modul mit dem Status *tested* versieht, geschieht ein Update auf dem Working Context des Qualitätssicherers, so daß der Testplan, das Modul, die Spezifikation und das Testprogramm zusätzlich angezeigt werden, da der Qualitätssicherer nun mit dem Test des Moduls beginnen kann. Umgekehrt verschwindet auf dem Working Context des Programmierers das Modul, sofern von ihm keine weiteren Aktivitäten mehr abhängen (wie etwa Importe). Wenn das Modul den Testaktivitäten nicht standhält, wird sein Zustand von *tested* wieder auf *not-implemented* zurückgesetzt und es erscheint erneut beim Programmierer und verschwindet beim Qualitätssicherer.

Definition eines Software–Prozesses in Merlin

Die Definition von Prozessen innerhalb von Merlin geschieht mittels Regeln, die in Prolog formuliert sind. Teile davon können auch mit einem graphischen Editor für erweiterte ER-Modelle und Finite State Machines realisiert werden [JPSW94]. Die Definitionen werden in drei unterschiedlichen Bereichen vorgenommen. Diese beziehen sich auf ein Projekt, einen Prozeß und den Kernel von Merlin.

Der Projektbereich Der Projektbereich umfaßt eine Menge von Regeln, die Fakten über das spezifische Projekt beinhalten. Da es sich nur Fakten handelt, muß an dieser Stelle keine Regel–Programmierung vorgenommen werden. Die Fakten beinhalten Daten, die insbesondere Benutzer mit Rollen und Verantwortlichkeiten sowie Dokumenten verknüpfen. Dies wird exemplarisch in Abbildung 15.1 gezeigt.

In dem Projekt *Scotty* gibt es drei Benutzer mit verschiedenen Rollen. Jedem Paar bestehend aus Rolle und Benutzer kann nun eine Reihe von Dokumenten zugeordnet werden, für die der Benutzer in der spezifischen Rolle zuständig ist. So ist der Benutzer *claus* in der Rolle des *compiler_writer* für das Dokument *pstc.eli* verantwortlich.

```
project(Scotty, [willi, claus, eed]).

has_roles(willi, [linda_guru, player]).
has_roles(claus, [compiler_writer, speaker]).
has_roles(eed, [chair]).

responsibilities(willi, linda_guru, [liblwp.a, liblinda.a]).
responsibilities(claus, compiler_writer, [pstc.eli]).
```

Abbildung 15.1: Definition eines Projektes

Der Prozeßbereich Bis jetzt ist nur angegeben, wer für was verantwortlich ist. Unbekannt ist noch, was man mit dem Dokumenten machen kann. Dies ist unabhängig von einem spezifischen Projekt, deshalb ist es auf der Prozeß–Ebene angeordnet. Es betrifft u.a. die Festlegung, was für Dokumententypen es gibt, welche Zustände ein Dokument besitzen kann, welche Beziehungen zwischen Dokumenten existieren, welche Tools auf Dokumente in einem spezifischen Zustand arbeiten dürfen, usw.

Ebenfalls auf dieser Ebene ist die Definition der *consistency conditions* und der *automation conditions* untergebracht, die man am ehesten mit den Regeln eines Makefiles vergleichen kann. Es wird angegeben, was automatisch von Merlin durchgeführt werden kann (bei automation conditions) oder durchgeführt werden muß (consistency conditions), um die Integrität des Working Contextes zu erhalten. Hierbei wird angeben, wie sich die Zustände des Dokumentes ändern, wenn weitere Nebenbedingungen erfüllt sind. An dieser Stelle kommen auch die Relationen zu anderen Dokumenten zum tragen, da die Definition des neuen Zustandes auch von denen der anderen Dokumente abhängen kann (siehe Abbildung 15.2).

```
automation_condition(c_module, implemented, not_yet_compiled,
      [[destination, imports, completed]]).
```

Abbildung 15.2: Definition einer automation condition

Diese Bedingung sagt aus, daß ein *c_module* von dem Status *implemented* nach *not_*-

yet_compiled wechseln kann, wenn alle Ziel–Dokumente (*destination*) der *imports*–Relation den Zustand *completed* besitzen.

Ebenfalls wird in der Prozeß–Definition festgehalten, welche Rolle auf welche Dokumentart zugreifen darf, wenn sich das Dokument in einem bestimmten Zustand befindet. Dies ist die Situation aus dem Beispiel, in dem je nach Zustand des Dokumentes entweder der Programmierer oder der Qualitätssicherer das Programm bearbeiten kann, weil es auf seinem Working Context erscheint.

Letzlich sei noch angemerkt, daß auch in dieser Ebene noch keine Regeln in Prolog formuliert wurden, sondern lediglich Fakten beschrieben werden.

Der Kernel Im Kernel von Merlin befinden sich die Prolog–Regeln, die mit den Fakten aus der Projekt– und Prozeß–Ebene online die Aktionen steuern bzw. berechnen. Der Kernel teilt sich in fünf Regel–Cluster auf: StartUp, WorkingContext, ChangedStates, TransactionManager und LockManager. Die letzen beiden werden im nächsten Abschnitt näher betrachtet.

Die Aufgabe des Kernels ist es, den Working Context eines Benutzers zu bestimmen und ihn in einem korrekten Zustand zu bewahren. Dazu gehört die Bestimmung der Dokumente, auf die der Benutzer arbeiten darf, sowie der Tools, die dazu benötigt werden. Bei Status–Änderungen berechnet der Kernel die Folgen, z.B., ob dies Auswirkungen auf den Working Context des Benutzers oder eines anderen Benutzers hat, ob consistency oder automation conditions bearbeitet werden müssen und so fort.

Konzeptionell wichtig ist dabei, daß der Kernel unabhängig von einem Software–Prozeß ist. Die verschiedenen Prozesse, etwa Wasserfall, Inkrementell, Transformation u.ä., werden vollständig auf der Prozeß–Ebene definiert und bedürfen keiner Modifikation des Kernels. Dies ist nur notwendig, wenn eine Modifikation der Benutzerschnittstelle oder etwa Erweiterungen wie ein anderes Transaktionskonzept oder Versionsverwaltung vorgenommen werden.

Nebenläufigkeit und Transaktionskonzept

Wenn mehrere Entwickler an einem Projekt arbeiten, bleibt es meistens nicht aus, daß ein Dokument gleichzeitig von verschiedenen Benutzers bearbeitet wird. Da dies zu Read/Write– bzw. Write/Write–Konflikten führen kann, muß dies durch einen Transaktions–Manager gesteuert werden. In traditionellen Datenbanken gibt es ausschließlich ACID–Transaktionen (ACID: Atomar, Concurrent, Isolated execution and Durable), die die Konflikte entweder serialisieren (pessimistisches Scheduling) oder aber eine Reihe von Transaktionen wieder rückgangig machen (optimistisches Scheduling). Dies ist aber unbefriedigend bei SDE's, da es zum Beispiel ungünstig ist, wenn eine Editor–Sitzung vom System wieder zurück gesetzt wird. Desweiteren ist es i.a. nicht möglich, Transaktionen zu schachteln bzw. sehr lange Transaktionen (über Tage und Wochen) durchzuführen. Da dies aber reale Anforderungen sind, ist in Merlin ein Transaktions–Manager integriert, der ein eigenes Transaktionskonzept realisiert.

Eine Aktivität innerhalb von Merlin kann eine von insgesamt fünf verschiedenen Transaktionstypen für ihre Durchführung benutzen. Diese lassen sich in einer ersten Einteilung als *user transactions* und als *process engine transaction* klassifizieren.

user transactions sind als einzelne Aktivitäten, die vom Benutzer angestoßen werden, immer kurze Transaktionen. Sie können sowohl optimistisch als auch pessimistisch durchgeführt werden. Es ist möglich, bei kurzen Transaktionen das Scheduling während der Transaktion zu ändern.

Die Transaktionen, die alle Aktivitäten innerhalb des Working Contextes steuern, sind lange Transaktionen, die immer pessimistisch durchgeführt werden, weil der Aufwand für ein Zurücksetzen viel zu groß ist.

process engine transactions sind Transaktionen für Aktivitäten, die von Merlin selber im Rahmen der automation oder consistency conditions veranlaßt werden. Sie werden immer mit einem pessimistischen Scheduling versehen.

Welche Transaktionsarten ablaufen können, ist von der Rolle des Benutzers, der Aktivität, den Zugriffsrechten und nicht zuletzt vom Projektstatus abhängig. So kann es z.B. kurz vor einer Deadline nicht mehr möglich sein, ein Modul mit einer optimistischen Transaktion zu bearbeiten, um die Änderungen wegen eines Rollbacks nicht wieder zu verlieren. Da in die Bestimmung der Transaktionstypen Informationen über das Projekt selber eingehen, muß der Transaktions–Manager in den Merlin–Kernel eingebunden sein und es kann daher nicht der Transaktions–Manager der Datenbank benutzt werden.

Die Unterscheidung der verschiedenen Transaktionsarten ist notwendig, weil diese eine spezifische Priorität besitzen. Mittels dieser Priorität wird bei einem konkurrierendem Zugriff entschieden, welche Transaktion sich durchsetzen darf und welche ein Rollback vornehmen muß. So haben consistency transactions die höchste Priorität, pessimistische User–Transaktionen haben eine höhere Priotität als die automation transactions.

Konflikte beim konkurrierendem Zugriff werden durch den Lock–Manager festgestellt, der die einzige Instanz ist, die sämtliche Locks setzt und damit als einziger Informationen über alle gelockten Dokumente und Zustände hat.

15.2.5 Prototypingumgebungen (Marcus Kirsch)

Im ersten Teil dieser Ausarbeitung werden zunächst Entwicklungswerkzeuge in Prototypingumgebungen in Anlehnung an [BKKZ92] und [PBDKS91] klassifiziert und Anforderungen an diese formuliert. Im zweiten Teil werden dann zwei konkrete Umgebungen für die Erstellung von Prototypen vorgestellt.

Klassifizierung von Prototyping-Werkzeugen

Wie auch in anderen Softwaretechnik-Ansätzen, spielen beim Prototyping die dafür verfügbaren Entwicklungswerkzeuge eine große Rolle für die Praktikabilität und Effizienz dieses Ansatzes. Da gerade für das Prototyping nur wenige Werkzeuge existieren, ist es wichtig, Anforderungen an solche Werkzeuge möglichst klar darstellen zu können. Die folgende Klassifizierung versucht dem gerecht zu werden.

Generatoren Mit Generatoren können Prototypen auf hohem Abstraktionsniveau spezifiziert werden, die dann in lauffähige Systeme übersetzt werden können. Der Entwickler braucht nur anzugeben, *was* zu tun ist, die Realisierung wird vom Generator übernommen. Die Spezifikation kann auf verschiedene Arten geschehen, z.B. durch eine Beschreibungssprache, durch Grammatiken oder durch graphische Spezifikation. Generatoren finden häufig Verwendung bei der Erstellung von Prototypen für Benutzungsoberflächen in der Form von Anwendungsgerüsten, die die Erzeugung einer einheitlichen Benutzungsschnittstelle ermöglichen, die dann nur noch durch anwendungsspezifische Funktionen ergänzt werden muß.

Datenbankorientierte Entwicklungssysteme Werkzeuge zur Entwicklung von Prototypen für Informationssysteme müssen folgende Aspekte unterstützen:

1. Beschreibung eines Datenbankschemas
2. Erzeugung einer Datenbank
3. Testen einer Datenbank
4. Erstellen einer entsprechenden Benutzungsoberfläche

Diese Anforderungen können auf unterschiedliche Weise erfüllt werden:

1. grafikorientiert
   ein Datenbankschema wird grafisch erstellt und durch Verwendung eines Generators die Datenbank erzeugt
2. Datendefinitionssprachen
   hier wird das Schema durch eine Spezifikationssprache beschrieben
3. Systeme der 4. Generation
   sind integrierte Pakete, die Schema-Editoren, Benutzungsoberflächen- und Anwendungsgeneratoren enthalten

Architektursimulatoren Die Entwurfsphase liefert eine Systemarchitektur, in der alle Systemkomponenten und deren Beziehungen zueinander spezifiziert sind. Zur Überprüfung einer solchen Architektur dienen Architektursimulatoren, die folgende Funktionen bieten sollten:

1. Simulation des Informationsflusses zwischen den Systemkomponenten

2. Einbindung des Architekturentwurfs in den Prototyp einer Benutzungsoberfläche
3. Anzeige und Anwahl von aktuellen Systemzuständen Aufzeichnen von Zustandsänderungen während einer Simulation
4. Wiedergabe einer solchen Aufzeichnung
5. Unterstützung inkrementeller Implementierung

Programmiersprachen Zur Erstellung von Prototypen geeignete Sprachen zeichnen sich vor allem aus durch:

1. hohes Abstraktionsniveau, das z.B. eine Spezifizierung des Kontrollflusses in einem Programm überflüssig macht
2. Interpretierung statt Compilierung
3. Konzepte, die die Wiederverwendung und leichte Erweiterbarkeit von Komponenten ermöglichen
4. Einbettung in eine Entwicklungsumgebung

Programmierumgebungen Programmierumgebungen zum Prototyping sollten folgende Tätigkeiten unterstützen:

1. Entwurf
2. Bearbeitung von Quelltexten und Dokumentation
3. Compilieren, Einbinden, Laden und Testen von Programmen
4. Konfigurations- und Projektverwaltung

Generell sollten Programmierumgebungen das verwendete Modell des Softwareentwicklungsprozesses unterstützen, in diesem Kontext also speziell Prototyping.

Zwei Beispiele für Programmierumgebungen

Die im folgenden vorgestellten Systeme versuchen die im vorhergehenden Teil formulierten Anforderungen an die Werkzeuge einer Prototypingumgebung zu erfüllen.

Topos [PBDKS91] soll die Entwicklung von hybriden Systemen unterstützen, d.h. von Systemen, die aus High-Level-Teilen ebenso wie aus wiederverwendeten Komponenten und in Sprachen der 3. Generation implementierten Teilen zusammengesetzt sind. Die zur Verfügung gestellten Werkzeuge sollen vorrangig folgendes leisten:

1. Unterstützung der Anforderungsanalyse und der Spezifikationsphase durch Erzeugung von Benutzungsschnittstellen,

2. Überprüfung einer Systemarchitektur durch einen Prototypen, wobei dieser Prototyp zusammen mit der Benutzungsschnittstelle ausgeführt werden kann.

Für den ersten Punkt wird das Kontruktionswerkzeug für Benutzungsschnittstellen *UICT* (User Interface Construction Tool), für den zweiten Punkt ein Simulator zur Verfügung gestellt. Letzterer ist in ein Werkzeug zur Systemkonstruktion *SCT* (System Construction Tool) eingebettet.

UICT ist ein Generatorensystem und besteht aus den Werkzeugen *DICE*, *UI-Übersetzer*, *UI-Interpreter* und einem *M2/C-Programmgenerator*. UICT unterstützt die Erzeugung von graphischen Benutzungsschnittstellen, die Elemente wie Fenster, Popup-Menüs, Dialogboxen und Listen verwenden. Eine graphische Benutzungsschnittstelle kann sowohl mit dem graphischen WYSIWYG-Editor DICE (Dynamic Interface Creation Environment) als auch mit Hilfe der Spezifikationssprache UISL (User Interface Specification Language) erzeugt werden.

DICE erzeugt ein C++-Programm, das mit einem Compiler direkt in ein Objektprogramm übersetzt werden kann. In UISL formulierte Spezifikationen werden mit dem UI-Übersetzer in eine *UI-Tabelle* überführt. Diese UI-Tabelle kann nun entweder mit dem M2/C-Programmgenerator in ein Modula2- oder ein C-Programm übersetzt werden oder durch den UI-Interpreter interpretiert werden.

Mit SCT werden vier Ziele verfolgt:

1. Bereitstellung einer Umgebung zum experimentellen Programmieren in einer Sprache mit starker Typisierung.

2. Erweiterung der experimentellen Programmierung zum experimentellen Entwurf.

3. Unterstützung verschieder Programmierparadigmen (z.B. modulares, funktionales, objektorientiertes und logisches Programmieren).

4. Unterstützung der Ausführung von hybriden Softwaresystemen, die aus in Modula2 formulierten Programmteilen und Komponenten, die in anderen Hochsprachen programmiert sind, bestehen.

Das erste Ziel wird mit einem Modula2-Interpreter erreicht, der es aus Gründen der Geschwindigkeit erlaubt, aktuell zu implementierende Module interpretativ auszuführen, während die restlichen Module mit ihrem Objektcode ausgeführt werden.

Das zweite Ziel wird durch den Simulator realisiert. Er simuliert den Daten- und Kontrollfluß für noch nicht implementierte Module mit Hilfe ihrer Schnittstellendefinitionen, während bereits implementierte Module interpretiert oder direkt ausgeführt werden.

Das dritte und vierte Ziel wird erreicht durch die Möglichkeit, Interpreter anderer Sprachen in SCT einzubinden.

Die genannten Funktionen werden innerhalb von SCT unter der Oberfläche des Konfigurationsverwalters *CM* (Configuration Manager) vereinigt. CM verwaltet Informationen über

1. *Dokumente*, die Informationen über das Softwaresystem beinhalten (Modula2-Programmtext, Dokumentationen sowie Formalismen für den Übergang zu eingebundenen Interpretern)

2. *Attribute* zur Klassifizierung der Dokumente

3. *Pfade* zu den Dokumenten

4. *Objektbibliotheken*, die vom Softwaresystem benötigt werden

5. eingebundene Interpreter

Dokumente können in einer nach ihren Attributen sortierten Liste selektiert werden und in einem *Implementor* genannten Werkzeug bearbeitet, durchwandert, ausgeführt und getestet werden.

ProLab [BKKZ92] unterstützt das Programmieren mit Prolog und ist im Gegensatz zu Topos „nur“ eine Programmierumgebung, d.h. es soll nur die „technischen“ Aspekte der Entwicklung eines Softwaresystems wie Implementierung, Test und Projektverwaltung unterstützen. ProLab ist eine Sammlung von Werkzeugen, auf die im folgenden näher eingegangen wird.

Der *Environment Manager* verwaltet für jeden Benutzer (hier: Entwickler) die individuelle Konfiguration seiner Arbeitsumgebung.

Der *Configuration Manager* übernimmt die gerade beim Prototyping wichtige Verwaltung von Versionen und Varianten des Softwaresystems. Abhängigkeiten zwischen Komponenten der Programmierumgebung wie Dateien, Prozeduren, Dokumentation, Versionen und Varianten werden in einem *Data Dictionary* des Configuration Managers vermerkt.

Das *Session Logbook* führt Buch über die Folge von Kommandos einer Entwicklersitzung, so daß sie wiederholt, mit Hilfe von Funktionen zum Kopieren und Einsetzen verändert oder wiederverwendet werden können. Damit können einmal durchgeführte Tests und Evaluierungen für zukünftigen Prototypen wiederholt werden.

Der *Module Manager* speichert Beziehungen zwischen Dateien, Prozeduren und Dokumentationen auf der Basis einer Datenbank. Dateien und Prozeduren werden untereinander über eine „enthalten“-Relation und beide werden über eine „erklärt“-Relation mit der Dokumentation verbunden. Dieses Werkzeug soll es ermöglichen, den Entwicklungsprozeß beim systematischen Prototyping übersichtlich und nachvollziehbar zu halten.

Der *Window Manager* dient der Entwicklung interaktiver Prototypen mit graphischer Benutzungsoberfläche.

Der *Procedure Manager* überprüft jede neue oder geänderte Prozedur auf Namenskonflikte und Unstimmigkeiten in den Deklarationen.

Der *Program Checker* überwacht die Einhaltung von Programmierungskonventionen und versucht, fehlerträchtige Programmteile durch Verwendung von Heuristiken zu entdecken.

Mit dem *Call-Tree Analyzer* kann nach undefinierten Prozeduren oder nach abgeschlossenen Teilsystemen innerhalb des Softwaresystems gesucht werden.

Mit dem *Type-Checker* können nichttypisierte Sprachen (in diesem Fall speziell Prolog) – die ja gegenüber anderen bevorzugt zum Prototyping verwendet werden – auf Typkonflikte untersucht werden.

Der *Pretty Printer* sorgt für eine einheitliche Darstellung sowohl von Quelltexten als auch von zur Programmausgabe bestimmten Texten. Außerdem stellt er ein Werkzeug zur Gliederung und der damit erreichbaren Verbesserung der Lesbarkeit dar. Dadurch wird die Entwicklung von Prototypen in Gruppenarbeit unterstützt.

Als zentrales Instrument zur Verbindung der meisten genannten und noch folgenden Werkzeuge dient der von einem Entwickler individuell verwendete *Editor*. Nach dem Beenden einer Bearbeitung werden die verschiedenen Werkzeuge zur Überprüfung der bearbeiteten Datei, zum inkrementellen Compilieren, zum Projektmanagement usw. aktiviert. Speziell die unmittelbar zuvor genannten Werkzeuge, angefangen mit dem Prodcedure Manager, können während des Prototypings schrittweise hinzugefügt werden, um softwaretechnischen Anforderungen gerecht zu werden, ohne in frühen Phasen durch sie behindert zu werden.

Mit dem *Test Driver* können Tests geschrieben, bearbeitet und geladen werden. Test bestehen aus Testumgebungen, Testdaten und den entsprechenden erwarteten Ergebnisdaten, die vom Test Driver automatisch mit den tatsächlich erzeugten verglichen werden können. So können einheitliche Tests für verschiedene Prototypen durchgeführt werden.

Beim *Debugger* wird zwischen einfachen und komplexen Debuggern unterschieden. Ein einfacher Debugger basiert auf der schrittweisen Ausführung, mit Möglichkeiten zur Parametrisierung, Setzen von Kontrollpunkten und Aussparung von Prozeduren beim schrittweisen Ausführen. Ein komplexer Debugger kann u.a. folgende Fragen beantworten: Wird dieser Prozedur immer vor Ausführung einer bestimmten Sorte von Unterprogrammen aufgerufen? Welche Prozeduren hat dieses Datum durchlaufen? An welchen Stellen wurde es verändert? In ProLab werden dazu die Ausführungsschritte eines Programms protokolliert und im Anschluß an die Ausführung analysiert.

Der *Source/Document Processor* ermöglicht die Integration von Programmtext und Dokumentation in einer Datei. Wahlweise kann Programmtext oder Dokumentation extrahiert werden. Hierdurch wird die schnelle Konstruktion von Prototypen ohne Vernachlässigung der Dokumentation unterstützt.

Der *Document Manager* fungiert als eine Art Kontrollinstanz, indem er auf fehlende Dokumentationen zu neuen oder geänderten Programmteilen aufmerksam macht. Diese Situation entsteht gerade beim Prototyping recht häufig. Zusätzlich erinnert der Document Manager an Termine und Vereinbarungen.

Der *Tutor* verbindet die Benutzerdokumentation mit der Systemdokumentation zu einer kontextabhängigen Hilfe. Dieses Werkzeug stellt auch Demonstrationen von Prototypen und Richtlinien für die Entwicklung und den Vergleich von Prototypen zur Verfügung.

Das *Newsboard* dient dem Nachrichtenaustausch zwischen Entwicklern und zwischen dem Kunden und den Entwicklern während des Tests eines Prototyps.

15.3 Formale Spezifikation

15.3.1 Komponentenorientiertes Spezifizieren (Stefan Schüler)

Um Flexibilität, Standardisierung, Qualität und Wiederverwendbarkeit in der Verarbeitung und Entwicklung von Software zu unterstützen, hat es sich als nützlich erwiesen, die Architektur eines Software-Paketes in eigenständige Blöcke, sogenannte *Software-Komponenten*, zu unterteilen. Software-Komponenten sind Einheiten, die bestimmte Dienste durch ihre Schnittstelle zur Verfügung stellen und lokale Datenstrukturen kapseln, die diesen Dienst realisieren.

Zunächst werden wir component description languages untersuchen und danach den Begriff der Software-Komponente im Kontext der formalen Spezifikationssprache Π untersuchen.

Spezifikation von Software-Komponenten

Software-Komponenten erlangen immer größeres Interesse, weil dieses Konzept die Wiederverwendbarkeit von Software und flexible Entwicklungen großer Softwaresysteme unterstützt sowie eine Standardisierung und Qualität erreicht wird, wie man es von anderen Ingenieursdisziplinen gewohnt ist.

Welche Anforderungen sind an Software-Komponenten zu stellen und wie sollen sie durch Programmiersprachen unterstützt werden?

Aufgrund des Aspektes der Wiederverwendbarkeit ist ein Charakterisitikum einer Komponente sicherlich, daß sie ohne Probleme aus ihrem Zusammenhang entfernt werden kann, ohne daß sie dadurch ihre Semantik verliert. Als Beispiel für solch eine Komponente kann eine Prozedurdeklaration in Pascal oder ein Prädikat in Prolog gesehen werden; eine Komponente ist also eine syntaktische Einheit, die anhand ihres Namens verwendet werden kann [GSC91].

Um diese Art der Verwendung zu erreichen, ist es nötig, Möglichkeiten der *Datenkapselung* zu untersuchen. Es kann sein, daß keine Sprachkonstrukte zur Verfügung gestellt werden, um Teile einer Komponente zu kapseln, wie es z.B. in Pascal geschieht, wo nur

Rudimente wie das Prozedurkonzept existieren. Wenn eine syntaktische – oder schwache – Datenkapselung erreicht wird, so muß nur die syntaktische Korrektheit zur Verwendung eines gültigen Ausdrucks beachtet werden. Dieses Konzept wird mit der Schnittstellendefinition in Modula verfolgt. Kann man hingegen eine semantische – oder starke – Datenkapselung verwenden, werden Konstrukte benutzt, die sowohl die Semantik als auch die Randbedingungen für die korrekte Benutzung der öffentlichen Teile einer Komponente spezifizieren. Beispiele dieser Art der Kapselung sind ACT TWO [EM90] und Π, wo Schnittstellen von Modulen aufgrund ihrer algebraischen Spezifikation definiert werden.

Diese Beziehungen zwischen Komponenten können sowohl *implizit* durch bloßes Nennen des Namens wie z.B. globale Namen in Prozedurrümpfen erreicht werden als auch *explizit* durch aktuellen Import oder durch formalen Import.

Weitere Aspekte von Komponenten beruhen auf der *Ausdrucksstärke der Programmiersprache*, inwiefern durch sie Parallelität, Fehlerbehandlung und Persistenz unterstützt wird.

Aus Sicht der *Prozeßentwicklung* sind gute Strukturierungsmöglichkeiten in einer problemorientierten und verständlichen Art nicht die Garantie dafür, ein schlechtes Design zu verhindern. Deswegen muß eine component description language alle Software-Entwicklungsstadien abdecken und für fein granulare Übergänge sorgen.

Auch *Design-Entscheidungen* wie *divide-and-conquer* beeinflussen Komponenten–Beschreibungen, weil dadurch kleinere Einheiten modelliert werden können. Als Indiz für die Tauglichkeit der Konstrukte, die für diesen Ansatz von der component description language zur Verfügung gestellt werden, ist der resultierende Graph, der die Beziehungen zwischen den Komponenten in seiner aktuellen Konfiguration wiedergibt: Ein azyklischer Graph oder Baum unterstützt divide-and-conquer besser als Graphen mit Zyklen. Das Prinzip des *separation of concerns* beeinflußt den Entwicklungsprozeß dahingehend, daß die Eigenschaften getrennt spezifiziert werden, die nicht zueinander gehören; dies wird sowohl durch Modularisierung oder Strukturierung der Komponenten als auch durch verschiedene View-Typen (vgl. 15.3.2) erreicht. Dadurch ist es möglich, durch verschiedene Realisierungen und unterschiedliche Umgebungen, in denen Komponenten wiederverwendet werden können, einen Pool von Komponenten zu schaffen, auf die man bei Bedarf zurückgreifen kann.

Es wurden Sprachen entwickelt, um diese Methoden zu realisieren und die die Konstruktion und Verwaltung von Programmen zu ermöglichen, die aus einer Zusammenstellung von Komponenten bestehen. Diese Programmiersprachen unterscheiden sich in ihrer semantischen Tiefe, mit der sie Komponenten unterstützen, und zwar von einfachen syntaktischen Konstrukten wie das Funktionenkonzept in C bis zu formalen Spezifikationssprachen wie ACT TWO oder Π, die die formale Definition von Komponentenbeziehungen erlauben.

Software-Komponenten in Π

Π unterstützt die Spezifikation von Software-Komponenten, mit deren Hilfe ein Software-System entwickelt wird [SG94]. Dafür müssen die erforderlichen Komponenten identifiziert und deren Beziehungen untereinander definiert werden. Komponenten haben genau spezifizierte Schnittstellen, die alle Informationen beinhalten, um eine Komponente als selbständige und unabhängige Einheiten modellieren zu können. Eine Zusammenstellung mehrerer solcher Komponenten wird modulares System genannt.

Das Komponentenkonzept von Π berücksichtigt folgende Eigenschaften:

- Jede Komponente kapselt einen oder mehrere Datentypen, um eine objektbasierte Strukturierung zu fördern.
- Nur azyklische Relationen zwischen den Komponenten sind erlaubt, um eine hierarchische Strukturierung des Systems zu erhalten.
- Komponentenkomposition, d.h. mehrere Komponenten können zu einer neuen ausführbaren Komponente zusammengefaßt werden.
- Jede Komponente besitzt eine Importschnittstelle, um Anforderungen für die realisierten Datentypen zu erhalten. Dabei wird das Konzept des formalen Importes verfolgt: Einzelne Komponenten können zunächst unabhängig entwickelt werden, aber bei der Relation von Komponenten muß eine passende Verbindung zwischen Export- und Importschnittstelle realisiert werden.
- Importierte Datentypen, die keine Auswirkung auf die Realisierung haben, können wieder exportiert werden, um Datentypen von unteren Ebenen auch in höheren Ebenen sichtbar zu machen. Diese Datentypen werden in der Common Parameter Section festgelegt.

Eine Software-Komponente in Π kann – wie Figur 15.3 darstellt – als 5-Tupel SC=(SCN, EXP, IMP, CP, BODY) aufgefaßt werden:

- **SCN**: Name der Komponente
- **EXP**: nicht leere Menge von Datentypen zur Beschreibung zur Verfügung gestellter Funktionalität für andere CEM's – CEM bedeutet Concurrently Executable Module (*Export Section*)
- **IMP**: eventuell leere Menge von Datentypen, die die möglichen Anforderungen beinhaltet (*Import Section*)
- **CP**: ggf. leere Menge von Datentypen, die die Beziehungen importierter und unverändert exportierter Datentypen verdeutlicht (*Common Parameters section*). Sie sind konzeptionell nicht notwendig.

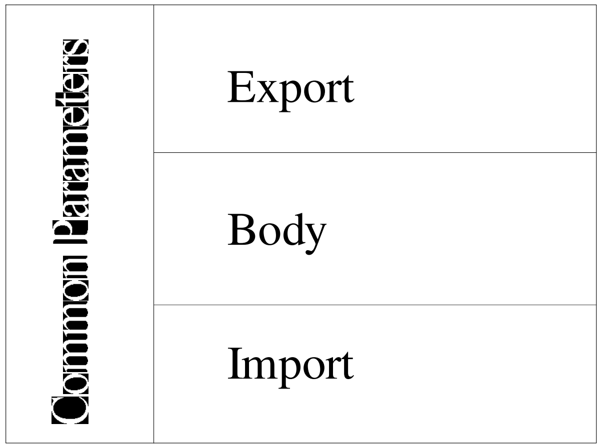


Abbildung 15.3: Software-Komponente

- **BODY** $\supseteq$ EXP: nicht leere Menge von Datentypen, die die importierten Datentypen behandelt und die Exportmöglichkeiten verfügbar macht (*Body section*)

Die Schnittstellen sind folgendermaßen definiert:

- **EXP_INT** := EXP $\cup$ CP (*Export Interface*)
- **IMP_INT** := IMP $\cup$ CP (*Import Interface*)

Während die Schnittstellen Formalismen schaffen, die Beziehungen einer Komponente zu anderen Komponenten herstellen, wird im Rumpf die Realisierung des gekapselten Datentyps festgelegt. Es gibt drei Realisierungsmöglichkeiten, die mit drei Komponentenarten korrelieren:

- **ps-construction** (programming-in-the-small):
  Realisierung einzelner Datentypen in einer Komponente (*Atomic Component*)
- **pl-construction** (programming-in-the-large):
  Verbindung mehrerer Komponenten zu einer neuen, komplexeren Komponente (*Composite Component*)
- **basic-construction**:
  Keine Informationen bzgl. der verwendeten Operationen vorhanden (*Basic Component*)

Die Sprache Π wurde mit dem Ziel entwickelt, eine formale Spezifikationssprache zu haben, in der

- eine modulare und hierarchische Struktur für die Beschreibung eines Systems erzwungen wird,
- ein einziges und somit einheitliches Strukturkonzept verfügbar ist,
- die Sprache um das Konzept des abstrakten Datentyps gesetzt wurde.

Wie wurden diese Ziele erreicht?

Das Hauptkonzept in Π ist die CEM-Spezifikation, was Concurrently Executable Module bedeutet, die die Beschreibung einer Klasse von Objekten ist. Ein Objekt ist z.B. die Instanz eines abstrakten Datentyps, der sowohl komplex als auch atomar sein kann.

Mit Hilfe des in 15.3.2 vorgestellten View-Konzeptes wird eine strenge semantische Kapselung erreicht, das Prinzip des *separation of concerns* verwirklicht und die *Common Parameter Section* unterstützen eine klar erkennbare Systemarchitektur.

Wenn man die in dem vorherigen Kapitel erarbeiteten funktionellen Anforderungen an eine component description language Punkt für Punkt durchgeht, sieht man, daß sie alle verwirklicht wurden, was ja auch das Ziel der Entwicklung der Spezifikationssprache Π war.

Wenn man aber noch andere wichtige Aspekte wie *Leistung* und *Zuverlässigkeit* berücksichtigen will, muß man wohl einen weiteren View-Typ hinzufügen oder eine weitere Schnittstelle, um auch solche Faktoren spezifizieren zu können. Obwohl diese Punkte wichtig sind, ist es doch schwierig sie zu erfassen, weil sie doch ein gewisses Maß an Subjektivität mit sich bringen, was sie aber nicht davon ausschließen sollte, so früh wie möglich im Software-Entwicklungsprozeß eingebunden werden.

15.3.2 Die Spezifikation von Softwarekomponenten in Π (Peter Jodeleit)

Das Konzept in Π ist eine Objektbasierte Strukturierung von Software. Dabei sind in den Objekten die Daten durch Operationen gekapselt und die zugrundeliegenden Datentypen der Objekte stehen in der Modulspezifikation.

Jedes Modulsystem besteht aus einer Hierarchie von Modulen, in denen jeweils Objekte eines bestimmten Datentyps gekapselt sind. Da die Operationen auf den Objekten parallel ausgeführt werden können, werden die Basisbausteine CEM (Concurrently Executable Module) genannt.

Jede Π–Komponente besteht aus vier Teilen,

- Export–Teil,
- Import–Teil
- Common–Parameters–Teil und
- Body–Teil.

Spezifikation der Komponenten–Schnittstelle

Jede Komponenten–Schnittstelle einer Π–Komponente enthält Datentypen, die anderen Komponenten zur Verfügung gestellt werden. Um die Beschreibungen der Eigenschaften zu strukturieren, werden in Π Views verwendet.

Views sind Spezifikationen, die nur gewisse Aspekte eines Datentypen berücksichtigen und andere vernachlässigen, die dafür dann in anderen Views spezifiziert werden. Jeder View hat eine eigene Darstellungsart, dessen Genauigkeit sich zwischen formal und natürlichsprachlich bewegt.

Durch das View–Konzept kann man sich bei der Entwicklung auf einen bestimmten Aspekt konzentrieren. Es können nach Bedarf neue Views zu den bestehenden hinzugefügt werden.

Zur Zeit gibt es drei Views in den Komponenten–Schnittstellen:

- den Type View, der die Invarianzeigenschaften eines Datentyps beschreibt,
- den Imperative View, der die Seiteneffekte beschreibt,
- den Concurrency View, der die möglichen Rangfolgen der Operationsaufrufe beschreibt.

Der Type–View

Algebraische Spezifikation ist eine Technik, in der Invarianzen von Objektklassen oder –typen durch Gleichungen zwischen den auf den Typen definierten Operationen spezifiziert werden. Durch diese Gleichungen wird meistens auch gleichzeitig die Funktionalität eines Datentyps charakterisiert. Für die Gleichungen sind verschiedene Notationsarten entwickelt worden.

Eine Spezifikation besteht im allgemeinen aus vier Teilen:

- einer Einleitung, in der die spezifizierte Sorte eingeführt wird und die Namen aller benutzten Spezifikationen aufgeführt werden,
- einer informellen Beschreibung der Operationen, die auf dieser Sorte definiert werden,
- einer Signatur, d.h. Art und Zahl der Parameter und Art der Ergebnistypen der Operationen und
- den Axiomen, in denen das Verhältnis zwischen den Operationen auf der Sorte spezifiziert wird.

Durch eine algebraische Spezifikation wird der Entwurf von Operationen einerseits und die Spezifizierung des Verhaltens der Operationen untereinander andererseits, miteinander verbunden.

Die Algebraische Spezifikation ist eine Technik, die sich sehr gut für eine Schnittstellenspezifizierung eignet und gut mit Objektbasiertem Design kombiniert werden kann.

Die normale Vorgehensweise bei der Algebraischen Spezifikation, um komplexe Datentypen zu definieren, ist, auf vorhandenen einfacheren Datentypen aufzusetzen. Diese können entweder als Bausteine verwendet werden oder erweitert werden[2]. Weiterhin können mittels generischer Parameter generische Datentypen definiert werden, um so schnell zu mächtigen Datentypen zu kommen.

In II wird die Technik der Algebraischen Spezifikation im Type View benutzt. Allerding werden keine generischen Datentypen zuf Verfügung gestellt, ebenso existiert nicht die Möglichkeit des enrichments. Auch die äußere Form weicht von der oben beschriebenen Art in einigen Punkten ab:

- direkt vor jedem Typ kann eine informelle Beschreibung stehen,
- die Signatur steht direkt bei der entsprechenden Operation,
- nach jeder Operation kann ein erklärender Text folgen,
- Axiome folgen direkt auf die Operationen, auf die sie sich beziehen.

Diese Reihenfolge bezieht sich auf export, common parameters und import section. Durch diese Form stehen Operationen, die in einem engen Kontext stehen, auch nahe beieinander, zusammen mit Erläuterungen, Signatur und den Axiomen, die sie erfüllen.

- Die Spezifizierung der Datentypen und Operationen werden auf export, common parameters und import section verteilt, je nachdem ob sie für den Benutzer sichtbar sein sollen, für interne Zwecke einer Komponente benötigt werden oder importiert werden.

Parametertypen und die dafür definierten Operationen sind kein syntaktischer Bestandteil der export section, sondern jeder Datentyp hat eine eigene Spezifikation, wodurch die getrennte Entwicklung unterstützt wird. Sie werden entweder unter den common parameters oder der import section aufgeführt. Erst die Vereinigung der Spezifikationen aller Teile ergibt eine komplette Spezifikation im algebraischen Sinne[3].

In der body section werden meist weitere Datentypen benötigt, die wesentliche Teile des beschriebenen Objektes und dessen Konstruktion ausmachen. Da deren Spezifikation vor dem Benutzer der Komponente verborgen bleiben sollten, werden sie in der body section spezifiziert.

Nicht für alle denkbaren Datentypen kann durch diese Art der Spezifizierung auch die Funktionalität der Operationen spezifiziert werden. Einerseits gibt es theoretische Grenzen bezüglich der Ausdrucksfähigkeit, andererseits sprechen softwaretechnische und pragmatische Gründe dagegen.

Ein Grund ist beispielsweise, daß, um ein sichtbares Verhalten zu spezifizieren, es notwendig werden kann, auf die interne Repräsentation eines Datentyps zurückzugreifen.

[2] Diese Technik nennt sich „enrichment“

[3] Der Datentyp `Boolean` stellt eine Ausnahme dar, er muß nicht explizit aufgeführt werden.

Dadurch würde das Prinzip des information–hiding gebrochen. Manchmal müßten auch zusätzliche Datentypen eingeführt werden, um ein sichtbares Verhalten zu spezifizieren, obwohl sie aus der Sicht des Benutzers nicht nötig wären.

Eine Klasse von Datentypen, die sich auf algebraische Art sehr schwer spezifizieren läßt, sind z.B. Datentypen mit Operationen, die eine Folge von Objektzuständen erzeugen und nicht einen einzelnen Zustandsübergang. Ergeben sich Zwischenzustände, die nur von einer Operation erzeugt werden, gibt es keine Möglichkeit, das Ergebnis einer auf diesen Zwischenzustand angewandten Operation zu definieren, ohne auf die interne Realisierung zurückzugreifen. Diese Art von Operationen kommen beispielsweise in polling systemen o.ä. vor.

Außer diesen Einschränkungen ist der type view ein nützliches Werkzeug zur formalen Spezifizierung wichtiger Eigenschaften von Datentypen. Weiterhin können dadurch auch semantische Informationen der Komponentenschnittstellen gegeben werden.

15.4 Die imperative Sicht in Π (Markus Brameier)

15.4.1 Spezifikation der Schnittstelle

Zusätzlich zu den Informationen der Typsicht spezifiziert die imperative Sicht auch die Namen der formalen Parameter und klassifiziert alle als in-, inout- oder return-Parameter, wobei „in“ *call-by-value-* und „inout“ *call-by-reference*-Parameter bezeichnet. Lassen sich aus gut gewählten Objektnamen die informellen Semantiken der Operationen ableiten, so werden durch die Klassifikation der Parameter die möglichen Seiteneffekte einer Operation wohldefiniert. Seiteneffekt sind nur über inout-Parameter möglich, weil keine globalen Variablen in Komponenten existieren (Prinzip der strengen Kapselung). Die in- und inout-Parameter entsprechen den Argumenttypen in der Typsicht. Der Ergebnistyp ist entweder als inout- oder als return-Parameter realisiert. Ein als inout-Parameter einer Operation übergebenes Objekt wird in der Operation *verändert* (*objekt-modifizierende Operation*). Das durch einen return-Parameter spezifizierte Ergebnis einer Operation ist dagegen immer ein neu *erzeugtes* Objekt (*objekt-liefernde Operation*).

Beispiel: Ausschnitt aus der Spezifikation der Monitor-Komponente
...
imperative view specification

  export

    type *Monitor*

      operation init_monitor (**in** r:Range; a:Act_Position) **returns** Monitor

```
        operation run_monitor (inout m:Monitior)

        operation stop_monitor (inout m:Monitior)
        ...
...
```

15.4.2 Spezifikation des Komponentenkörpers

Die Implementation einer Komponente erfolgt hier mit imperativen Algorithmen und Sprachstrukturen. In der imperativen Sicht wird so der Daten- und Kontrollfluß in den Operationen deutlich. Die Sprachelemente sind denen moderner Hochsprachen wie Pascal ähnlich. Eine Besonderheit in Π ist die **„cobegin...||...coend“**-Anweisung (innerhalb des **begin...end**-Rumpfes einer Operation), die eine parallele Ausführung und Synchronisation von Programmsequenzen ermöglicht. Zudem wird in der imperativen Sicht die Kommunikation von Komponenten über geteilte Daten unterstützt. Die **„share <Objektname> with <Objektname>,... end share“**-Anweisung legt dabei das geteilte Objekt (= Instanz einer Komponente) und die Objekte, mit denen es geteilt wird, fest. Ein geteiltes Objekt ist gleichzeitig Teil mehrerer (komplexer) Objekte.

Beispiel: Producer-Consumer-Prinzip
(kein „-Problem“, weil nur eine Komponente gleichzeitig ausführbar ist.)

```
cem PRODUCER_CONSUMER
  ...
  imperative view specification
    export
      type producer_consumer
      ...
    import
      type buffer
      type producer
      type consumer
      ...
    body
      ...
        operation init_producer_consumer (...) returns producer_consumer
          declare
            p :producer;
            c :consumer;
            b :buffer;
          begin
```

```
            share b with p,c end share; b wird implizit als call-by-reference übergeben.
            p := init_producer (b,...); Objekt-Initialisierungen
            c := init_consumer (b,...);
            ...
            return make_tuple (p,c);
        end
    ...
end cem PRODUCER_CONSUMER
```

15.4.3 Nebenläufigkeit (Knuth Waltenberg)

Einleitung

Diese Abschnitt beschäftigt sich mit der Spezifikation von Parallelität[4] von Operationen innerhalb eines Moduls durch Pfadausdrücke – zuerst in allgemeiner Form, dann speziell in der Sprache Π. Die Spezifikation mit Pfadausdrücken ist eine Methode unter mehreren, die aber hier nicht oder nur rudimentär behandelt werden. Auch kann die behandelte Methode auf Parallelitätsbeziehungen zwischen Modulen bzw. Objekte ausgeweitet werden. Dies wird ebenfalls hier nicht weiter verfolgt. Interessierte sollten die Dissertation von Frau Silke Seehusen lesen [See87].

Voraussetzung für die Pfadausdruck-Methode ist, daß die Entwicklung der statischen Struktur des Sofwaresystems auf Basis des Modulkonzeptes erfolgt ist. Dies ist in Π der Fall. Die Grundidee besteht nun darin, aus den abgeleiteten Anwendungen in der Praxis auf die erlaubten parallelen Ausführungen innerhalb des Moduls zu schließen.

Integritätsbedingungen

Eine mögliche parallele Ausführung von Operationen in einem Modul wird beschränkt durch sogenannte Integritätsbedingungen. Diese Integritätsbedingungen beschreiben die erlaubten Werte der Objekte (Daten) und die erlaubten Veränderungen der Werte. Es soll hier angenommen werden, daß eine Ausführung einer Operation für sich alleine keine Integritätsbedingungen verletzt. Eine parallele Ausführung kann aber gegen die Bedingungen verstoßen. Zur Lösung dieses Problems muß die Parallelität der Operatoren eingeschränkt werden. Dies sollte aber nicht erst in der Implementationsphase geschehen, da eine Modulspezifikation eine komplette Beschreibung des Moduls sein sollte. Außerdem sollte die Semantik der Operationen und Daten berücksichtigt werden. Dies geschieht z.B. nicht in dem Kriterium der Serialisierbarkeit von Transaktionen in Datenbanksystemen. Ein weiteres Konzept, das die parallele Ausführung von Operationen

[4] Allgemein versteht man unter „nebenläufig“ die unabhängige Bearbeitung von Operationen oder Prozessen. Dies ist ein allgemeinerer Begriff als „parallel“. Die Art der rechnerinternen Verarbeitung ist aber hier nicht relevant, so daß beide Begriffe synonym zueinander benutzt werden können.

auf gleichen Objekten einschränkt, ist das Monitor-Konzept von C.A.R. Hoare. Dieses Konzept müßte einigen Leuten aus der Vorlesung Betriebssysteme bekannt sein. Es beschränkt die parallele Ausführung so stark, daß die Operationen nicht mehr parallel behandelt werden können. Die nebenläufige Ausführung sollte aber gefördert werden, da sie eines der Ziele der heutigen Softwareentwicklung ist.

Spezifikation der Parallelität mit Pfadausdrücken

Die Methode der Pfadausdrücke versucht möglichst viel der potenziellen Parallelität zu erhalten und berücksichtigt die Semantik der Operationen und Daten. Pfadausdrücke wurden von R.H. Campbell und A.N. Habermann eingeführt. Sie sind ein wesentliches Konzept der Softwareentwicklungsumgebung COSY. Außerdem wurden sie in die Programmiersprache „Path Pascal" und deren Weiterentwicklung „Distributed Path Pascal" verwendet und nicht zuletzt in der Softwarebeschreibungsprache Π, die hier behandelt wird.

Pfadausdrücke

Mir Pfadausdrücken wird die erlaubte parallele Ausführung von Operationen beschrieben. Einige Operationen dürfen nur sequentiell ausgeführt werden, andere aber auch parallel zueinander. Im Monitorbeispiel dürfen `create_file` oder `open_file` nur vor den anderen Operationen des Datentyps ausgeführt werden. Im folgenden wird nun die syntaktische Form der Pfadausdrücke in Π, die BNF-ähnlich ist, beschrieben:

Operationsname: Die einfachsten Pfadausdrücke sind Operationsnamen. Der Pfad, der durch diesen Ausdruck definiert wird, besteht aus einer Operationsinitiierung und einer folgenden Beendigung. Der Pfad besteht aus genau einer Ausführung der Operation.

```
operationsname::= name
```

Einfacher Ausdruck und Ausdruck: Wie in arithmetischen Ausdrücken kann aus einem Ausdruck wieder ein Ausdruck aufgebaut werden. Ein Ausdruck ist ein einfacher Ausdruck, eine Sequenz, eine Alternative oder Parallelität.

```
ausdruck::= einfacher_ausdruck |
            sequenz |
            alternative |
            parallelitaet
```

Ein einfacher Ausdruck kann ein Operationsname sein. Außerdem kann er ein geklammerter Ausdruck sein. Dies dient zur Bildung von komplexen Ausdrücken. Die weiteren Alternativen der Regel werden nachfolgend erklärt.

```
einfacher_ausdruck::= operationsname |
```

```
"(" ausdruck ")" |
wiederholung |
parallelitaetsabschluss |
option |
pfadname
```

Sequenz: Eine Sequenz oder Aufeinanderfolge von Pfaden bedeutet, daß zuerst der Pfad bearbeitet werden muß, der links vom Symbol steht und danach der rechte.

```
sequenz::= einfacher_ausdruck ";" einfacher_ausdruck
```

Alternative: Eine Alternative von Pfaden (gegenseitiger Ausschluß, mutual exclusion) bedeutet, daß nur eine der beiden Möglichkeiten zu einem Zeitpunkt durchlaufen werden darf.

```
alternative::= einfacher_ausdruck "|" einfacher_ausdruck
```

Parallelität: Mit diesem Konstrukt wird angegeben, daß zwei Pfade zeitlich überlappend oder in beliebiger Folge durchlaufen werden dürfen. Nur die Ordnung innerhalb der Pfade muß sichergestellt werden.

```
parallelitaet::= einfacher_ausdruck "+" einfacher_ausdruck
```

Wiederholung: Mit diesem Konstrukt wird die Möglichkeit beschrieben, einen Ausdruck beliebig oft sequenziell zu wiederholen. Der Ausdruck muß mindestens einmal durchlaufen werden.

```
wiederholung::= "[" ausdruck"]"
```

Parallelitätsabschluß: Bei einem Parallelitätsabschluß ist es möglich einen Pfad gleichzeitig und auch hintereinander beliebig oft zu durchlaufen. Auch hier muß der Pfad mindestens einmal durchlaufen werden.

```
parallelitaetsabschluss::= "{" ausdruck "}"
```

Option: Eine Option beschreibt die Bedingung, daß der Ausdruck vollständig oder gar nicht durchlaufen werden muß.

```
option::= "(*" ausdruck "*)"
```

Pfaddefinition: Eine Pfaddefinition soll den Pfadausdruck eines Moduls festlegen. Sie besteht aus der Angabe des Pfadausdrucks und eventuell einer Reihe von Definitionen von Teilpfaden. Ein Pfadausdruck der Pfaddefinition beschreibt den Pfad, der beliebig oft wiederholt werden kann. Dieser steht deshalb implizit in Wiederholungsklammern.

```
pfaddefinition::= pfadausdruck { benannter_ausdruck }

pfadausdruck::= path_expression ausdruck path_name_list |
                 path_expression  path_name_list

benannter_ausdruck::= pfadname "::=" einfacher_ausdruck

pfadname::= name
```

„path expression“ und „path name list“ sind in Π benutzte Schlüsselwörter.

Assoziativität Die Operatoren für die Sequenz, die Alternative und Parallelität sind assoziativ (Beweis siehe Dissertation von Silke Seehusen). Deshalb können einige runde Klammern weggelassen werden, obwohl sie laut obiger Definition verwendet werden müßten.

Modulpfadausdrücke

Die oben genannten Pfadausdrücke werden nun wie folgt verwendet. Für ein Modul wird ein Pfadausdruck angegeben – der Modulpfadausdruck. Dieser beschreibt die erlaubten Verarbeitungsfolgen der Operationen auf einem Objekt. Er ist für jedes Objekt eines Moduls gleich.

Ein Modulpfadausdruck legt die Folgen der Operationsausführung fest, die ein Modul anderen Modulen anbietet. Somit gehört er zu der Exportschnittstelle des Moduls. Deswegen dürfen in ihm auch nur exportierte Datentypoperationen vorkommen. Dies ist in Π durch die „Concurrency View“ umgesetzt worden.

Die Concurrency View (nebenläufige Sicht) in Π

Die nebenläufige Sicht ist eine der drei Sichten in der Exportschnittstelle von Modulen in Π. Die Syntax der Pfadausdrücke in der nebenläufigen Sicht ist in den vorhergehenden Abschnitten bereits beschrieben worden.

Die nebenläufige Sicht bietet aber noch eine weitere Möglichkeit Verarbeitungsbeschränkungen zu beschreiben. Teilweise müssen für die Ausführung von Operationen bestimmte Vorbedingungen erfüllt sein. Zum Beispiel darf die Operation `reset_alarm` aus dem Monitorbeispiel nur benutzt werden, wenn ein Alarm auch besteht. Dies wird in Π durch einen einfachen Ausdruck am Ende der Pfadbeschreibung erreicht. Hier ein Beispiel:

```
precondition definition list
   precondition of reset_alarm(m) is not (is_alarm(m))
```

15.4.4 Die Spezifikation von Konfigurationen (Knuth Waltenberg)

Die Sammlung von Komponenten, welche gemäß ihren Schnittstellenspezifikationen miteinander verbunden sind, nennt man Konfiguration. Diese wird in Π in einer Konfigurationsspezifikation genauer beschrieben. Diese Spezifikation enthält erstens eine Aufzählung der enthaltenen Komponenten, zweitens eine Aufstellung der Verbindungen zwischen den Komponenten und drittens eine Aufstellung der Verbindungen der Komponenten mit den exportierten Datentypen der Exportschnittstelle und den importierten Datentypen der Importschnittstelle. Dies wird nun genauer beschrieben.

Die Komponenteninkarnation

Wenn spezifizierte Datentypen unterschiedlich interpretiert werden, führt dies zu verschiedenen Instantiierungen dieses Typs. Wird zum Beispiel der Typ „Item" verschieden interpretiert, führt dies zu verschiedenen Instantiierungen einer „List of Items", nämlich zu einer Liste von Integerwerten, zu einer Liste von Buchstaben u.s.w..

Verschieden Instantiierungen einer Komponente nennt man Komponenten- inkarnation. In Π stehen diese am Anfang jeder Konfiguration. Sie bestehen aus einem eindeutigen Namen und dem Name der betroffenen Komponente.

Beispiel: Das Modul PCS_Unit_1

```
configuration PCS_Unit_1
  ...
  component incarnations
  Monitor: MONITOR;
  Valve  : VALVE;
  ...
  component connections
  ...
end configuration PCS_Unit_1
```

Die Komponentenverbindungen

Es ist eine syntaktische Notwendigkeit, daß jede Operation einer Importschnittstelle mit einer exportierenden Operation verbunden sein muß. Beide Operationen müssen die gleiche Anzahl an Argumenten besitzen und die Operationsabsichten müssen übereinstimmen, d.h. objektmodifizierende Operationen dürfen nur auf objektmodifizierende Operationen abgebildet werden und Operationen, die Objekte zurückgeben, dürfen nur auf solche abgebildet werden die dies ebenfalls tun.

Außerdem müssen auf der semantischen Seite ebenfalls Bedingungen erfüllt sein, die nun überprüfbar sind: Zum Beispiel, wenn alle Komponenten einer Konfiguration algebraisch spezifiziert sind (in der Type View), so kann man die algebraische Korrektheit der

Komponentenverbindungen überprüfen. Auch durch die nebenläufige Sicht sind Abstufungen von Kompatibilität definiert, die es ermöglichen eine Verbindung zwischen zwei Komponenten dahingehend zu kontrollieren, ob sie erlaubt ist oder nicht. Die Grundidee ist hier, daß die Menge der erlaubten Verarbeitungsreihenfolgen eine Teilmenge der der möglichen Verarbeitungsreihenfolgen ist.

Eine korrekte Beschreibung der Verbindungen setzt voraus, daß die Komponenten vollständig definiert sind. Dies ist in Π bereits in der Inkarnation geschehen. Deshalb schließt sich an ihr die Aufzählung aller Komponentenverbindungen des zu spezifizierenden Moduls an. Beispiel:

```
component connections
   connection of Monitor
      from Monitor_Tuple import
         type Monitor_Tuple <- Tuple_5
            operations
              make_monitor_tuple <- make_tuple_5;
              get_range <- select_1;
              ...
      from Valve import
         type Actuator <- Valve
            operations
              init_actuator <- init_valve;
              get_position <- get_valve_position;
              ...
      ...
 end configuration PCS_Unit_1
```

Die Verbindungen mit der Export- und Importschnittstelle

Zuletzt findet eine Abbildung aller exportierten Datentypen auf die Exportschnittstelle und eine Sammlung aller offenen Importe der benutzten Komonenten in der Importschnittstelle statt.

Dieser Schritt macht für den Export deutlich, welche Datentypen im Innern der beteiligten Komponenten konstruiert wurden. Er macht ebenfalls überprüfbar, welche Komponente einer Konfiguration den exportierten Datentyp kennt. Außerdem ermöglicht er eine Umbenennung der Datentypen.

Die Abbildung auf die Importschnittstelle ermöglicht eine Zusammenfassung aller offenen Importe der beteiligten Komponenten zu einer Importspezifikation. Dies ermöglicht weiterhin, daß Datentypen die importiert werden und verschiedene Datentypen- und Operationsnamen benutzen aber den gleichen Datentypen meinen, zu einem Importdatentyp der Konfiguration zusammengefaßt werden können.

Die Syntax dieser Abbildungen entspricht der der Komponentenverbindungen, wodurch sich ein Beispiel erübrigt.

15.5 Die inkrementelle Entwicklung von Software-Komponenten (Markus Brameier)

Der hier vorgestellte Ansatz einer komponenten-orientierten Entwicklung beschreibt den stufenweisen inkrementellen Entwurf einer Software-Architektur aus einzelnen unabhängigen Komponenten, als Grundlage für die (formale) Spezifikation mit Π. In den Komponenten werden Funktionen mit (persistenten) Daten. Eine Komponente wird als ein Objektyp im System verstanden und bildet eine wiederverwendbare Softwareeinheit, in der Funktionen mit (persistenten) Daten gekapselt beschrieben werden. Flexibilität, im Sinne einer leicht korrigierbaren und anpaßbaren Software-Architektur, kann nur erreicht werden, wenn sich die Entwicklung ebenso auf die Beziehungen zwischen den Objekten in einem System wie auf die Objekte selbst konzentriert. Unterstützt wird die Entwicklung einer allgemeinen Lösung des Problems, aus der sich ein konkretes System durch Spezialisierung ableiten läßt. Die komponenten-orientierte Entwicklung stellt eine neue Methode zur Lenkung der verschiedenen Stufen des Entwurfs von der informellen Problemdefinition bis zur Modulhierarchie dar. beschriebenen Oft ist der lineare Durchlauf durch die vier im folgenden beschrieben Entwicklungsschritte mehrmals während des Entwurfs erforderlich, bis eine zufriedenstellende Software-Architektur gefunden ist:

15.5.1 Definition der Systemziele

In diesem Entwicklungsschritt werden die grundlegenden Systemfunktionen informell aufgelistet, so wie sie sich aus der Sicht des Kunden darlegen. Zudem werden qualitative und quantitative Eigenschaften, wie Verläßlichkeitsgrad, Umfang, Leistung, Einschränkungen und Bedingungen für die Korrektheit notiert, so weit sie sich schon erkennen lassen. Der Schritt kann iterativ für jede Systemfunktion wiederholt werden, indem diese weiter informell durch Unterfunktionen beschrieben wird.

15.5.2 Informationsanalyse

Zu jeder Funktion müssen die notwendigen Daten, mit denen sie in Beziehung steht, identifiziert und dokumentiert werden. Eine Funktion kann auf drei Informationsquellen zugreifen, die Eingabedaten $i_1...i_n$, die Ausgabedaten $o_1...o_m$ und persistente Daten (siehe Abbildung 15.4). Wie persistente Daten in Π realisiert werden können, zeigt ein Beispiel am Ende des Kapitels. Alle persistenten Daten eines Systems repräsentieren den Zustand des Systems. Schon in dieser Phase kann eine Zerlegung der Daten oder Funktionen notwendig werden, um sinnvolle Komponenten zu isolieren.

Die Beziehungen zwischen den Systemfunktionen untereinander ergeben sich aus der Analyse des Datenflusses im System. Dabei lassen sich Abhängigkeiten im Datenfluß zwischen zwei Funktionen erkennen, wenn ein Teil der Ausgabe der einen Funktion Eingabe der anderen ist. In der graphischen Darstellung werden die entsprechenden Aus-und Eingabepfeile der Funktionsnotationen zu einem zusammenhängenden Diagramm verbunden. Überlappende Ein- und Ausgabedaten können die Neudefinition von Daten durch Vereinigung oder Aufspaltung von Datenteilen sinnvoll machen. Die Beziehungen werden wie die Funktionen informell beschrieben.

15.5.3 Aufstellung der Integritätsbedingungen

Die Aufgabe besteht darin, die zwischen Funktionen und Daten erkannten Beziehungen durch strenge Integritätsbedingungen (z.B. zur Konsistenz) zu präzisieren (informell). In der *concurrency view* der Π-Spezifikation lassen sich die Bedingungen durch **path expression** und **precondition...of...is...** formalisieren.

15.5.4 Modularisierung

Der *Funktionen-Daten-Graph* des letzten Entwicklungsschrittes soll nun so in Partitionen (Cluster) von Funktionen aufgeteilt werden, daß jede Partition alle von ihr benutzten persistenten Daten enthält und die entsprechenden Integritätsbedingungen sicherstellen kann.

Die Cluster können als Vorformen der späteren Module angesehen werden, in denen die persistenten Daten den inneren Zustand des Moduls repräsentieren und die Import- und Export-Daten im Modul den offenen, das heißt nicht mit Funktionen innerhalb des Moduls in Beziehung stehenden, Ein- und Ausgabedaten der Modulfunktionen entsprechen. Jede Funktion wird genau einem Modul zugeordnet. Die persistenten Daten des Systems sollten sich so auf die Module aufteilen, daß keine zentralen Zustandskomponenten entstehen, auf die gleichzeitig von Funktionen aus unterschiedlichen Modulen zugegriffen wird. Ziel ist somit eine Partitionierung des Systemzustands, in der eine bestimmte Partition (Modul) genau einen bestimmten Teil des Zustands beinhaltet. Um dies zu erreichen, dienen zwei Entwurfstechniken, die Zerlegung (*decomposing*) von Funktionen und die Clusterbildung (*clustering*), als adäquate Hilfsmittel.

Die Zerlegung einer Funktion kann sich an den Daten der Funktion oder an der Bildung von Unterfunktionen orientieren, abhängig davon was natürlicher ist oder in Bezug auf die dezentrale Modularisierung sinnvoller erscheint. Bei der daten-orientierten Zerlegung werden erst die Daten einer Funktion geteilt und dann passende Unterfunktionen auf den Teilen definiert. Bei der funktionsorientierten Zerlegung ergeben sich die Unterfunktionen dagegen direkt aus natürlichen Unteraufgaben der Funktion. In beiden Fällen müssen die Beziehungen zwischen Funktionen bzw. zwischen Funktionen und Daten entsprechend neu definiert werden, was iterativ durch Wiederholung des zweiten Entwicklungsschrittes erfolgt.

Cluster auf diesen verfeinerten Funktionen und Daten entstehen aus der Vereinigung von persistenten Daten oder über gemeinsame persistente (Teil)daten. Durch Verschmelzung ähnlicher persistenter Daten(typen) zu einem neuen Datum und entsprechender Anpassung der abhängigen Funktionen (genauer ihrer Beschreibungen) kann ein Cluster gebildet werden. Identische persistente Daten motivieren die Clusterbildung natürlich direkt. Läßt sich in verschiedenen persistenten Daten(beschreibungen) eine gemeinsame Unterkomponente identifizieren (siehe Abbildung 15.5), kann diese extrahiert werden und ein selbständiges persistentes Datum bilden. Jede Funktion, die mit einem alten persistenten Datum in Beziehung stand, kann dann so in zwei Funktionen aufgespalten werden, daß eine Unterfunktion auf die gemeinsame Datenkomponente und die Restfunktion auf die alte Restkomponente des Datums zugreift. Zudem muß eine Beziehung zwischen beiden Funktionen beschrieben werden.

Entstehen bei der Modularisierung zu komplexe Module, kann der gesamte Entwicklungsprozeß iterativ auf ein Modul angewandt werden, was zu einer Verschachtelung von Modulen führt. Bei der Spezifikation mit Π kann diese durch Konfigurationen von Komponenten realisiert werden.

Das Ergebnis der iterativen Modularisierung und der komponenten- orientierten Entwicklung insgesamt ist eine Modulhierachie, in der nur azyklische *use*-Beziehungen zwischen Modulen existieren. Die Module der obersten Ebene sollten die grundlegenden Systemfunktionen wiederspiegeln.

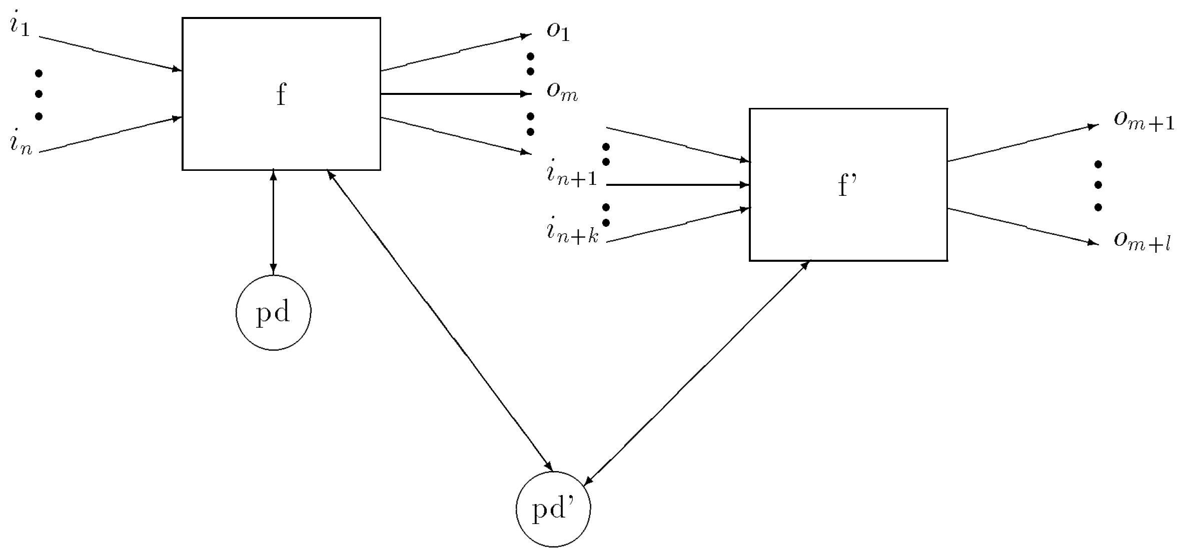


Abbildung 15.4: Graphische Notation

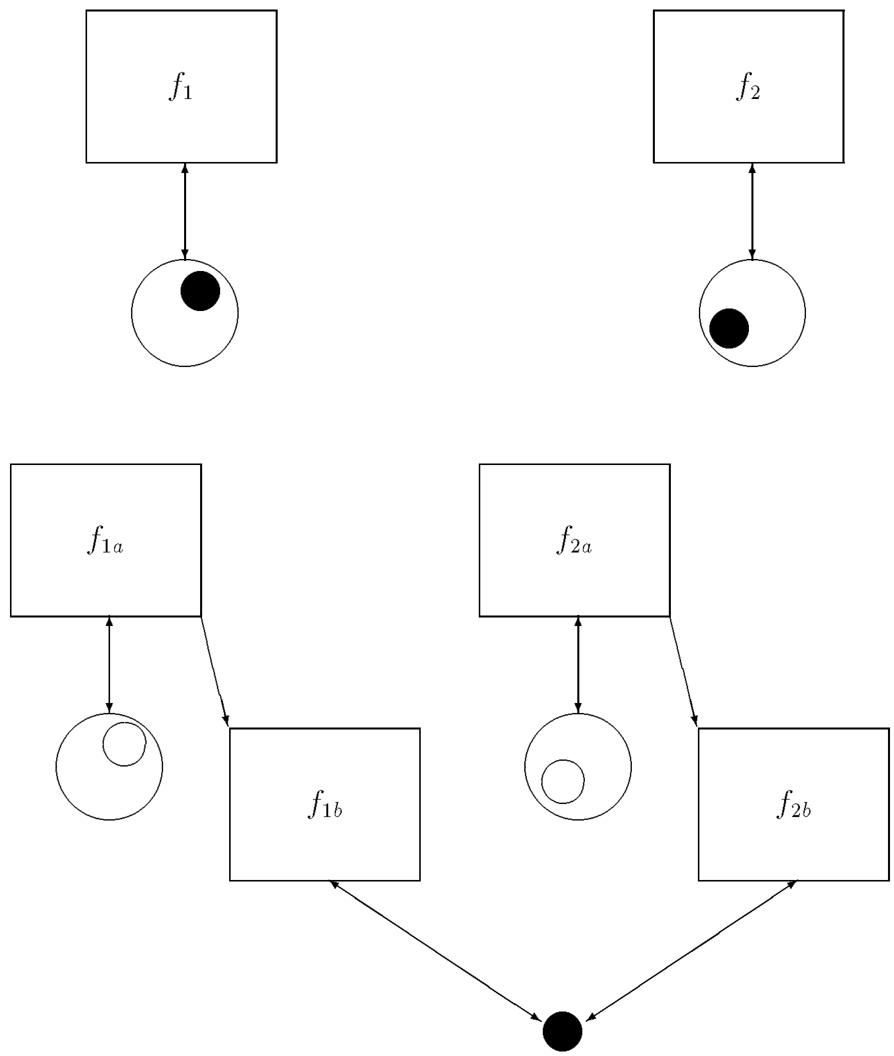


Abbildung 15.5: Clusterbildung

15.5.5 Spezifikation einer Π-Komponente für persistente Daten

```
cem persistent_data

  type view specification

    export

      type persistent_integer

        operation init_integer: range → persistent_integer
          description
            {Die Operation reserviert Speicherplatz für ein persistentes
            Integer-Objekt mit Wertebereich range.}

        operation is_persistent: persistent_integer → boolean
          description
            {Die Operation überprüft, ob das Integer-Objekt schon in
            database gespeichert ist.}

        operation read_integer: persistent_integer → persistent_integer
          description
            {Die Operation ließt den Inhalt des persistenten Integer-
            Objekts aus.}
          variables
            r: range
          equations
            read_integer (init_integer (r)) = init_integer (r)
            read_integer (persistent_integer) =
              if is_persistent (persistent_integer)
              then read_integer (persistent_integer)
              else read_integer (init_integer (r))

        operation write_integer: persistent_integer, value → persistent_integer
          description
            {Die Operation schreibt value in das persistente Integer-Objekt.}
          ...

    import

      type database
        ...
```

body

construction of type persistent_integer **is** database
...

end cem persistent_data

15.6 Einführung in PiLS (Dirk Niemann)

Einleitung

Die Sprache Π [SG94] erlaubt die Spezifikation von Software-Komponenten unter dem Aspekt verschiedener Views. Hieraus ergeben sich beinahe zwangsläufig Spezifikationen, die sehr umfangreich werden und in denen sich Teile der verschiedenen Views überschneiden.
Diese Eigenschaften von Π verlangen schon während der Spezifikation nach Werkzeugen, welche in der Lage sind, folgende Aufgaben zu übernehmen:

- Überprüfung von Syntax und Semantik der Spezifikation
- Überprüfung der Kompatibilität der verschiedenen Schnittstellen
- Berücksichtigung von redundanten Teilen der verschieden Views, d.h. die best mögliche Ableitung eines Views aus einem anderen
- Unterstützung der Erstellung von Benutztbeziehungen (Import-/Export-Interface) einzelner Komponenten
- Generierung von ausführbaren Programmen

Für diese Aufgaben wurde die Π-Entwicklungsumgebung PiLS (Pi-Language Syntax Directed Development Environment) [SG94] entwickelt.

Die Werkzeuge von PiLS

Bei den im folgenden genannten Werkzeugen handelt es sich nicht um alleinstehende Programme. Vielmehr sind die verschiedenen Werkzeuge in eine komplexe grafische Benutzeroberfläche integriert worden.

Der Library-Manager

Beim Starten von PiLS erscheint als erstes der Library-Manager (s. Abb. 15.6). Dessen Aufgabe ist es, die verschiedenen Konfigurationen und CEMs hierarchisch zu verwalten. Die momentane Struktur umfaßt Bibliotheken (Verzeichnisse), Konfigurationen und CEMs. Als Funktionalitäten werden alle Standard-Operationen (öffnen, löschen, usw.) angeboten. Außerdem werden von hier aus alle weiteren Werkzeuge gestartet.

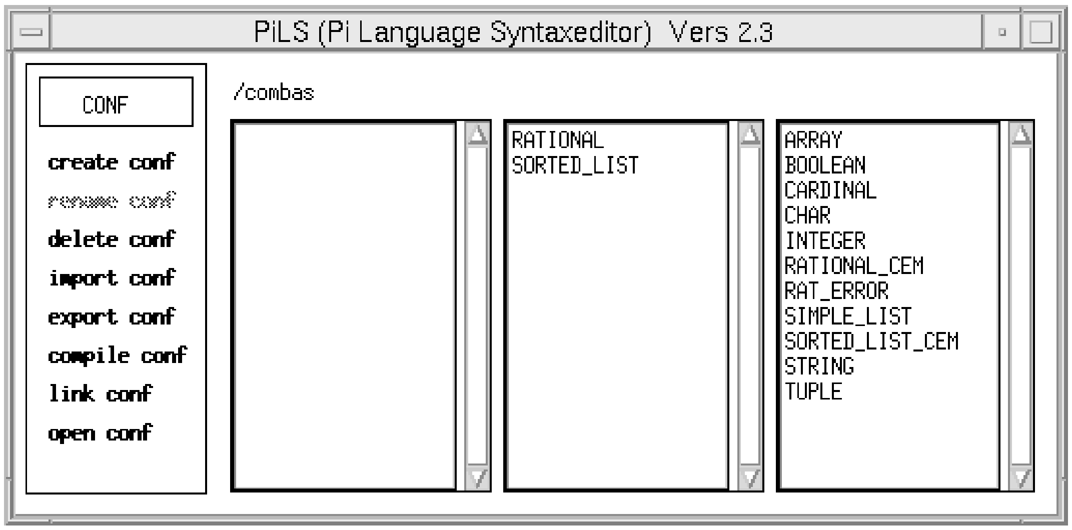


Abbildung 15.6: Das Fenster des Library-Managers

Um die Entwicklung von Spezifikationen innerhalb mehrerer Personen zu unterstützen, ist für die Zukunft die Integration von Funktionen zum Management von Zugriffsrechten und verschiedenen Versionen geplant.

Der CEM-Editor

Der CEM-Editor dient zur Spezifikation der Schnittstellen und der Body-Section von CEMs. Es findet hierbei eine Gliederung der Eingabe in Abschnitte, Datentyp und Operationen statt.

Ausgangspunkt ist das in Abb. 15.7 gezeigte Fenster. Von hier aus können die verschiedenen Abschnitte eines CEMs angewählt werden.

Nach der Auswahl einer Sektion kann nun die gewünschte Operation ausgewählt werden (s. Abb. 15.8). Die Eingabe kann entweder im syntax-gesteuerten Modus (s. Abb. 15.9) oder im einfachen Textmodus (s. Abb. 15.10) geschehen. Ein Wechsel zwischen den beiden Modi kann jederzeit erfolgen - auch wenn Teile der Spezifikation noch nicht korrekt sind. Um die schrittweise Implementierung zu unterstützen ist es sogar notwendig, solche inkorrekten Eingaben zu akzeptieren. Eine Hervorhebung durch kursive Schrift weist dabei den Benutzer dennoch auf eventuelle Fehler hin.

Weiterhin bietet der CEM-Editor die Möglichkeit, durch Ableitungen bestehende Spe-

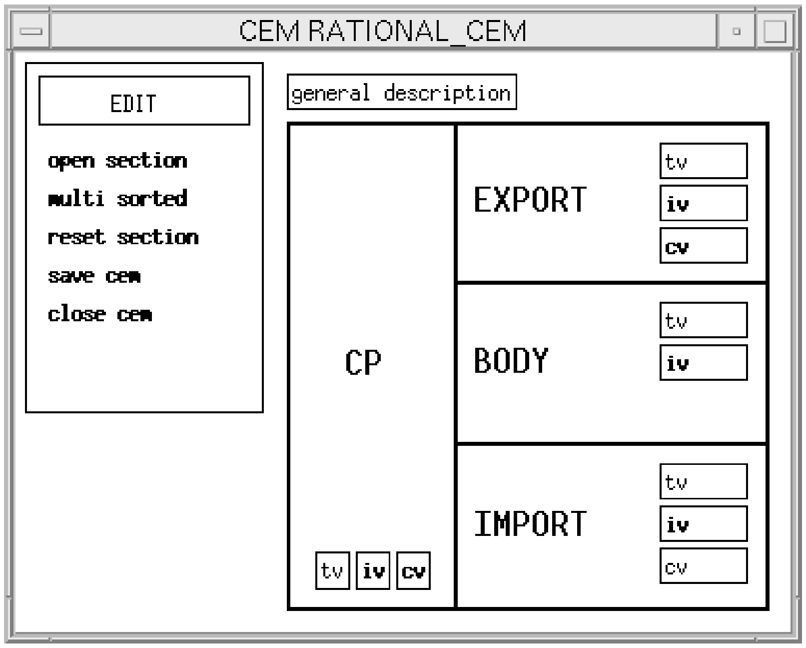


Abbildung 15.7: Das CEM-Fenster

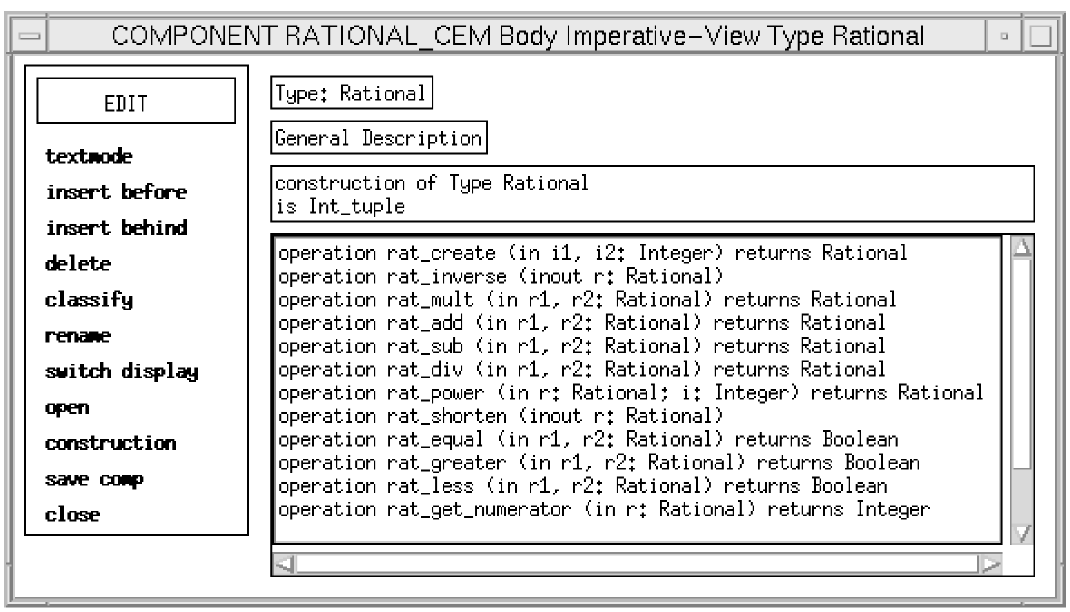


Abbildung 15.8: Auswahl der Operationen eines CEM

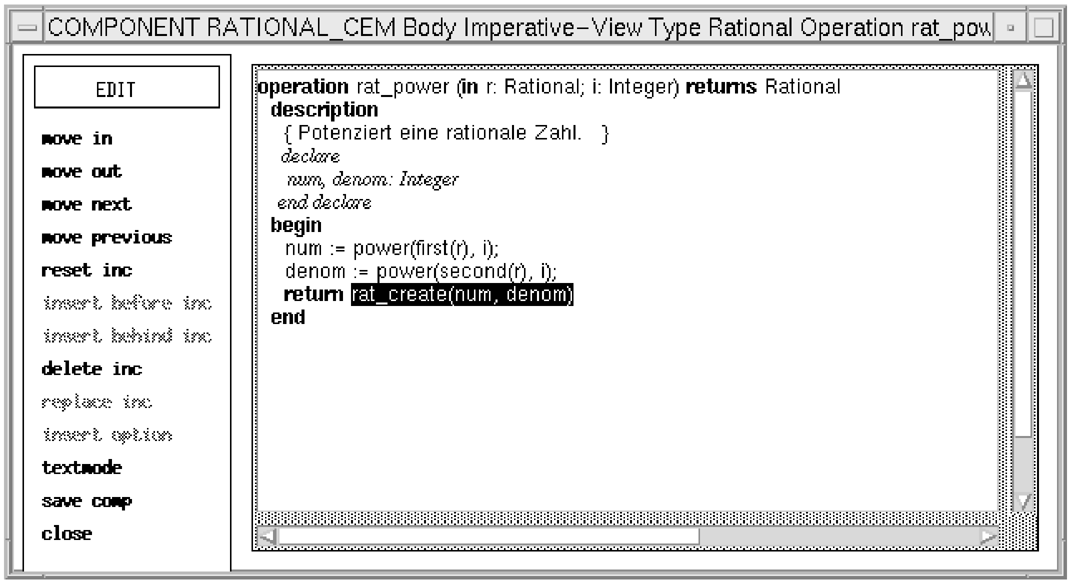


Abbildung 15.9: Editieren im syntax-gesteuerten Modus

zifikationen zu erweitern. So kann man z.B. innerhalb eines Views aus der Body-Section die Export-Section oder eine Sektion eines Views in die eines anderen überführen.

Der Architektur-Editor

Der Architektur-Editor (s. Abb. 15.11) bietet die komfortable Möglichkeit, Konfigurationen und ihre Benutztbeziehungen mit Hilfe grafischer Symbole zu spezifizieren. Auch der Architektur-Editor unterstützt die inkrementelle Entwicklung. Mit Hilfe von Boxen und initialen Angaben über deren Funktion kann zunächst die Struktur einer Konfiguration festgelegt werden. Später können dann diese Boxen zu CEMs oder Konfigurationen mit detaillierten Spezifikationen erweitert werden.

Der Compatibility-Checker

Obwohl er nicht als eigenständiges Werkzeug zu erkennen ist, bietet der Compatibility-Checker eine wichtige Funktionalität. Seine Aufgabe ist es zu überprüfen, ob zwei Schnittstellen kompatibel zueinander sind. Hierbei unterscheidet er zwischen inkompatibel, syntaktisch kompatibel und semantisch kompatibel. Der Compatibility-Checker kann also zum einen dazu benutzt werden, die Richtigkeit von Schnittstellen zu überprüfen, zum anderen könnte er aus einer bestehenden Bibliothek potentielle Kandidaten für Komponenten mit passenden Schnittstellen liefern.

Der Π-Compiler

Der Π-Compiler generiert aus einer Komponente zunächst C-Quelltext und anschlie-

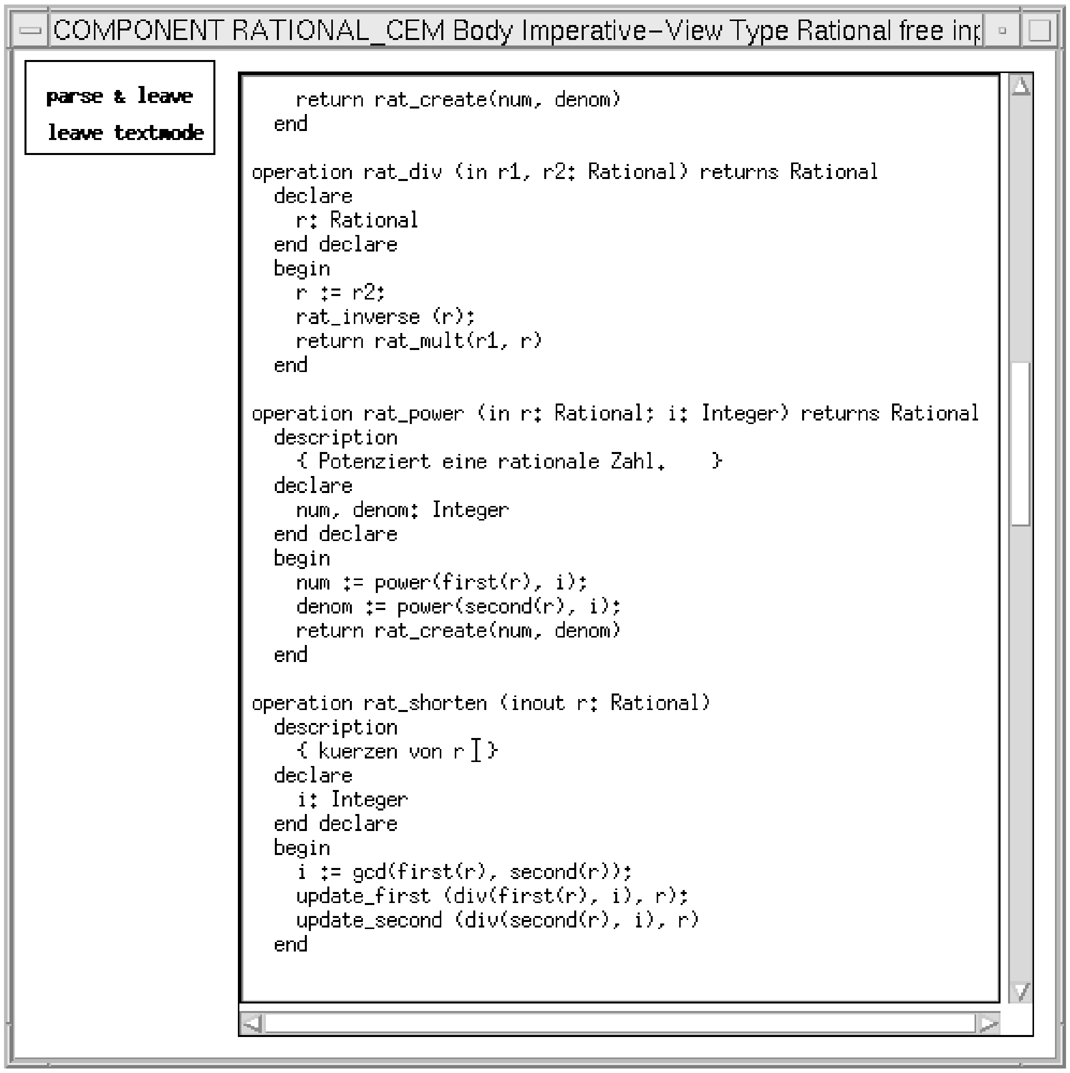


Abbildung 15.10: Editieren durch einfache Texteingabe

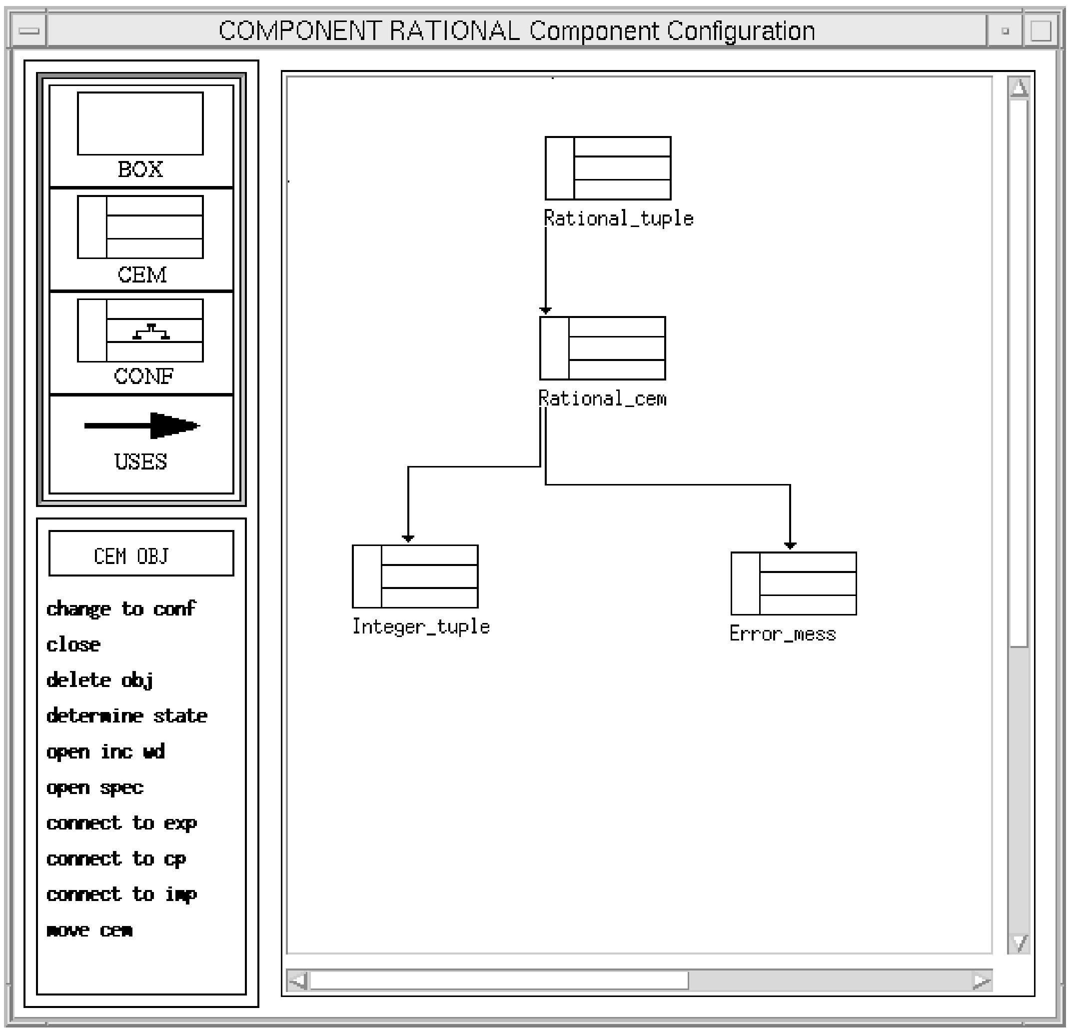


Abbildung 15.11: Der Architektur-Editor

ßend ein ausführbares Programm. In der momentanen Version werden Spezifikationen im Imperative-View unterstützt. Optional können auch Teile des Concurrency-View integriert werden, die dann bei Laufzeit entsprechende Überprüfungen vornehmen.
Um auch einzelne CEMs oder auch noch nicht fertige Spezifikationen übersetzen zu können, ist es möglich eine Umgebung hinzuzubinden, die zur Laufzeit alle unbekannten Werte abfragt (s. Abb. 15.12).

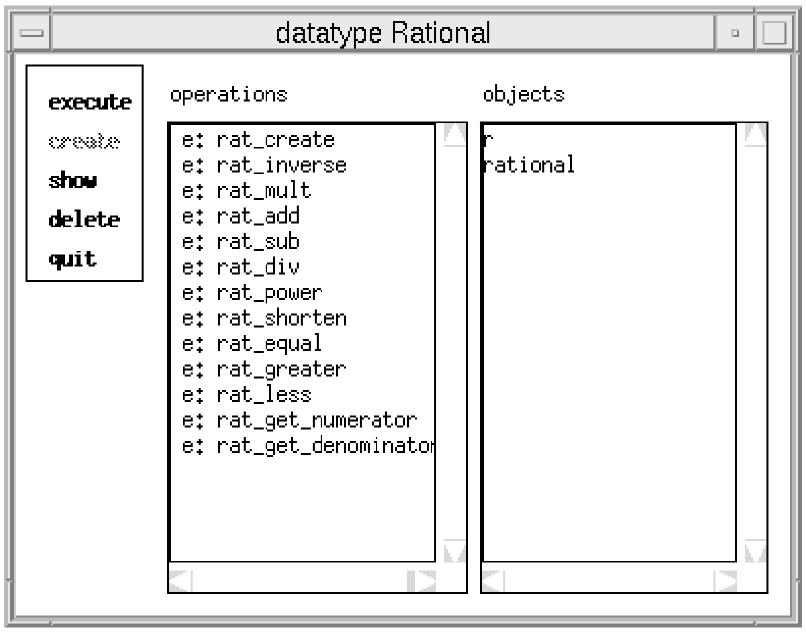


Abbildung 15.12: Eingabe von zur Laufzeit unbekannter Werte

Schlußwort

Mit dem Werkzeug PiLS wird dem Benutzer ein leistungsfähiges Werkzeug zur Spezifikation in Π zur Verfügung gestellt. Allerdings ist deutlich zu spüren, daß sich das Programm noch in der Entwicklung befindet. So sind zahlreiche Funktionen noch nicht implementiert und beim willkürlichen Anklicken verschiedener Buttons ist ein Programmabsturz nicht unwahrscheinlich. Weiterhin sorgt die Arbeitsgeschwindigkeit teilweise für unfreiwillige Pausen. Ein weiteres Manko ist außerdem das Fehlen von Standardbibliotheken.

15.7 Formale Spezifikation einer Softwareentwicklungsumgebung (Jörn Bodemann)

Mit "Software Engineering" bezeichnet man das Erstellen von Software, die gewissen Gütekriterien genügt. Mit "Software Engeneering Environmet" (SEU) bezeichnet man ein System, das die Erstellung von Software erleichtert.

Ein SEU arbeitet mit Informationen über:

1. Die zu entwickelnde Software (z.B. Spezifikation, Quellcode, Testdaten...).
2. Dem Projekt zur Verfügung stehenden Resourcen (z.B. Mitarbeiter, Pflichtenverteilung...).
3. Allgemeine und firmenspezifische Richtlinen, die den Softwareentwicklungsprozeß koordinieren helfen sollen.

Jede SEU besteht aus zwei Teilen, wobei der erste Teil den Softwarelifecycle selbst unterstüzt (TOOLs) und der zweite Teil die Kommunikationsstruktur unter den verschiedenen Tools, sowie die Graphische Oberfläche und weitere administrative Fähigkeiten zur Verfügung stellt (FRAMEWORK) [NIS91]. Ein Framework stellt also Kommunikationsstrukturen und andere Leistungen zur Verfügung, die die Erzeugung von Tools erleichtern und vereinheitlichen. Durch ein Framework (FW) werden plattformunabhängige Tools überhaupt erst ermöglicht.

Ein FW stellt demgemäß bestimmte Leistungen zur Verfügung. Diese werden Services oder Leistungen genannt, und ihrer Funktionalität nach in verschiedene Gruppen aufgeteilt. Diese Gruppen sind nicht wie Layers in hierarchische Teile aufgeteilt, sondern stehen mit einer Ausnahme gleichberechtigt nebeneinander.

Die Funktionalität der SEU wird also nur durch TOOLs erweitert, die z.B. zur Kommunikation untereinander eine Leistung des FW nutzen, nämlich den Communication Service.

Die Leistungen eines FW werden durch folgende sechs Servicegruppen erbracht:

1. Object Management Service: übernimmt das Laden, Speichern usw. von Objekten und definiert Beziehungen dieser untereinander.
2. Process Management Service: Unterstützt die Entwicklung während des gesamten Softwarelifecycles.
3. Communication Service: Bietet die Strukturen für die inter-TOOL und die inter-Service Kommunikation, z.B. MessagePassing, RPC, Data-sharing.
4. User Interface Service: Stellt eine einheitliche Benutzeroberfläche zur Verfügung, die alle TOOLs für die Interaktion mit dem Benutzer verwenden.

5. Policy Enforcement Services: Sicherheits-, als auch Integritätsbedingungen können durch diesen Service realisiert werden.

6. Framework Administration and Configuration Services: Bietet administrative Funktionen, wie zum Beispiel das Integrieren neuer TOOLs in das FW.

Somit sind alle Leistungen des FW in sechs Gruppen aufgeteilt. Um jedoch verschiedene FW miteinander vergleichbar zu machen, werden die Leistungen zusätzlich in sogenannte Dimensionen aufgeteilt. Der Begriff Dimension soll andeuten, daß diese untereinander unabhängig sind, und zwar soweit, das eine Änderung einer Eigenschaft eines Service in einer Dimension nicht eine Änderung des Service in einer anderen Dimension nach sich zieht.

Die Vorteile einer Strukturierung der Leistungen in Dimensionen sind:

1. Wichtige Eigenschaften der einzelnen Leistungen werden herausgearbeitet.

2. Die Beschreibungen der einzelnen Leistungen sind in ihrer Art konsistent.

Für die Einteilung der Leistungen werden folgende Dimensionen vorgegeben:

1. Conceptual: Beschreibt die Funktionalität einer Leistung ohne jedoch auf Implementation oder Interface einzugehen.

2. Operations: Beschreibt die Menge der Operationen, die zur Verfügung gestellt werden.

3. Rules: An die Objekte und Funktionen eines Services sind oftmals Bedingungen wie etwa Pre- oder Postconditions, Restriktionen usw. geknüpft. Diese werden hier beschrieben.

4. Types: Beschreibt das benutzte Datenmodell dieses Service.

5. External: Beschreibt, wie der Service von Nutzern angesprochen wird. Das kann z.B. ein Procedurinterface oder eine Abfragespache sein.

6. Internal: Hier werden Implementationsdetails diskutiert.

7. Related Services: Wie ein Service mit anderen interagiert wird hier beschreiben, sowie in statische und dynamische Interaktion unterschieden.

Nicht jedes System, das durch dieses Referenz Modell beschrieben wird, muß alle Services anbieten.

Somit steht ein RM zur Verfügung, das einen Anforderungskatalog für eine SEU liefert. Der zweite Schritt auf dem Weg zu einer Implementation einer SEU ist nun, aus diesem Anforderungskatalog eine Spezifikation zu entwickeln.

Diese Spezifikation soll nun durch die Kombination des ER-Formalismus und der Spezifikationssprache Π erfolgen.

Produkte und Prozesse der SEU werden mittels eines ER-Formalismus spezifiziert, der um einen Objektansatz erweitert ist. Bei diesem wird wie bei dem allgemeinen ER-Ansatz ein Objekt (vormals Entität) durch seinen Typ klassifiziert.
Die statische Struktur eines Objekts im erweiterten ER-Ansatz wird durch die enthaltenen Attribute festgelegt. Das Objekte selbst hat keinen darstellbaren Typ. Insoweit unterscheiden sich das erweiterte- und das Standard-ER-Modell nicht.
Zusätzlich werden nun erweiterte Beziehungstypen eingeführt.

1. IS-PART-OF,
2. IS-A,
3. USES

Weitere Datentypen kommen nicht hinzu.
Die benutzten Abstraktionskonzepte sind also :

- Klassifikation: Objekte werden durch Objekttypen klassifiziert.
- Beziehungstypen: IS-PART-OF, IS-A, USES.

Die benutzten Datentypen sind:

- Standardtypen: INTEGER, STRING.
- Strukturierte Datentypen: Mengen, Tupel, Aufzählung.
- ADT: durch Angabe der Operationen.

Die Ergebnisse der Spezifikationen mittels des ER-Modells sollen nun als Eingabe für die Spezifikationssprache II benutzt werden.
Jede modellierte Entität wird durch eine II-Komponente spezifiziert.

16. Evaluierung der Strategien

Im folgenden findet sich eine Bewertung der verschiedenen Algorithmen, die wir im Rahmen der Planung implementiert haben.

Wie zu erwarten war, stellen die Strategien *Schwachsinn* und *Kurzsichtig* keine ernstzunehmenden Gegner dar.

Der *Distanz* Algorithmus bietet dem Spieler die Möglichkeit erste Erfahrungen mit dem Computer zu machen, ohne die Lust am Spiel durch allzu schlecht spielende Detektive abgeschreckt zu werden.

Distanz-Summe stellt die höchsten Anforderungen an das Spielgeschick des menschlichen Gegners. Die Detektive spielen zwischen dem dritten und dem dreizehnten Zug stark genug, um auch einen erfahrenen Spieler in Bedrängnis zu bringen. Danach macht der sinkende Ticketvorrat den Detektiven zu schaffen. Es stellt sich herraus, daß der Vorrat nicht genügend in die Bewertung einfließt. Nutzt man die Möglichkeit, den Detektiven mehr als die in den Spielregeln vorgesehenen Tickets für das Spiel zur Verfügung zu stellen, bleibt das Spiel auch über den 15. Zug hinaus sehr kniffelig. Der Algorithmus spielt so gut, daß es Mister X häufig nur durch den Einsatz eines Doppelzuges gelingt, aus der Gefahrenzone zu entkommen. Auch durch den Einsatz von Blacktickets kann der menschliche Spieler die Detektive stark verwirren. Ein gezielter Einsatz eines solchen Tickets, z.B. nach der Fahrt mit der U-Bahn, läßt die Zahl der möglichen Positionen, an denen sich Mister X befinden kann, so stark in die Höhe schnellen, daß kein koordinierter Spielzug mehr entsteht. Sobald sich Mister X wieder zeigen muß, gelangen die Detektive wieder zur gewohnten Stärke. Dieses Verhalten deckt sich in der Tendenz mit den Erfahrungen, die wir beim intensiven Spielen gegeneinander in der ersten Seminarphase gemacht haben.

Obwohl der *Referenz* Algorithmus einen Zug voraus berechnet, und somit stärker spielen sollte, als *Distanz-Summe* ist dies in der Praxis nicht der Fall. Wir wissen nicht warum.

Mix ist kein eigentlicher Algorithmus, sondern hat das Ziel, dem menschlichen Gegner die Chance zu nehmen, gezielt das Wissen über das Verhalten der Algorithmen auszunutzen. Dies geschieht, indem vor jedem Zug zufällig bestimmt wird, welcher Algorithmus als nächstes verwendet wird.

17. Kritik und Vorschläge zur Weiterentwicklung von ProSet

17.1 Einführung

Wir haben uns kritisch mit der Sprachdefinition von ProSet auseinander gesetzt. Dabei gingen insbesondere die Erfahrungen, die während der Implementierung des Programms ScotlandYard gewonnen wurden, ein. Man muß allerdings berücksichtigen, daß einige der Probleme während der Implementierung nicht vom Konzept sondern durch die erste Realisierung des ProSet–Compilers und der Laufzeitbibliotheken ausgelöst wurden und deshalb in diesem Dokument nicht diskutiert werden.

Die Kritik wird in fünf verschiedene Abschnitte geteilt, die weitestgehend abgeschlossene Einheiten sind. Es werden Module, ProSet–Linda, das Typsystem, die Syntax sowie die Semantik diskutiert.

17.2 Module

17.2.1 Geteilte Modulvariablen *(shared variables)*

In ProSet ist jede Instanz eines Moduls ein eigenständiges Objekt, das keine Seiteneffekte zuläßt, da globale Variablen in Modulen nicht unterstützt werden und importierte Variablen nur call-by-value– bzw. rd–Parametereigenschaften haben. Das bringt aber den Nachteil mit sich, daß zwischen verschiedenen Instanzen eines Moduls keine Kommunikation über gemeinsame *(geteilte)* Variablen möglich ist, schließt man einmal den verhältnismäßig langsameren Datenaustausch über Tupelräume aus.

Sinnvoll wäre deshalb die Einführung einer Notation, die das Teilen bestimmter lokaler Variablen (Modulattribute) zwischen allen Instanzen eines Moduls - ähnlich den Klassenvariablen *(static variables)* in C++ - ermöglicht. Gerade in einer Prototyping–Sprache wie ProSet, die parallele Programmierung unterstützt, kann die Datenteilung zwischen Modulinstanzen zur Erhöhung der Ausdrucksfähigkeit der Sprache beitragen. Zudem verhindert dieses Konzept Inkonsistenzen, da nicht jede Modulinstanz die lokale Variable selbstständig aktualisieren muß. Geteilte Modulvariablen lassen sich zum Beispiel durch folgende Syntax beschreiben:

shared visible <Modulvariable>.

Das Teilen aller Modulattribute macht allerdings wenig Sinn, weil dadurch eine mehrfache Instanziierung des Moduls überflüssig wird. Die Instanzen eines Moduls unterscheiden sich ja gerade durch unterschiedliche Attributwerte, die verschiedene Zustände wiederspiegeln. Somit bliebe nur die Möglichkeit, bei der Instanziierung unterschiedliche Werte zu importieren, um Instanzen zu erzeugen, die zumindest in ihrer Initialisierung differieren.

Wäre ProSet eine rein sequentielle Programmiersprache, wären keine gleichzeitigen Zugriffe über zwei Modulinstanzen auf geteilte Attribute möglich. Parallele Programmierung erfordert dagegen zusätzlich einen Mechanismus, der gegenseitigen Ausschluß *(mutual exclusion)* bei konkurrierenden Zugriffen auf geteilte Daten sicherstellt. Dies kann durch Mechanismen wie Semaphore oder Monitore realisiert werden, was hier aber nicht näher diskutiert werden soll. Das grundsätzliche Problem geteilter Variablen besteht darin, daß bei zwei nahezu gleichzeitigen parallelen Schreibzugriffen, ein Update, nämlich das erste, überschrieben wird. Nach dieser Semantik liefert ein Lesezugriff genau den Wert des letzten Schreibzugriffs auf die Variable.

Gegenseitiger Ausschluß allein garantiert aber noch nicht die Korrektheit paralleler Zugriffe:

Wenn zwei Module M und M' das gemeinsame Datum x teilen, könnte ein Zugriff über eine Prozedur aus M' den Zustand von x so verändern, daß eine Prozedur aus M das Datum x nicht mehr verändern dürfte.

Zur Lösung diese Problems müssen entsprechende Vorbedingungen vor dem Aufruf von Modulprozeduren überprüft werden, die auf das geteilte Datum zugreifen. Dies kann entweder modulextern in Form von

if <Bedingung> **then** <Modul>.<Prozeduraufruf>

bei jedem Prozeduraufruf explizit erfolgen oder implizit im Modul garantiert werden, was sich z.B. bei der Definition der Modulprozedurn durch ein Schlüsselwort **requires**, auf das die Vorbedingung folgt, angeben läßt:

procedure <Modulprozedur>(...) **requires** <Vorbedingung>

Die Vorbedingung wird automatisch beim Aufruf der Prozedur überprüft. Ist das Ergebnis der Auswertung wahr, wird die Prozedur unter Berücksichtigung des gegenseitigen Ausschlusses ausgeführt. Andernfalls wird der aufrufende Prozeß suspendiert, bis die Bedingung erfüllt ist.

Geteilte Variablen nur innerhalb eines Moduls zu erlauben, hat den Vorteil, daß alle Operationen zur Manipulation dieser Daten wohldefiniert sind. Dies sichert die Einhaltung der durch die Vorbedingungen definierten Integritätsbedingungen und erhält die Konsistenz der Daten.

17.2.2 Geteilte Modulinstanzen *(Monitore)*

In PROSET sind Module eigenständige Datentypen, deren Werte die Instanzen bilden. Wenn in einer parallelen Umgebung mehrere Prozesse gleichzeitig Zugriff auf eine Modulinstanz haben sollen, müssen wieder gegenseitiger Aussschluß und die Konsistenz der Dateninhalte gewährleistet werden. Dazu ist die Einführung einer speziellen Moduldefinition notwendig:

shared module <ID>
...
end <ID>.

Die Instanzen dieses Modultyps haben Monitoreigenschaften, d.h. sie garantieren zum einen, daß nur ein Prozeß gleichzeitig Operationen in einer Modulinstanz ausführen kann. Prozesse, die auf eine bereits „besetzte" Instanz zugreifen wollen, werden suspendiert. Zum anderen müssen wieder implizit die Vorbedingungen für die Ausführung einer Monitoroperation vor jedem Zugriff getestet werden.

Die Eigenschaften lassen sich über entsprechende Betriebssystemaufrufe implementieren. Zum Beispiel kann gegenseitiger Ausschluß durch Semaphore realisiert werden.

Bei Monitoren können im allgemeinen dann Deadlocks auftreten, wenn ein Prozeß aus einem Monitor heraus, einen anderen Monitor betreten darf *(nested monitor calls)*:

Zum Beispiel besteht dann die Möglichkeit, daß zwei Monitorprozesse gleichzeitig auf den Eintritt in den Monitor des jeweils anderen Prozesses warten.

17.2.3 Überladen von (Modul–)Prozeduren

Aufgrund der Instanziierung von Modultypen *(Schablonen)* und der schwachen Typisierung sind Module in PROSET schon von Natur aus generisch. Die für generische Module typische Übergabe von Typparameteren *(typparametrisierte Module)* kann entfallen. Zudem lassen sich den (gleichen) importierten Variablennamen sowie den Parametern der Modulprozeduren verschiedene Werte übergeben. Weil zur Zeit der Compilierung keine Typprüfung erfolgt, wird erst zur Laufzeit die eventuelle Inkompatiblität zwischen einem übergeben Wert und der Implementierung einer Prozedur festgestellt.

Der Programmierer hat nur die Möglichkeit durch explizite Typabfragen mit Hilfe der Standardoperation **type** und entsprechenden Fallunterscheidungen generische Prozeduren zu erzeugen. Einfacher wäre es hier, wenn die Möglichkeit bestände, einen Prozedurnamen dahingehend zu überladen, daß mehrere Prozeduren gleichen Namens aber mit unterschiedlichen Prozedurköpfen und Implementierungen zugelassen sind. Der Compiler sollte dann einen Prozeduraufruf automatisch der richtigen Implementierung zuordnen können, was wiederum nur mit einer optionalen Möglichkeit zur Angabe von Parametertypen *(optionale Typisierung)* möglich ist.

17.2.4 Zusicherungen *(assertions)*

Um die Konsistenz eines Moduls sicherzustellen und dessen Wiederverwendbarkeit zu erhöhen, bieten sich Zusicherungen in Form von Modulinvarianten sowie Vor– und Nachbedingungen für Modulprozeduren an. Der Programmierer ist damit in der Lage Axiome und Integritätsbedingungen direkt in das Modul zu integrieren.

- Vorbedingungen drücken aus, was vor der korrekten Ausführung einer Modulprozedur erfüllt sein muß, und könnten, wie oben vorgeschlagen, durch das Schlüsselwort **requires** eingeleitet werden.

- Nachbedingungen sind nach Ausführung einer Prozedur erfüllt und sollten wie die Vorbedingungen optional hinter dem Prozedurkopf stehen können. Hierzu bietet sich in Analogie das Schlüsselwort **implies** an:

  procedure <Modulprozedur>(...)
  requires <Vorbedingung>
  implies <Nachbedingung>.

- Modulinvarianten müssen von allen Instanzen zu jedem Zeitpunkt garantiert werden und geben die Möglichkeit allgemeine Konsistenzbedingungen, die vor und nach Ausführung jeder Modulprozedur gelten müssen, zu beschreiben. Dazu ist eine EIFFEL–ähnliche invariant–Bedingung am Ende des Moduls geeignet.

Wenn eine der genannten Bedingungen nicht erfüllt ist, sollte eine Ausnahmebehandlung *(exception)* erfolgen.

17.2.5 Ein Beispiel

```
shared module buffer
  import BUF_SIZE, ...
  export put, get, ...
begin
  visible n = 0;    -- Anzahl Elemente im Puffer
  visible buf = []; -- Puffer

  procedure put(x) ... requires n < BUF_SIZE ...
  procedure get(x) ... requires n > 0        ...
invariant
  n => 0 and n <= BUF_SIZE;
end buffer;
```

17.2.6 Die Schnittstelle der Module

Ein Problem der schwachen Typisierung ist, daß sowohl bei Prozedurdefinitionen keine Angaben über die Parametertypen gemacht werden als auch in den Modulschnittstellen lediglich die Variablennamen der importierten Wete und exportierten Prozeduren aufgeführt sind. Damit lassen sich Module und Prozeduren nicht ohne Kenntnis ihrer Implementationen benutzen, was die Wiederverwendbarkeit und Wartbarkeit von ProSet–Code erschwert. Ohne eine detailierte Dokumentation der Modulschnittstelle, ist der Benutzer gezwungen, die Syntax der von dem Modul angebotenen Prozeduraufrufe aus dem Quelltext zu entnehmen, was bei komplexen Modulen wohl eher unzumutbar sein dürfte.

Das letzte Problem ließe sich durch die obligatorische Angabe der genauen Prozedurköpfe (statt nur der Prozedurvariablen) der aus einem Modul exportierten und in ein Modul importierten Prozeduren lösen.

17.3 Linda

17.3.1 Templates

Die Tupel, über die Linda die Kommunikation realisiert, besitzen für den Compiler keine Semantik. Es kann nicht geprüft werden, ob ein Tupel im Tupelraum vorliegen kann, den man gerade versucht, zu matchen. Damit sind Deadlocks durch einfache Tippfehler Tür und Tor geöffnet. Als Lösung bietet sich an, ein Typ–Konzept für Linda–Tupel einzuführen. Wir nennen es *Templates*, die man ähnlich wie Records in anderen Sprachen definieren kann. Innerhalb eines Templates können die Anzahl der Elemente, konstante Ausdrücke und Typen der variablen Bestandteile angegeben werden. So kann der Compiler beim Aufru einer Linda–Operation die Anzahl der Parameter und die Typen überprüfen. Wenn sich nur die Werte der konstanten Bestandteile geändert hat, braucht man dies nur an einer Stelle im Quelltext zu tun. Damit wird die Wartbarkeit des Quelltextes erheblich verbessert.

Syntax

Die syntaktische Struktur von Templates sieht aus wie ein Record in Pascal–orientierten Sprachen. Signifikant ist sowohl die Reihenfolge der Elemente als auch ihr Typ und ihr Verhalten während des Matchings von Linda–Tupeln (*Charakteristik*) und optional ein hier vereinbarter Wert. Letzteres ist zwingend notwendig für Elemente der Charakteristik `const`.

⟨*template–def*⟩ ::= ⟨*identifier*⟩ `'='` `'template'` ⟨*members*⟩ `'end'` `'template'`.

⟨*members*⟩ ::= ⟨*member*⟩ [';'] | ⟨*member*⟩ `';'` ⟨*members*⟩.

⟨*member*⟩ ::= ⟨*match*⟩ ⟨*identifier*⟩ [`':'` ⟨*type*⟩] [`':='` ⟨*expression*⟩].

⟨*match*⟩ ::= `'const'` | `'actual'` | `'formal'`

Semantik

Die Semantik einer Template–Deklaration und einer Template–Benutzung basiert auf dem Matching–Verhalten des Templates. Es folgt eine Auflistung der Matching–Eigenschaften von Templates in Abhängigkeit der Charakteristik:

const Der Wert des Elementes wird bei der Deklaration festgelegt. Er kann im laufenden Programm nicht geändert werden. Es ist sinnvoll, einen schreibenden Zugriff als Fehler zu bezeichnen. Elemente mit dieser Eigenschaft dienen in erster Linie dazu, den Tupel zu charakterisieren und entsprechen damit der gängigen Praxis, den Tupeln einen konstanten Namen zu vergeben.

actual Der Wert des Elementes wird beim Matchen nur gelesen. Er kann aber zur Laufzeit des Programmes explizit gesetzt werden. Diese Charakteristik dient dazu, aus einer Menge von möglichen Tupeln eines mit einem bestimmten Wert zu extrahieren.

formal Der aktuelle Wert des Elementes wird beim Matchen ignoriert, der Wert nach dem Matching entspricht dem des Tupels aus dem Tupelraum. Dies entspricht einem *lvalue*, wie er normalerweise durch ein Fragezeichen in Linda bezeichnet wird.

Eine weitere wesentliche Eigenschaft für das Matching–Verhalten ist der Typ der Elemente. Wenn bei der Definition des Templates explizit ein Typ für ein Element angegeben wurde, dann wird dies beim Matching berücksichtigt. Damit kann die Kommunikation mit Linda und den Templates typsicher gestaltet werden.

Im Typsystem stehen Templates an einer exponierten Stelle, weil sie nicht zur Definition von anderen Typen benutzt werden können. Das beinhaltet auch, daß Templates selber nicht aus Templates zusammengesetzt werden können. Die Begründung hierfür ist, daß das Matching–Verhalten für andere Typen nicht definiert ist, und somit nicht klar ist, was damit gemeint sein könnte. Die Elemente eines Templates selber können aber aus einem beliebigen anderen Typ bestehen.

Beispiele

Das Tupel `Detective_Position` aus dem ScotlandYard–Pogramm kann mit dieses Templates beschrieben werden:

```
detective_position = template
                       const name: string = 'detective_position';
                       actual nr : integer;
                       formal pos : integer;
                     end template;
```

Die Benutzung dieses Templates für Detektiv 2 könnte z.B. so geschehen:

```
var dp : detective_position;
    my_pos : integer;
begin
   dp.nr := 2;
   fetch dp at Communication_TS end fetch;
   my_pos := dp.pos;
end;
```

Um mit Hilfe von Templates eine generische Client–Server–Kommunikation zu gestalten, kann man ausnutzen, daß das Matching–Verhalten verschiedener Templates gleich sein kann. So würde etwa ein Server keine Typen festlegen (sofern man nicht die xor–Verknüpfung beim Fetchen verwenden möchte):

```
generic_request = template
                    formal message : string;
                    formal id : atom;
                    formal content;
                  end template;


generic_answer = template
                   actual id : atom;
                   formal content;
                 end template;
```

Auf der Client-Seite würde man dies exakter festlegen und insbesondere ein Format für die Antworten spezifizieren.

```
my_request = template
               const  message : 'shortest_path';
               actual id      : atom;
               actual content : record
                                  from, to : integer;
                                end record;
             end template;

my_answer = template
              actual id : atom;
              formal path : tuple of integer;
            end template;
```

Aus diesem letztem Beispiel läßt sich eine möglich Erweiterung schon absehen: Man könnte ein Subtyp– oder objekt–orientiertes Konzept einführen, bei dem allgemein gehaltene Templates näher spezifiziert werden, aber so, daß die allgemeinen Templates alle spezielleren Templates matchen können, so wie dies hier bei den beiden Request– und Answer–Tupeln geschehen ist. Auf diese Weise können eine Vielzahl von ähnlichen Tupeln von einer Quelle abgeleitet werden. Ein Beispiel ist die Definition des `shortest_path`–Request:

```
my_request = template like generic_request
               redefine content as
                 actual content : record
                                    from, to : integer;
                                  end record;
             end template;
```

17.3.2 Höhere Kommunikationskonzepte

In Linda ist die Kommunkation zwar sehr einfach und abstrakt von jeder parallelen Hardware, aber es werden keine höheren Konzepte der Kommunikation realisiert. Weder werden klassische Synchronisationsverfahren wie Monitore angeboten, noch ein Message–Send und –Receive. Diese müssen explizit nachprogrammiert werden. Es wäre sehr angenehm, wenn entsprechende Libraries zur Verfügung gestellt werden würden.

17.3.3 Spezifikation

Die Entwicklung paralleler Programme ist sehr komplex, weil durch die parallelen Zweige des Programmes und deren Synchronisation und Kommunkation eine weitere Größenordnung der Komplexität hinzukommt. Durch den Einsatz von Linda ist man in der glücklichen Lage, von der Hardware und Betriebssystemsoftware abstrahieren zu können. Dies stellt eine wesentliche Vereinfachung dar. Dennoch bleibt die Komplexität bestehen. Angenehm wäre eine Spezifikationsmöglichkeit für die parallelen Konzepte, die zum Beispiel Analysen über das Deadlock–Verhalten des Programmes erlaubt. Eine Möglichkeit wären etwa Petrinetze. Im Sinne des Prototypings angenehm wäre dann weiterhin eine Coderahmen–Generierung, um so das gewünschte und modelierte Verhalten korrekt in das Programm abzubilden.

17.4 Typsystem

17.4.1 Statisches Typsystem

Es gibt keine Möglichkeit den Typ von Variablen, Funktionen, Modulen und Parametern fest zu vergeben und vom Compiler überprüfen zu lassen. Dies bewirkt zwar eine maximale Flexibilität, aber die Anzahl der möglichen Fehlersituationen, in die man durch dieses schwache Typisieren gelangen kann, macht den Nutzen wieder zunichte. Als weitere Unannehmlichkeit brauchen Variablen nicht deklariert zu werden. Damit wird bei Tippfehlern kein Fehler zur Compile–Zeit generiert, sondern Code für eine neue Variablen eingeführt. Dies führt dann garantiert[1] falschen Ergebnissen während der Laufzeit.

17.4.2 Records

Geordnete komplexe zusammengesetzte Datenstrukturen lassen sich in ProSet nur durch Tupel und ihre numerische Indizierung realisieren. Eine symbolische Adressierung kann durch die Verwendung von numerischen Konstanten und Makros simuliert werden. Somit sind wir ungefähr in den sechzigern Jahren, zur hohen Zeit von FORTRAN IV und Algol60, aber vor dem Auftauchen von Simula-67 und Pascal. Diese Sprachen führen das geniale Konzept eines Verbundes, genannt Record, ein.

17.4.3 Container

Mengen und Tupel können unter anderem als Container für Daten aufgefaßt werden. Unter diesem Gesichtspunkt wäre es angenehm, wenn man den Typ der Elemente dieses Containers angeben können. Somit wären generische Container möglich; der Compiler könnte Operationen auf den Elemten des Containers überprüfen.

[1] Willi ist in der Lage, ein Beispiel anzugeben, daß diese Aussage bzgl. des *garantiert* eingeschränkt werden kann. Das Beispiel kann bei Willi jederzeit nachgefragt werden, EMail an `willi@ls10.informatik.uni-dortmund.de` genügt...

17.4.4 Tupelzugriff und Funktionsaufruf

Syntaktisch sind Tupelzugriffe und Funktionsaufrufe mit nur einem Argument nicht voneinander zu unterscheiden. Dies ist zwar mathematische konsequent, da es semantisch äquivalent ist. Beim Programmieren jedoch ist es sehr störend, da man nicht genau sieht, ob hier eine Variable benutzt wird oder aber eine Funktion. Dies erschwert wesentlich die Lesbarkeit des Quelltextes und damit verbunden die Wartbarkeit.

In ProSet brauchen keine Variablen explizit definiert zu werden. Das führt in diesem speziellen Fall dazu, daß dann, wenn man sich beim Funktionsnamen vertippt, ProSet eine neue Variable (als Tupel) einführt, und darauf eine Selektion vornimmt. Dies ist unsinnig, sofern nicht in Form einer Schleife dieser Variablen ein Wert zugewiesen wird, und anschliessend dieselbe Stelle nocheinmal durchlaufen wird. Auf diese Weise werden kleine Fehler in ein Programm eingebaut, die von einem Menschen nur schwer aufzudecken sind. Einem Compiler würden dieser Fehler sofort auffallen, sofern er über eine entsprechende Typ–Überprüfung sowie unter Umständen einem Datenfluß–Analysator verfügt.

17.5 Syntax

Während der Arbeit mit ProSet sind uns einige Sachen bzgl. der Syntax aufgefallen, welche Anlaß zur Kritik geben.

Obwohl die `loop`-Anweisungen alle mit einem `end` abgeschlossen werden, scheint es in einer `repeat`-Schleife doch überflüssig zu sein, nach einem `until` noch ein `end` zu verlangen.

Die ganzen Ein-/Ausgabe Befehle blähen die Anzahl der Schlüsselwörter ebenfalls auf. Man kann diese Schlüsselwörter durch jeweils eine Funktion für die Eingabe und eine für die Ausgabe darstellen.

So ist z.B. auch die Anweisung `pass` durch ein Schlüsselwort gegeben, obwohl unserer Meinung nach auch die „leere Anweisung“ erlaubt sein sollte.

Laut Sprachbeschreibung von ProSet stellt das Schlüsselwort `newat` nur ein neues Atom bereit. Dieses läßt sich ebenfalls durch eine Funktion erledigen. Ebenso können die Schlüsselwörter `arb` und `random`, die die gleiche Funktionalität nur für verschiedene Datentypen zur Verfügung stellen, durch eine Funktion bereitgestellt werden. In diesem Kontext lassen sich dann noch die Schlüsselwörter `pow`, `domain` und `range` nennen, welche auch durch entsprechende Funktionen implementiert werden könnten.

17.6 Semantik

In ProSet werden sowohl die einfachen Datentypen wie `integer`, `real` und `string` als auch zusammengesetzte Datentypen wie `set` und `tuple` nur über Wertsemantik behandelt (vgl. [DWH+94]). Deshalb wird bei jeder Kopie der Inhalt der ursprünglichen Variable `a` an die neue Variable `b` übergeben, was den Nachteil der Speicherverschwendung, aber den Vorteil der einfacheren Handhabung bei der Kommunikation über Linda-Tupel-Räume in verteilten Systemen mit sich bringt. Sind die Rechner von gleicher Architektur, so arbeitet das Modell sehr gut, wenn eine unterschiedliche Rechnerarchitektur vorliegt, kann mit Hilfe von Typinformation oder einer kanonischen bzw. genormten Darstellung für alle Datentypen das Bit-Format festgelegt werden. Eine ausführliche Diskussion dieser Problematik befindet sich in [Tan92]. Es ist ein weiterer Vorteil der Wertesemantik, daß keine Seiteneffekte etwa bei Funktionen auftauchen können, da zwei Variablen nie identische Werte besitzen. Diese an und für sich angenehme Eigenschaft wird dann zum Nachteil, wenn man zum objektorientierten Paradigma wechselt.

Mit der Einführung der objektorientierten Programmierung wird es nötig, zumindest zusätzlich Referenzsemantik in ProSet zu integrieren. Die Philosophie geht davon aus, daß es Objekte (= Werte) gibt, und Namen (= Variablen, Objektidentifizierer) für die Objekte gibt. Somit existieren zwei Arten von Gleichheit, und zwar Werte-Gleichheit und Identität von Objekten. Nur über die Namen bzw. Objektidentifizierer können die Objekte selber angesprochen werden. Wie im wirklichen Leben, kann ein Objekt mehrere Bezeichnungen haben, über die das Objekt modifiziert werden kann. Diese Modifikationen sind damit automatisch auch für alle anderen Namen dieses Objektes sichtbar. Dies steht im Widerspruch zur Wertsemantik. Wertsemantik wird zumeist in OO–Sprachen nur für kleine Werte, wie etwa Zahlen, Buchstaben oder Wahrheitswerte benutzt. In Sprachen wie Smalltalk 80 ist es nicht möglich, für andere Wertetypen eine Wertsemantik zu verwenden, in Eiffel dagegen ist Referenzsemantik die Defaultsemantik, man kann aber Klassen definieren, die die Eigenschaft der Wertsemantik besitzen. In C++ dagegen hängt dies von Deklaration der Variablen ab, ein Ansatz, der unserer Meinung nach für mehr Verwirrung sorgt, als daß diese größtmögliche Flexibilität sich auszahlt.

17.7 Ein Resumee

Die Sprache ProSet in der momentan vorliegenden Implementation scheint uns ein geeignetes Werkzeug, um komplexe Algorithmen, die möglichst in einer mathematischen Formulierung vorliegen, schnell und ohne großen Aufwand in ausführbare Programme zu überführen. Dabei kann ein großer Teil der deklarativen Mengenbeschreibungen der Algorithmen eins zu eins übernommen werden. Mit Hilfe von Linda ist es möglich dieses auch für parallele Algorithmen zu tun, aber dort ist die Umsetzung der abstrakten Parallelitätsbeschreibung nicht so einfach zu übernehmen, wie die übliche mengenorientierte Beschreibung von sequentiellen Algorithmen. Für diesen Ansatz sind Module und Persistenz (in Sinne von Datenbanken, die Massendaten verwalten und in Zugriff haben) nicht notwendig, sondern nur angenehme Eigenschaften, die man in einigen Fällen nutzbringend einsetzen kann.

Für eine Weiterverwendung der so implementierten Algorithmen in Produktionsprogrammen, etwa durch den Ansatz des Transformationellen Programmierens, eignet sich die Sprache nicht. Die Qualitätsanforderungen der Software–Technologie wie insbesondere Robustheit und Korrektheit lassen sich mit ProSet nicht realisieren. Die Anzahl der potentiellen Fehlerquellen zur Laufzeit, die der Compiler nicht elimieren kann, steht im krassen Gegensatz zur vereinfachten Implementierung komplexer Algorithmen, die meistens nur einen kleinen Teil des gesamten Systems ausmachen. Wenn man ein komplexes System mit ProSet schreiben möchte, so muß der Compiler (und die weiteren Tools drumherum), nicht nur durch Operation auf hoher semantischer Ebene den Entwickler unterstützen, sondern gleichzeitig beim korrekten Zusammenbau der einzelnen Komponenten helfen, so daß zur Laufzeit keinerlei inkompatible Operationen ausgeführt werden können. Dies betrifft sowohl Module, als auch Tupel in Linda und die Daten in dem Programm, seien sie persistent oder nicht. Die beschriebene Problematik läßt sich durch die Einführung von statischer Typisierung vollständig umkehren, deshalb ist dies unser dringenster Wunsch für die Weiterentwicklung von ProSet.

Wenn man die Algorithmen, die man in ProSet formuliert hat, in ein anderes System einbinden möchte, hat man hier die Problematik, daß Daten zwischen zwei verschiedenen Sprachen und wahrscheinlich Sprach–Philosophien ausgetauscht werden müssen. Dieses ist Problem eigener Dimension, das hier nicht diskutiert werden soll.

18. Eine Sprachschnittstelle für ProSet

18.1 Motivation

Bevor verschiedene Realisierungsmöglichkeiten einer Sprachschnittstelle zwischen ProSet und anderen Programmiersprachen diskutiert werden, sollen hier nun einige Gründe genannt werden, die für eine solche Schnittstellen sprechen. Hierbei ist vor allem interessant, welche Richtungen die Schnittstelle zur Verfügung stellen muß, d.h. reicht es aus, wenn man andere Sprachen von ProSet aus aufrufen kann bzw. umgekehrt oder ist eine Schnittstelle für beide Richtungen notwendig.

Wichtigstes Argument für eine Sprachschnittstelle ist die Möglichkeit der Anbindung von grafischen Benutzeroberflächen. Da üblicherweise die Benutzeroberfläche das Programm ereignisorientiert steuert, müssen auf jeden Fall ProSet-Funktionen von der Oberfläche aufgerufen werden können. Dies würde aber nur dann ausreichen, wenn der Idealfall einträfe und das Programm nicht direkt in das Aussehen der grafischen Benutzeroberfläche eingreifen muß. Dies läßt sich aber häufig nur dadurch erreichen, daß Kontrollstrukturen durch die GUI selbst realisiert werden, komplexe Datentypen als Rückgabeparameter erforderlich werden oder die GUI zahlreiche Hilfsfunktionen aufrufen muß. Läßt man die Funktionalität der Regelsprache der GUI außer Acht, so erscheint also auch eine Schnittstelle von ProSet zur GUI als wünschenswert. Insbesondere bei der Implementierung von Scotland Yard hat sich gezeigt, daß eine Kommunikation in beide Richtungen nicht zu vermeiden ist.

Ein weiterer Grund für die Bereitstellung einer ProSet-Schnittstelle in anderen Programmiersprachen ist der Aspekt des Algorithmen-Prototypings. Hierbei könnte ein Programm bei der Lösung bestimmter Algorithmen auf besondere Eigenschaften von ProSet zurückgreifen. Allerdings ist dabei zu berücksichtigen, ob wesentliche Sprachelemente wie Parallelität und Persistenz überhaupt sinnvoll in eine Wirtsprache integriert werden können. Nur wegen der Unterstützung von Mengen und Tupeln wird schließlich keiner auf eine andere Sprache zurückgreifen.

Für die Möglichkeit, andere Sprachen von ProSet aus aufzurufen spricht vor allem die daraus resultierende Verfügbarkeit von zahlreichen Betriebsystem-Funktionen, was in den meisten Fällen allerdings nur eine C-Schnittstelle voraussetzen würde. Auch ein

Zugriff auf beliebige Bibliotheken anderer Sprachen scheint wünschenswert. Inwiefern ProSet von besonderen Eigenschaften anderer Sprachen profitieren könnte hängt dabei natürlich von der importierten Sprache selbst ab.

18.2 Arten der Einbindung

In diesem Abschnitt werden die Vor- und Nachteile der möglichen An- bzw. Einbindungsarten diskutiert. Im Mittelpunkt sollen dabei insbesondere die folgenden Methoden stehen:

- **Linken:** Bei dieser Lösung wird von den verschiedenen Sprachen zunächst jeweils Objektcode erzeugt, der anschließend zu einer ausführbaren Programmdatei gelinkt wird. Für diese Möglichkeit spricht dabei insbesondere der geringe Realisierungsaufwand. Problematisch hierbei ist allerdings die Datenrepräsentation. Für den Benutzer ließe sich die explizite Verwendung von Konstrukturen und Selektoren vermutlich nicht vermeiden. Inwiefern eine Kapselung der einzelnen Funktionsaufrufe sinnvoll oder gar notwendig ist, wäre dabei auch noch zu klären.

- **Embedded ProSet:** Bei dieser Art der Einbindung handelt es sich um eine Schnittstelle in beiden Richtungen. Hierbei ist es möglich in dem Quelltext einer Funktion verschiedene Programmiersprachen zu benutzen. Das Resultat für den Benutzer wäre dabei eine transparente Schnittstelle, da für ihn praktisch eine Programmiersprache existiert, deren Funktionsumfang aus der Summe der Leistungsfähigkeit von ProSet und der eingebundenen Sprache besteht. Ein weiterer Vorteil für den Benutzer wäre die dabei zwangsläufig vorhandene Transparenz bei der Datenrepräsentation. Ein Nachteil könnte allerdings die Unübersichtlichkeit sein, die beim Vermischen der Syntax verschiedener Sprachen entstünde. Dies betrifft insbesondere äquivalente Sprachkonstrukte wie beispielsweise Schleifen. Unübersehbar ist sicherlich auch der mit der Realisierung dieser Möglichkeit verbundene Aufwand. So muß ein Präprozessor entwickelt werden, um Quelltext für eine gemeinsame Sprache zu erzeugen. Berücksichtigt man, daß ProSetQuelltext zunächst in C-Code transformiert wird, wäre aber eine Lösung, die sich auf die Sprache C beschränkt, durchaus denkbar. Außerdem muß eine gewisse Kompatibilität der Datentypen gewährleistet sein, um die Vorteile dieser Lösung voll auszuschöpfen, wie z.B. der Zugriff auf ProSetTupel in C-Funktionen.

- **IPC (auch über (ProSet-) Linda):** Eine Realisierung der Sprachschnittstelle durch Integrierung einer IPC hätte insbesondere den Vorteil, daß eine solche Lösung universell einsetzbar wäre. So könnte die IPC auch zur Kommunikation zwischen mehreren parallel laufenden Prozessen genutzt werden. In diesem Zusammenhang sollte auch berücksichtigt werden, daß durch ProSet-Linda eine Kommunikation zwischen ProSet-Prozessen über Tupelräume bereits möglich ist. Eine Schnittstelle, die anderen Sprachen den Zugriff auf diese Tupelräume ermöglicht, würde also schon – mit gewissen Einschränkungen – ausreichen. Problematisch könnte allerdings die Systemabhängigkeit einiger IPCs (z.B. Message-Queues) sein, sofern Portabilität auf andere Plattformen von Bedeutung ist. Bei der Realisierung ist außerdem der zusätzliche Aufwand zur Implementierung der IPC zu berücksichtigen.

18.3 Datenrepräsentation

Unabhängig von der Art der Einbindung treten Probleme bei der Kopplung der unterschiedlichen Datenrepräsentationen auf. Insbesondere die Mengenorientiertheit von ProSet sorgt hier für Schwierigkeiten, da diese von den potentiellen Wirtssprachen nicht unterstützt wird. Aber auch Datentypen der Wirtssprachen, wie z.B. Verbünde (Records in Pascal oder Structs in C), lassen sich nur schwer oder gar nicht auf äquivalente ProSet-Datentypen abbilden.

Bei der Betrachtung dieser Probleme ist es nützlich, die betrachteten Datentypen hinsichtlich ihrer Komplexität zu klassifizieren und diese dann getrennt zu diskutieren.

Wir wählten eine Einteilung in vier Klassen, die im folgenden kurz erleutert wird.

- **Atomare Datentypen** sind Typen, die den natürlichen Zahlen, den reellen Zahlen oder alphanumerischen Zeichen entsprechen (z.B. Integer).
- **Plane Datentypen** sind Verkettungen atomarer Datentypen (z.B. Listen oder einfache Tupel).
- **Rekursive Datentypen** sind Verkettungen atomarer und/oder planarer Datentypen (z.B. Bäume, beliebige Mengen).
- **Module**

Für einfache Probleme kann der Austausch von atomaren Datentypen ausreichen, komfortabler und wünschenswerter wäre aber die Unterstützung zumindest von planaren Datentypen.

Dahingegen ist die Unterstützung von rekursiven Datentypen nicht unbedingt notwendig. Der „gemischte" Einsatz von ProSet und Wirtssprache ist ja gerade dazu da, daß in der jeweiligen Sprache das implementiert wird, was in ihr am einfachsten und effizientesten zu formulieren ist. Eine denkbare Anwendung wäre z.B. die Verwaltung von Mengen in ProSet, mit entsprechenden Einfüge- und Auslese-Prozeduren, die dann atomare Datentypen benötigt bzw. liefert. Diese können dann in der Wirtsprache weiterverarbeitet werden.

18.3.1 Unterstützung von Datentypen der Wirtssprache durch ProSet

Die Unterstützung atomarer Datentypen einer Wirtssprache durch ProSet bereitet keine Probleme, da ProSet alle üblichen atomaren Datentypen anbietet. Verkettete Datentypen könnten in Tupel umgesetzt werden, solange die Struktur plan (oder zumindest nicht-zyklisch) ist. Für Strings existiert ein entsprechender Datentyp in ProSet. Arrays können ebenfalls in Tupel umgesetzt werden. Probleme bereiten Typen wie Records, bei denen die Elemente über ihren Namen angesprochen werden. Ein solcher Typ sollte in ProSet aufgenommen werden, da er in anderen Sprachen weit verbreitet und in ProSet nur mit großer Mühe emulierbar ist. Komplexe Datentypen lassen sich nur über Umwege (s. Datenstruktur für den Graphen von ScotlandYard) in ProSet darstellen. Auf jeden Fall geht die Möglichkeit des komfortablen Zugriffs durch die Umsetzung verloren.

18.3.2 Unterstützung von ProSet-Datentypen durch die Wirtssprache

Übliche atomare Typen stellen auch hier keine Probleme dar. Anders sieht dies mit dem speziellen ProSet-Datentypen `om` aus. Dieser ist nur schwer in eine semantisch äquivalente Form zu transformieren. Denkbar wäre es, `om` in der Wirtssprache durch den Nullzeiger (`NIL`) darzustellen.

Mengen lassen sich problemlos in Listen umsetzen, allerdings mit dem Verlust der Reihenfolgelosigkeit. Auch Tupel – laut Definition besitzen sie eine unendliche Kardinalität – lassen sich zu Listen konvertieren, da sich die Unendlichkeit nur auf die Anzahl der `om`s bezieht. Unendlich viele Elemente ungleich `om` sind dagegen nicht zugelassen (z.B. keine Konstruktion der natürlichen Zahlen möglich). Die Reihenfolge der Elemente in den Tupeln bleibt in Listen erhalten.

18.4 Externe Datenhaltung

Anstelle der direkten Datenübergabe zwischen ProSet und Wirtssprache wird ein Modul zwischengeschaltet, das die Daten aus Modulen verschiedener Wirtssprachen halten kann. Dieses könnte dann von den verschiedenen Sprachen aus wie eine Datenbank bedient werden. Die Art des Zugriffes könnte von jeder Sprache aus in einer einheitlichen Form realisiert werden (z.B. über eine Library für jede unterstützte Sprache). Das Anstoßen anderer Module geschieht dann über IPC.

Eine andere Möglichkeit wäre ein RPC-Konzept, das um die Konvertierung in/von ProSet-Datentypen von/nach Datentypen der Wirtssprache(n) erweitert ist. Dies ist die allgemeinste Art der Anbindung, die auch am einfachsten erweiterbar ist. Für jeden neue Wirtssprache muß nur die Konvertierung hinzugefügt werden.

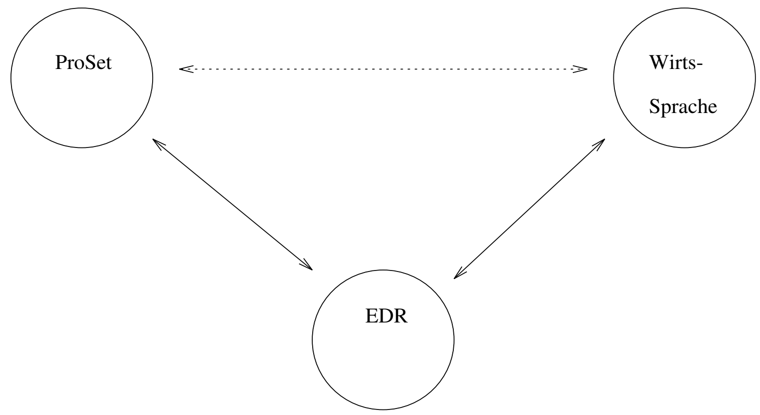


Abbildung 18.1: Externe Datenrepräsentation (EDR)

18.5 Sprachen

- C und C++
- Tcl/Tk

18.5.1 C/C++

Die Motivation für eine einfache Anbindungsmethode der Sprache C an ProSet ist zum einen die momentan große Verbreitung der Programmiersprache C und deren Verfügbarkeit. Wenn ein proprietäres Programmiersystem heutzutage eine Anbindung an eine verbreitete Sprache anbietet, so ist C mit Sicherheit die beste Wahl. Fast alle Systeme zur Erstellung und Benutzung graphischer Benutzungsoberflächen bieten mindestens eine C-Schnittstelle. Einige bieten darüber hinaus eine C++-Schnittstelle. Gleiches gilt für andere Systeme, wie z.B. Datenbanken, Netzwerkanbindung und teilweise auch für mathemathisch-numerische Systeme.

Eine GUI Anbindung für ProSet in C

Wie kann die Anbindung einer graphischen Benutzungsschnittstelle an ProSet in C aussehen? Die Implementierung der graphischen Benutzungsumgebung in C selbst ist wohl für eine Prototypingsprache kein adäquates Mittel. Es existieren verschiedene Werkzeuge, die eine C-Anbindung der mit ihnen erstellten graphischen Benutzungsumgebungen ermöglichen. Eine der momentan populärsten X11-Toolkits in der Unix Welt ist wohl OSF/Motif, der zum einen die direkte C-Programmierung ermöglicht, zum anderern aber auch eine compilierbare Scriptsprache (UIL) zur Erzeugung der graphischen Elemente bietet. Desweiteren existiert auch ein graphischer Motif Interfacebuilder der entsprechenden Motif Code erzeugt.

Die einzige Notwendigkeit auf der PROSET-Seite wäre die Implementierung einer komfortablen C-Schnittstelle.

Realisierung Die Einbettung von C Funktionen in PROSET ist noch realisierbar, da die Sprache PROSET keinem endgültig festgesetztem Standard unterliegt. C hingegen ist aus genau diesem Grunde nicht auf PROSET anpaßbar. Die komfortable Benutzung von PROSET Funktionen aus dem C Code ist daher schon weitaus schwieriger realisierbar. Ein Problem, vor dem auch Embedded SQL für C steht. Eine solche Schnittstelle muß aber nicht zwangsläufig den direkten Weg, also Einbettung von C Funktionen in PROSET gehen. Ein Funktionscompiler, der aus einer Schnittstellenbeschreibung die entsprechenden Adaptionsfunktionen für beide Seiten (C und PROSET) erzeugt, kann die Anbindung recht bequem machen, wenn die Schnittstellenbeschreibung erstellt ist. Dieser Compiler erzeugt dann den notwendigen Code zur Typenkonversionen, wobei auch notwendigerweise die Vielfalt der Datentypen durch die Fähigkeiten des Funktionscompilers beliebig eingeengt werden kann. Weiterhin bildet er für die C Seite die PROSET Funktionen auf C Funktionen, und die C Funktionen auf entsprechende PROSET Funktionen ab. Die so erstellte Bibliothek kann den mit den PROSET und den C Objektendateien einfach zusammengebunden werden. Solche Compiler werden insbesondere für transparente Netzwerksysteme eingesetzt. Der nicht nur auf SUN Systemen verbreitet SUN RPCgen ist ein hervorragendes Beispiel für einen solchen Compiler.

Eine GUI Anbindung für PROSET in C++

Für C++ gilt natürlich das Gleiche wie für C. Der einzige Unterschied ist die Möglichkeit, PROSET-Datentypen auf der C++ Seite in Form von entsprechenden Klassen zu definieren, und so auf natürliche Weise die Schnittstelle um komplexere PROSET Datentypen zu erweitern.

18.5.2 Tcl/Tk

Tcl, die Tool Command Language ist eine interpretierende Scriptsprache. Die große Stärke dieser Sprache ist ihre einfache Erweiterbarkeit. Die Erweiterbarkeit beschränkt sich nicht nur auf das Hinzufügen neuer Funktionsbibliotheken, sondern es ist auch möglich, mit den vorhandenen Sprachkonstrukten gänzlich neue Sprachkonstrukte, wie z.B. Schleifen oder konditionale Verzweigungen, zu definieren. Ein weiterer interessanter Punkt ist die Datenrepräsentation. Alle Daten wie Zahlen, Zeichenketten, Listen und sogar die Variablenamen selbst werden als String repräsentiert. Den Kern des Interpreters bildet eine Tabelle, in der alle Befehlsnamen eingetragen werden. Ein Statement wird vom Interpreter grundsätzlich als Kommandoname mit Stringargumenten behandelt. Das Programm kann diese Tabelle manipulieren, und auch eigene lokale Tabellen nebst Interpreter erstellen. Auch die Anbindung von C Primitiven wird über genau diese Struktur einfach erreicht. Auf diese Weise kann z.B. auch bei der Erzeugung einer Variablen ihr Name und eine entsprechende Funktion in diese Tabelle aufgenommen werden. Mit diesem Mechanismus wird eine gewisse Objektsicht[1] erreicht.

Genau diese Eigenschaften begründen wohl die Entstehung des Tcl Toolkits Tk. Tk ist eine Bibliothek zur Erstellung von graphischen Benutzungsoberflächen die auf das X11 System aufsetzt. Die beschriebene Interpreterstruktur und die C Schnittstelle ermöglicht erst die bequeme Einbindung der für die meisten X11 Toolkit typischen Callbackstruktur.

Tcl/Tk und ProSet

Da für ProSet bis dato keine Schnittstelle für die Implementierung graphischer Benutzungsoberflächen vorhanden ist, liegt es natürlich nahe Tcl/Tk in Bezug auf ProSet zu betrachten. Es gibt sicherlich verschiedene Anbindungsmöglichkeiten um ProSet mit Tcl/Tk zu verbinden.

Realisierung von ProSet/Tk Eine der wohl interesanntesten ist die direkte Anbindung von Tk ohne Tcl. Die Realisierbarkeit dieses Ansatzes hängt prinzipiell davon ab, wie weit Tk in seiner Implementierung unabhängig von Tcl ist. Wenn dem so ist, kann die Tcl Funktionstabelle im Prinzip von ProSet verwaltet werden. Alle Callbacks können so direkt auf ProSet Funktionen abgebildet werden. Andersherum kann ProSet dann über die Tabelle Tk Funktionalität benutzen.

Realisierung einer ProSet Tcl/Tk Schnittstelle Ein anderer Ansatz ist die Anbindung von ProSet an Tcl. Hierbei müsste eine Schnittstelle geschaffen werden, die in der Lage ist, die untypisierten Tcl Variablen nach ProSet und zurück zu transportieren. Mit der Prämisse, daß die Parameter feste und vor allen Dingen atomare Typen besitzen (keine Funktionen etc.) ist dies sicherlich machbar.

18.5.3 Objective ProSet

Eine objektorientierte Erweiterung von ProSet wäre durch das Einbetten von objektorientierten Sprachmitteln möglich. Brad J. Cox beschrieb bereits eine solche Einbettung von Objektkonzepten in vorhandene Sprachen. Das wohl bekannteste Ergebnis ist die Programmiersprache Objective C, welche von der Firma NeXT bereits zur Implementierung einer Betriebsystemschnittstelle und einer hervorragenden graphischen Benutzungsschnittstelle erfolgreich eingesetzt wurde. Hewlet Packard, Digital und Sun haben sich mittlerweile Lizenzen dafür gesichert. Aber auch Objective Pascal, Objective Fortran und sogar Objektive Cobol sind bereits verfügbar. Letztendlich ist aufgrund

[1] Aufgrund der fehlenden Beschreibungsmöglichkeiten z.B. für Klassenhierarchien, kann von echter Objektorientiertheit hier wohl nicht gesprochen werden.

der Smalltalk-ähnlichen Struktur von Objective C die Erzeugung einer graphischen Benutzungschnittstelle, entsprechende Klassenbibliotheken vorausgesetzt, verglichen mit anderen Systemen nahezu ein Spaziergang.

19. Hypermedia-Werkzeuge zur Softwareentwicklung

19.1 Motivation

Bei der modernen Softwareentwicklung entstehen sehr viele Dokumente unterschiedlicher Art: neben dem klassischen Textdokument können auch Ton- und Bilddokumente hilfreich sein, um eine Verständigung zwischen Auftraggeber und den verschiedenen Personen des Entwicklerteams zu unterstützen. Diese unterschiedlichen Dokumente können in einem Hypermedia-System integriert werden. Von der Verwendung von Hypermedia in der Softwareentwicklung versprechen wir uns im wesentlichen folgende Vorteile:

1. Die zentrale Verwaltung sämtlicher Erzeugnisse (Dokumente, Programme, Prototypen, Testpläne etc.) eines Softwareentwicklungsprozesses. Die zentrale Verwaltung unterliegt einer Person oder Personengruppe, so daß Zuständigkeiten von dieser Person/Gruppe delegiert werden können.
2. Prototyping wird unterstützt durch:
   - Vereinfachung der Kommunikation zwischen Entwickler, Anwender und Auftraggeber durch eine einheitliche Oberfläche auf einem gemeinsamen Datenbestand. So kann der Auftraggeber zum Beispiel verschiedene (von den Entwicklern freigegebene) Prototypen ausprobieren und testen.
   - Engere Abstimmung zwischen verschiedenen Teilnehmern. Jeder kann aktuelle Anmerkungen zu Dokumenten, die er bearbeitet oder die ihm zur Information vorliegen, lesen. Es wird niemand „vergessen“ und niemand verwendet ein nicht aktuelles Dokument.
   - Frühe Dokumentation ermöglicht das frühzeitige Nutzen von Prototypen und somit das Aufdecken und Beseitigen von Unklarheiten und Ungenauigkeiten der Anforderung.
3. Eine Vereinfachung der Kontrolle und Verwaltung von Versionen durch sogenannte Versionslinks (siehe unten).
4. Einbindung multimedialer Dokumente, wie zum Beispiel

- Videofilme über den Auftraggeberbetrieb zur Präsentation der Arbeitsabläufe.
- Interviews mit Auftraggeber und Anwendern.
- Fotographien, Formulare und Skizzen.
- Animationen von Programmen und Spezifikationen.

Im folgenden definieren wir Dokumentenarten, Links, Sichten und Rollen, die im Softwareentwicklungsprozeß auftreten können.

19.2 Dokumentenarten

Bei den Dokumentenarten haben wir uns grob an die Phasen des Softwareentwicklungsprozesses gehalten. Allerdings spezifizieren wir hier nur grobgranulare Dokumente.

19.2.1 Analysephase

In der Analysephase gibt es die *informelle Anforderungsbeschreibung*, die vom Auftraggeber erstellt wird und von den Entwicklern ergänzt werden kann. In dieser Dokumentenart können unterschiedliche multimediale Inhalte enthalten sein, z.B. die oben erwähnten Interviews mit dem Auftraggeber.

19.2.2 Entwurfsphase

In der Entwurfsphase fallen zwei Dokumentarten an:

- In der *Spezifikation* erfolgt eine formale Beschreibung des Problems und der Lösungsansätze.
- Die *Dokumentation* dient als Ausgangsdokument für die späteren Handbücher und einem Hilfesystem für die Anwender. Außerdem dient sie dem Verständnis und der Bewertung vorliegender Prototypen.

19.2.3 Implementierungs- und Testphase

Während der Implementation und des Tests entstehen Dokumente dreier verschiedener Arten.

- Der *Quellcode* besteht aus einfachen Textdateien.
- Zu den *ausführbaren Programmen* gehören neben dem Endprodukt auch Zwischenversionen und Prototypen.
- Beim Testen der Programme entstehen *Testdokumente*, die Testdaten, Protokolle o.ä. enthalten.

19.2.4 Einsatz- und Wartungsphase

In der letzten Phase fallen keine neuen Dokumentarten mehr an. Hier werden lediglich andere Sichten auf bestehende Dokumente aus den vorhergehenden Phasen eingesetzt.

19.3 Links

Links sind Verknüpfungen zwischen verschiedenen Dokumenten oder innerhalb eines Dokuments. Sie ermöglichen das Navigieren im Hypermediasystem. Wir haben fünf Arten von Links definiert.

- Ein *Verfeinerungslink* führt von einer groben Beschreibung zur detaillierteren Beschreibung des gleichen Sachverhalts.
- Ein *Strukturlink* verweist auf eine andersartige Darstellung des gleichen Sachverhalts.
- Ein *Reihenfolgelink* ermöglicht es dem Benutzer, Dokumente in thematischer Reihenfolge abzurufen.
- Ein *Versionslink* verbindet verschiedene Versionen eines Dokumentes.
- Ein *Querverweis* schließlich bildet eine Verbindung verschiedener Sachverhalte.

19.4 Sichten

Sichten sind als Filter zu verstehen, die aus den gesamten Daten eines Dokuments nur diejenigen zur Ansicht freigeben, die für eine bestimmte Aufgabe relevant angesehen werden.

Wir haben insgesamt sechs Sichten definiert.

- In der Sicht *Anforderungsdefinition* werden aus den Dokumenten der Art Anforderungsbeschreibung und formale Beschreibung die Definitionen extrahiert.
- Die *Entwurfssicht* bietet die Daten des Entwurfsdokuments an.
- Auf den Quellcode kann die *Programmtextsicht* angewendet werden. Eventuell unter Berücksichtigung des *literate programming* .
- Auch *ausführbares Programm* bzw. *Prototyp* ist eine Sicht auf Dokumente des Hypermediasystems.
- Für den Anwender ist die *Hilfesicht* gedacht (als Online-Dokumentation oder Handbücher).
- Um die Entwicklungsabfolge zu extrahieren, gibt es die Sicht auf *Versionen*.

19.5 Rollen

In einem Softwareentwicklungsprozeß gibt es verschiedene Rollen, in denen Personen agieren können. Zu Rollen gehören jeweils spezifische Befugnisse bezüglich des Produktes und somit auch bezüglich des Hypermediasystems. Eine Person kann mehreren Rollen inne haben.

- Der *Auftraggeber* ist die Person, die ein Problem hat, das durch Software gelöst werden soll.
- Der *Endanwender* ist derjenige, der mit dem fertigen Produkt arbeiten soll.
- Die Rolle des *Entwicklers* zerfällt in eine bezüglich der Kompetenz hierarchische Rollenunterteilung.
  - Der *Manager* leitet das Projekt (Projektmanager) oder eine Teilgruppe (Gruppenmanager).
  - Der *Implementierer* erstellt den Quellcode und ausführbare Programme bzw. Prototypen.
  - Der *Designer* entwirft Lösungsansätze.
  - Der *Tester* testet ausführbare Programme und Prototypen nach vorgegebenen Testplänen und führt eine Bewertung dieser Zwischenprodukte vor.
  - Der *Dokumentierer* dokumentiert den Ablauf des Softwareentwicklungsprozesses.

19.6 Anforderungsdefinition und Exploration

Im folgenden beschäftigen wir uns mit den speziellen Bereichen Anforderungsdefinition und exploratives Prototyping: In wie weit läßt sich Hypermedia für die Softwareentwicklung in diesen Bereichen einsetzten?

19.6.1 Hypermedia beim Erstellen der Anforderungsbeschreibung

Wie wir in der ersten Phase festgelegt haben, ist die Anforderungsbeschreibung informell. Daher darf ein Hypermediasystem den Auftraggeber bei seiner Beschreibung des Problems nicht zu stark einschränken. Das hat zum Beispiel zur Folge, daß neben klassischen Textdokumenten auch Ton-, Film- und Grafikdokumente in die Anforderungsbeschreibung aufgenommen werden dürfen. Diese Dokumente sollen dabei helfen, die vorgezeichneten Verständnisschwierigkeiten zwischen Auftraggeber und Entwickler auf ein Minimum zu beschränken: „Ein Bild sagt mehr als tausend Worte“.

Durch diese in ihrem Aufbau sehr unterschiedlichen Einzeldokumente ergeben sich für das Hypermediasystem besondere Schwierigkeiten.

Strukturierung der Dokumente

Der Auftraggeber soll bei der Erstellung der Anforderungsbeschreibung in Bezug auf die Gestaltung größtmögliche Freiheit genießen. Das Hypermediasystem muß nun den Entwickler in die Lage versetzen, dieses Chaos so zu strukturieren, daß in der Entwurfsphase ohne allzu großen Aufwand daraus eine formale Spezifikation erstellt werden kann. Dies sollte sich jedoch realisieren lassen, ohne daß der Auftraggeber dies bemerkt, d.h. für den Auftraggeber sollen seine Dokumente immer so angeordnet sein, wie er es für sinnvoll hält. Dazu bietet sich eine extra Sicht für die Entwickler an.

Diese Entwicklersicht sollte es ermöglichen,

- für die weitere Arbeit eher unwichtige Dokumente und Dokumentstellen auszublenden (wichtig: dies muß für die unterschiedlichen Dokumentformen Text, Ton bzw. Bild gleichermaßen möglich sein);
- die Reihenfolge der Dokumente sollte nicht unbedingt chronologisch sein, sondern frei konfigurierbar, so daß zum Beispiel der logische Ablauf eines Prozesses dargestellt wird;
- wenn der Auftraggeber ein bestehendes Dokument der Anforderungsbeschreibung ändert, muß der Entwickler die Möglichkeit haben, sowohl die geänderte als auch die originale Version abzurufen, inklusive eventueller Zwischenversionen.

Außerdem sollte es dem Entwickler möglich sein, Anmerkungen, wie zum Beispiel formale Beschreibungen, an die Dokumente des Auftraggebers an beliebiger Stelle im Dokument anzuhängen. Diese können dann zum Beispiel mit einer speziellen Sicht extrahiert werden, um sie als Ausgangspunkt(e) für die formale Spezifikation zu verwenden.

Links innerhalb von Ton- und Filmdokumenten

Die Möglichkeit der Integration von Ton- und Filmdokumenten bietet (aus der Sicht der Anforderungsdefinition für ein Hypermediasystem) nicht nur Vorteile. Längere Dokumente dieser Art können möglicherweise viel unwichtige Information enthalten. Ein Hypermediasystem muß nun den Benutzer bei der optimalen Verwendung dieser Dokumente unterstützen. Es muß daher die Möglichkeit bestehen, innerhalb von diesen Dokumenten mit Links an bestimmte, unter einem Aspekt interessante Stellen zu springen. Eine Möglichkeit, dies (optisch) zu realisieren, ist es, beim Abspielen eines Ton- bzw. Filmdokumentes bestehende Links von der aktuell gespielten Stelle in einer Liste anzuzeigen. Sie werden beim Erreichen einer Stelle in diese Liste eingefügt und nach einer bestimmten (variierbaren) Zeit nach Verlassen der Stelle wieder aus der Liste entfernt.

Damit besteht zum Beispiel die Möglichkeit, von Stellen, in denen gesagt wird „... wie ich später noch näher beschreiben werde ...“ an die Stelle der näheren Beschreibung vorzuspringen (optional könnte auch wieder ein automatischer oder vom System angebotener Rücksprung stattfinden).

19.6.2 Hypermedia beim explorativen Prototyping

Eine weitere Anwendung für ein Hypermediawerkzeug in der Softwareentwicklung ist die Unterstützung des explorativen Prototypings. Im Gegensatz zum evolutionären Prototyping wird hier nicht ein Teil des Softwareproduktes unter Mithilfe verschiedener Beteiligter immer weiter entwickelt und verbessert, sondern es werden mehrere unabhängig nebeneinander stehende Prototypen entworfen, die dann bewertet werden.

Aufgrund dieser Bewertung fällt dann die Entscheidung, die zur Weiterentwicklung eines, unter Umständen auch mehrerer Teilprodukte führt. Bei dieser Vorgehensweise findet eine Bewertung nicht nur durch die an der Entwicklung beteiligten Personen, sondern speziell durch die späteren Anwender statt.

Während die Anwender ihre Erfahrungen zumeist in Form eines Interviews oder einer schriftlichen Bewertung abgeben, die dann vom Hypermediasystem wie vorgeschlagen verwendet werden kann, sind von den Entwicklern häufiger Meßwerte zu erwarten.

Dabei wird zwischen einer objektiven und einer subjektiven Bewertung unterschieden. Grundlage der objektiven Bewertung sind Kennzahlen, die bestimmte Leistungscharakteristika wiederspiegeln. Standardkriterien sind z.B.

- Geschwindigkeit,
- Ressourcenverbrauch,
- Verhalten in Ausnahmesituationen.

Eine subjektive Bewertung ist nur möglich, wenn eine Interaktionsschnittstelle zumindest Teil des zu untersuchenden Prototypen ist. Diese könnte mit folgenden Kriterien bewertet werden:

- Ergonomie,
- Übereinstimmung mit den erwarteten Anforderungen (im Gegensatz zu den formulierten Anforderungen),
- Integration in vorhandene Softwarestrukturen,
- subjektive Geschwindigkeit.

Die so erzielten Ergebnisse sind wiederum Dokumente, die nur durch Nutzung eines Hypermediasystems effizient integriert werden können. Daran schließt sich eine Präsentation der Ergebnisse an.

Da sowohl die subjektiven als auch die objektiven Angaben mittels Statistiken ausgewertet werden sollen, ist zunächst ein Konzept erforderlich, das einen Vergleich zwischen subjektiven und objektiven Kriterien ermöglicht. Ein naheliegendes Konzept wären Fragebögen, die entweder standardisiert nach bestehenden Normen vom System selbst zur Verfügung gestellt werden oder individuell eingebunden werden können.

Gerade Fragebögen ermöglichen eine automatisierte Auswertung. Durch die Einbindung in ein Hypermediasystem können mit Hilfe von Fragebögen folgende Funktionen realisiert werden:

- Eindeutige Zuordung eines Fragebogens zum bewerteten Prototypen.
- Die genauen Zeitangaben ermöglichen ein präzises Verlaufsprotokoll, an dem man die Wirkungen der Veränderungen am Prototypen auf den Benutzer zeitlich verfolgen kann (Trends).
- Auch die Fragebögen sind Gegenstand der Entwicklung. Falls gegen Ende eines Projektes festgestellt wird, daß Anforderungen unberücksichtigt geblieben sind, können die Fragebögen dahingehend ergänzt werden.

Ein entscheidender Nachteil uns bekannter bisher verwendeter Ansätze [NO93], [FHS+92] ist, daß Daten, die über verschiedene Prototypen gesammelt worden sind, nur sehr schwer ausgewertet werden können. Dies liegt zum einen an der knappen Ressource Zeit, und zum anderen an der Unvergleichbarkeit verschiedener Darstellungsformen. Ziel muß es also sein, ein transparent in die SEU eingebettetes Tool zur Verfügung zu stellen, daß eine Darstellung der Daten liefert, mit der die Auswertung ermöglicht wird.
Ohne entsprechende Tools, die möglicherweise transparent in die SEU eingebettet sind, wird eine solche Analyse sehr erschwert.
Ein Werkzeug zur Auswertung der erstellten Statistiken sollte folgende Funktionen bieten:

- Automatische Datenextraktion aus standardisierten Dokumenten wie z.B. den Fragebögen.
- Statistische Analyse der gewonnenen Daten.
- Tabellarische und graphische Darstellung der Ergebnisse.

Die Ergebnisse dieses Werkzeuges werden innerhalb des HMS in einer Dokumentenart *Statistik* mit folgenden Anforderungen präsentiert:

1. Einheitliches Aussehen zur geforderten Vergleichbarkeit.
2. Quellen der Statistik müssen für Analysierer transparent sein.

Aus obigen Ausführungen ergeben sich zwei neue Rollen im Softwareentwicklungsprozeß:

1. Der Datenmoderator, eine Person, die die Testdaten sammelt und allen anderen zur Verfügung stellt.
2. Der Datenanalysierer, der die Daten analysiert.

20. Zu guter letzt...

20.1 Abschlußbemerkungen der Betreuer

Die Arbeit mit dieser Projektgruppe hat Spaß gemacht! Trotz gelegentlicher Koordinationsprobleme zwischen den Teilgruppen (und auch innerhalb der Teilgruppen) war die Stimmung doch immer gut und Schläge unter die Gürtellinie sind — soweit wir das überblicken konnten — nicht vorgekommen. Die aufgetretenen Koordinationsprobleme und das gelegentliche Nichteinhalten der Spezifikation kann (zumindest teilweise) wohl auch uns als Betreuer angelastet werden. Eine detailliertere Spezifikation hätte wohl einige der Probleme nicht auftreten lassen oder zumindest früher aufgedeckt. Zukünftigen Betreuern von Projektgruppen sei empfohlen, die Koordination der Teilgruppen auf eine *formale administrative* Basis — im Sinne eines explizit unterstützten Software-Prozesses — zu stellen. Beispielsweise könnte eine Teilgruppe die Programme einer anderen Teilgruppe systematisch gegen deren Spezifikation testen. Derartige Regeln sollten die Kreativität der Teilnehmer möglichst nicht beeinträchtigen (jede Medaille hat immer auch zwei Seiten). Sinnvoll erscheint uns auch eine größere Flexibilität in der Einteilung der Teilgruppen, um auch bei unterschiedlichen Fortschritten in den Teilgruppen geeignet reagieren zu können.

Die anfänglichen Probleme mit dem ProSet-Compiler führten zwar gelegentlich zu (kurzfristigen) Frustrationen bei einigen Teilnehmern, konnten jedoch die Begeisterung für die zu lösende Aufgabe kaum bremsen. Informatiker spielen ja von Natur aus gern (mit dem Computer). Trotz vieler Probleme mit den (teilweise noch vorhandenen) Schwächen von ProSet, konnte das Minimalziel — ein funktionierendes Scotland-Yard-Programm — erreicht werden. Insbesondere die graphische Benutzungsschnittstelle hat unsere Erwartungen übertroffen und die Planung der Detektive hat sich als recht erfolgreich erwiesen, wie wir u.a. bei der Online-Präsentation in Amsterdam sehen konnnten. Die Regel-Gruppe sei nicht unerwähnt, da sie für den nötigen *Klebstoff* zwischen den beiden anderen Komponenten gesorgt hat. Da die Realisierung des Scotland-Yard-Programms mehr Zeit in Anspruch genommen hat als ursprünglich geplant, konnte in der zweiten Phase die Spezifikation von Werkzeugen leider nur ansatzweise geschehen.

20.2 Abschlußbemerkungen der Studenten

Zu der abschließenden Bemerkung der beiden PG-Betreuer Willi Hasselbring und Claus Pahl wollen wir, die Studenten, auch einen Teil beitragen. Dies scheint uns zur Zeit besonders wichtig, da viel über die Durchführung von PG's diskutiert wird.

Die Stimmung während des ganzen Jahres unserer PG war ausgezeichnet und von uns allen, den Studenten, nicht in dieser Form erwartet worden. Viele Berichte, Gerüchte und der Vortrag von Benedikt Stockebrandt bei der Vorstellung der verschiedenen Projektgruppen hatten einige Teilnehmer erheblich verunsichert. Einige von uns hatten den Eindruck, die PG sei ein Jahr des absoluten Chaos' im Streit mit den Betreuern und dem Zittern um den PG-Schein. Diese Vorstellungen haben sich in unserer PG nicht verwirklicht, und die Gründe dafür wollen wir in diesem kleinen Erfahrungsbericht zusammenfassen.

Professor Doberkat, der Leiter der Veranstaltung, kam mit uns nur während der ersten Seminarphase in Kontakt und präsentierte sich als sehr umgänglicher Mensch, der den beiden Betreuern aus unserer Sicht freie Hand ließ. Dieser Eindruck bestätigte sich während der ganzen Zeit.

Willi und Claus haben somit den Grundstein für die Atmosphäre der PG in der ersten PG-Sitzungen selbst gelegt. Ihr Führungsstil war zu keiner Zeit autoritär, sie erwarteten jedoch engagierte Mitarbeit. So wurde z.B. das Fehlen eines Studenten im Protokoll vermerkt, gelegentliche Verspätungen blieben jedoch unberücksichtigt und wurden nicht, wie sonst üblich, auf die Minute genau im Protokoll vermerkt. Dies führte dennoch nicht dazu, daß jede Sitzung verspätet begann.

Dieses kulante Verhalten wurde von Anfang an von allen PG-Teilnehmern übernommen. Streitigkeiten kamen nicht vor. Untereinander entwickelte sich ein erhebliches Maß an Flexibilität und Freundschaft. Dies spiegelte sich zum einen in der Arbeitsweise der PG wieder, da Arbeiten anderer bei Gelegenheit übernommen wurden, zum anderen auch in der Form von gemeinsamen PG-Treffen in der Freizeit.

Zur angenehmen Arbeitsatmosphäre trug auch die Tatsache bei, daß die Zielanforderungen realistisch waren und zudem flexibel angepaßt wurden. Die von uns verwendete und am Lehrstuhl entwickelte Programmiersprache ProSet bereitete einige Probleme. Die aufgedeckten Schwächen wurden von den Betreuern zwar schnell beseitigt, es wurde allerdings klar, daß wir nicht im geplanten Zeitraum das gesetzte Ziel erreichen konnten. Trotzdem wurde von uns nicht verlangt, in der vorlesungsfreien Zeit zu arbeiten, sonderen es wurden die Anforderungen für den zweiten Teil der PG eingeschränkt.

Wir fassen Projektgruppen als eine Lehrveranstaltung auf, die in erster Linie das Arbeiten in Teams vermitteln soll. Gerade unter diesem Gesichtspunkt war unsere PG erfolgreich.

Ein häufig geäußerter Kritikpunkt an der Veranstaltung PG ist das Bestreben mancher Betreuer, diese zu einem besseren Programmierpraktikum verkommen zu lassen. Am Anfang unserer PG war z.B. die Frage zu klären, mit welchem Interface-Builder gearbeitet werden soll. Da am Lehrstuhl der ISA-Dialog-Manager verwendet wurde, wäre es für die Betreuer sicherlich arbeitssparend gewesen, wenn auch wir diesen benutzt hätten. Speziell weil viele unserer dann zu entwickelnden Funktionen am Lehrstuhl hätten weiterverwendet werden können. Uns wurde die freie Wahl gelassen, und wir haben uns für Tcl/Tk entschieden. Eine solche Vorgehensweise wirkte auf uns außerordentlich motivierend und nicht zuletzt deshalb wurde die graphische Benutzungsoberfläche weit über die Minimalanforderungen hinaus weiterentwickelt. Dies führte außerdem zum Einsatz von Tcl/Tk am Lehrstuhl.

Sinnvoll erschien uns auch die Möglichkeit, auch an den Rechnern des Lehrstuhls zu arbeiten, und zwar aus zwei Gründen: Zum einen waren die Betreuer immer nur zwei Räume entfernt, was zu einem engeren Kontakt führte. Zum anderen durften wir die Ressourcen des Lehrstuhls benutzten, was manche Arbeiten erheblich beschleunigte.

Zur Überraschung aller konnten wir uns noch am Ende der PG alle ausstehen und hatten auf der Abschlußfahrt eine Menge Spaß. Im Gegensatz zu den vielen kursierenden Berichten über mißglückte PGs wollen wir hiermit einmal deutlich machen, daß es auch anders geht. Vielleicht hilft dieser Bericht anderen herauszufinden, was in ihrer PG schief gelaufen ist. In diesem Sinne wollen wir uns ausdrücklich bei allen Beteiligten für das Gelingen der PG bedanken.

Literaturverzeichnis

[Bal90] Henri E. Bal. Languages for Parallel Programming. Rapport IR–226, Vrije Universiteit Amsterdam. Faculteit for Wiskunde en Informatica, October 1990.

[Bal91] Henri E. Bal. A comperative study of five parallel programming languages. In *European spring 1991 conference on open distributed systems*, Tronsø, Norway, May 1991.

[Bau93] F. L. Bauer. Software Engineering — wie es begann. *Informatik–Spektrum*, 16:257–258, 1993.

[Bjo91] Robert D. Bjornson. *Linda on Distributed Memory Multiprocessors*. PhD thesis, Department of Computer Science, Yale University, 1991.

[BKKZ92] R. Budde, K. Kautz, K. Kuhlenkamp, and H. Züllighofen. *Prototyping – An Approach to Evolutionary System Development*. Springer–Verlag, 1992.

[BZ90] Wolfgang Becker and Adreas Zell. Kooperatives Planen unabhängiger Agenten. Technical report, Institut für parallele und verteilte Höchstleistungsrechner. Universität Stuttgart, September 1990.

[Cag] Martin R. Cagan. The hp softbench environment: An architecture for a new generation of tools.

[Con] X Consortium. *Xlib Programming Manual for Version 11*.

[DF89] E.-E. Doberkat and D. Fox. *Software Prototyping mit SETL*. Leitfäden und Monographien der Informatik. Teubner–Verlag, 1989.

[DFG+92] E.-E. Doberkat, W. Franke, U. Gutenbeil, W. Hasselbring, U. Lammers, and C. Pahl. ProSet — a language for prototyping with sets. In N. Kanopoulos, editor, *Third International Workshop on Rapid System Prototyping*, pages 235–248, 1992.

[DWH+94] E.-E. Doberkat, W.Franke, W. Hasselbring, C. Pahl, H.-G. Sobottka, and B. Sucrow. ProSet — prototyping with sets. language definition. Technical report, Software Technology / Computer Science. University Dortmund, 1994.

[EM90] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 2*. EATCS Monographs on Theoretical Computer Science. Springer–Verlag, 1990.

[FGH+93] W. Franke, U. Gutenbeil, W. Hasselbring, C. Pahl, H.-G. Sobottka, and B. Sucrow. Prototyping mit Mengen — der ProSet–Ansatz. In *Proc. Requirenments Engeneering — Prototyping*, Bonn, 1993.

[FHS+92] James C. Ferrans, David W. Hurst, Michael A. Sennett, Burton M. Covnot, Wenguang Ji, Peter Kajka, and Wei Ouyang. Hyperweb: A framework for hypermedia-based environments. *ACM*, 12:1–10, 1992.

[Flo84] C. Floyd. A systematic look at prototyping. In R. Budde, K. Kuhlenkamp, L. Mathiassen, and H. Züllinghofen, editors, *Approaches to Prototyping*. Springer Verlag, 1984.

[GC89] David Gelernter and Nicholas Carriero. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[GC92] David Gelernter and Nicholas Carriero. Coordination Languages and their Significance. *Communications of the ACM*, 35(2):97–107, February 1992.

[Gel85] David Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[GJM91] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, 1991.

[GSC91] Michael Goedicke, Harald Schumann, and Joachim Cramer. On the specification of software components. In *Proc. of the 6th Intl ACM/IEEE Workshop on Software Specification and Design*, Oktober 1991.

[Has93] Wilhelm Hasselbring. Prototyping parallel algorithms with ProSet–linda. In J. Volkert, editor, *Parallel Computation (Proc. Second International ACPC Conference)*, pages 135–150, Gmunden, Austria, October 1993. Springer–Verlag.

[HD91] H.Haugeneder and D.Steiner. Cooperation structures in multi–agent–systems. In Springer-Verlag, editor, *Verteilte KI und kooperatives Arbeiten*, volume 291 of *Informatik–Fachberichte*. W.Brauer and D.Hernandez, 1991.

[HI88] S. Hekmatpour and D. Ince. *Software Prototyping, Formal Methods and VDM*. Addison-Wesley, 1988.

[HP83] B. Steinbrüggen H. Partsch. Program transformation systems. *ACM Computer Surv.*, 15:199–236, 1983.

[HP87] B. Möller H. Partsch. Konstruktion korrekter Programme durch Transformation. *Informatik Spektrum*, 10:309–323, 1987.

[JPSW93] Gerald Junkermann, Burkhard Peuschel, Wilhelm Schäfer, and Stefan Wolf. Merlin: Supporting cooperation in software development trough a knowledge-based environment. SWT Memo 70, Lehrstuhl für Software–Technologie, Fachbereich Informatik, Universität Dortmund., September 1993.

[JPSW94] Gerald Junkermann, Burkhard Peuschel, Wilhelm Schäfer, and Stefan Wolf. Merlin: Supporting cooperation in software development trough a knowledge-based environment. In Anthony Finkelstein, Jeff Kramer, and Bashar Nuseibeh, editors, *Software Process Modelling and Technology*, number 3 in Advanced software development series, pages 103–129. Research Studies Press Ltd., 1994.

[Kel93] Udo Kelter. Integrationsrahmen für Software-Entwicklungsumgebungen. *Informatik-Spektrum*, 16, 1993.

[MK89] A.S. Tanenbaum M.F. Kaashoek, H.E. Bal. Experience with the distributed data structure paradigm in linda. *USENIX/SERC Workshop on Experiences with Building Distributed and Multiprocessor Systems*, pages 175–191, Oktober 1989.

[MS91] J. Müller and J. Siekmann. Structured social agent. In Springer-Verlag, editor, *Verteilte KI und kooperatives Arbeiten*, volume 291 of *Informatik–Fachberichte*. W.Brauer and D.Hernandez, 1991.

[Nag93] M. Nagl. Software-Entwicklungsumgebungen: Einordnung und zukünftige Entwicklungslinien. *Informatik-Spektrum*, 16, 1993.

[NIS91] NIST ISEE Working Group. *Reference Model for Frameworks of Software Engineering Environments*. ECMA TC33 Task Group, 1991.

[NO93] Susanne Neubert and Andreas Oberweis. Einsatzmöglichkeiten von hypertext beim software engineering und knowledge engineering. In *Forschungsberichte*. H. Schmeck and D. Seese and W. Stucky and R. Studer, 1993.

[PBDKS91] G. Pomberger, W. Bischofberger, W. Pree D. Kolb, and H. Schlemm. Prototyping–oriented software development — concepts and tools. 12:43–60, 1991.

[PS92] Burkhard Peuschel and Wilhelm Schäfer. Efficient execution of rule based persistent software processes. In *Proceedings of the 14th ICSE*, Melbourne, Australia, 1992. IEEE Press.

[PW93] Burhard Peuschel and Stefan Wolf. Architecural support for distributed process-centred software development environments. In *Proceedings of the 8th International Software Process Workshop*, Dagstuhl, Germany, March 1993.

[RB91] A. Sherman R. Bjornson, C. Kolb. Ray tracing with network linda. *SIAM News*, Januar 1991.

[RF93] Wilhelm Schäfer Reiner Fehling. Opus: Konzept und werkzeug für die unterstützung verteilter, modularer software–entwicklung. Technical Report 68, Lehrstuhl für Software–Technologie, Fachbereich Informatik, Universität Dortmund, Mai 1993.

[See87] Silke Seehusen. Bestimmung von parallelitätseigenschaften in modularen systemen mit pfadausdrücken. Forschungsbericht 246, Universität Dortmund, 1987.

[SG94] H. Schumann and M. Goedicke. Component–oriented software development with Π. Internal Report, 1994.

[Spi83] Ravensburger Spiele. Scotland Yard. Otto Maier Verlag, Ravensburg, Germany, 1983.

[SW93] A. Spillner and F. H. Winkler. 25 Jahre Software–Engineering. *Informatik–Spektrum*, 16:259–260, 1993.

[SW94] Wilhelm Schäfer and Stefan Wolf. Cooperation patterns for process-centred software development environments. Technical Report 73, Lehrstuhl für Software–Technologie, Fachbereich Informatik, Universität Dortmund, September 1994.

[Tan92] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 1992.

[You] Doug Young. *X Window System, Programming and Applications*.

Teil V

Anhang

A. noweb–Index

A.1 Index der Bezeichner

Dieser Index wird automatisch generiert.