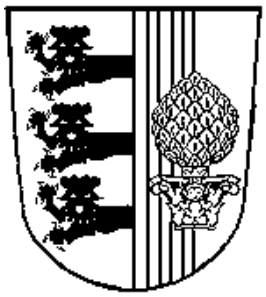
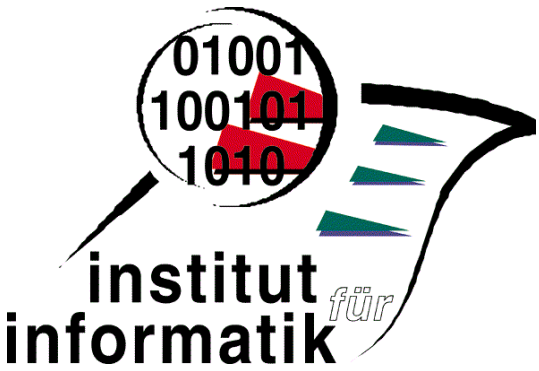
Universität Augsburg



Objektorientiertes Design eines Werkzeugs zum Effizienzvergleich asynchroner Systeme

Elmar Bihler

Report 2000-02 Januar 2000



Institut für Informatik
D – 86135 Augsburg

Copyright © **Elmar Bihler**

Institut für Informatik
Universität Augsburg
D-86135 Augsburg, Germany
http://www.Informatik.Uni-Augsburg.DE
— all rights reserved —

Objektorientiertes Design eines Werkzeugs zum Effizienzvergleich asynchroner Systeme

Elmar Bihler*

Institut für Informatik, Universität Augsburg
D-86135 Augsburg, Germany

Zusammenfassung

Das Werkzeug FastAsy implementiert Algorithmen, um durch Petri-Netze gegebene asynchrone Systeme zum Zweck der Effizienzmessung bezüglich einer Testhalbordnung zu vergleichen. Im Rahmen dieser Arbeit werden ein neues Klassendesign für FastAsy und die neu entwickelte graphische Oberfläche des Tools vorgestellt. Durch das neue Design soll FastAsy auf eine wesentlich größere Problemklasse anwendbar sein.

1 Einleitung

Anders als in synchronen Systemen können wir bei asynchronen Systemen, wie sie z.B. durch viele Petri-Netze modelliert werden, den Effizienzbegriff nicht an einer globalen Zeit festmachen. Versehen wir nämlich ein solches System einfach mit einer systemweiten Uhr, beobachten wir, daß ein Ablauf einerseits in Nullzeit ablaufen kann (d.h. jede Transition schaltet sofort), andererseits aber beliebig lang dauern kann, da den Transitionen ja zunächst keine Obergrenzen für die Zeit, die sie mit dem Schalten warten, gesetzt sind. Daraus wird sich schwerlich ein zeitlicher Effizienzbegriff herleiten lassen, denn diese best- und worst-case-Szenarien sind offenbar wenig aussagekräftig.

Die Idee ist nun, solche asynchronen Abläufe mit lokalen Zeitinformationen zu versehen. Aus Gründen der Entscheidbarkeit wird diese Zeitinformation in diskreten Einheiten gemessen, in allen bisher untersuchten Settings lies sich jedoch immer ein Resultat zeigen, das alle Aussagen, die daraus für konkrete Systeme gewonnen werden, wieder auf die Ebene kontinuierlicher Zeiteinheiten anhebt. In dieser Arbeit wollen wir uns auf die Weiterentwicklung eines Werkzeugs konzentrieren, das auf der Grundlage solcher Effizienzbegriffe asynchrone Systeme automatisch miteinander vergleicht. Für Einzelheiten zu den theoretischen Grundlagen werden wir nur auf die entsprechenden Arbeiten verweisen. Zu nennen wären etwa [Vog95], [JV96], [Vog96], [BV97], [Bih98].

Im Rahmen dieser Arbeiten wurden nun im Laufe der Zeit verschiedene Netzerweiterungen und dazugehörige Effizienzbegriffe untersucht, die viele zentrale Merkmale gemeinsam haben. So wurde etwa der Zeitbegriff immer durch eine Modifikation der normalen Schaltregel eingeführt, die Effizienzsemantiken basierten immer auf einer Art Testszenario und, was in unserem Zusammenhang natürlich besonders wichtig war, es entstanden

* Die vorliegende Arbeit wurde von der DFG im Rahmen des Projekts „Halbordnungstesten“ unterstützt.

jeweils entscheidbare Charakterisierungen der Testordnungen, die mittels einer Erweiterung des Erreichbarkeitsgraphen maschinell generierbar sind.
Schon ziemlich früh entstanden aus den oben angegebenen Arbeiten die ersten Versionen unseres Tools FastAsy, das zunächst für das in [JV96] untersuchte Modell für gegebene Petri-Netze die Semantiken berechnen und diese vergleichen konnte. Diese Versionen verfügten über ein einfaches Kommandozeilen-Interface, so daß zur graphischen Bearbeitung der Netze und auch zu deren Simulation auf das verbreitete PEP-Tool [PEP] zurückgegriffen wurde. Zu dieser Zeit war das Tool im großen und ganzen nur zur internen Verwendung am Lehrstuhl gedacht.
Nach einer Reihe von Detailverbesserungen sowohl der Funktionalität als auch der Implementierung und sehr nützlichen, aber eher peripheren Erweiterungen des Tools kamen nun in jüngerer Zeit einige Ideen auf, die über ein größeres Redesign nachdenken ließen:

- FastAsy sollte in Zukunft parallel mit verschiedenen (vorhandenen und zukünftigen) Versionen des Effizienzbegriffs umgehen können.
- Eng damit zusammen hängt die Forderung, verschiedene Klassen von zeitbehafteten Netzerweiterungen zu unterstützen. Damit geht allerdings eine zumindest teilweise Abwendung von PEP als Netzeditor und Simulator einher, denn dieses Tool ist von seiner Architektur her nicht dazu gedacht, proprietäre Erweiterungen in den Netzen zu verarbeiten.
- Aus diesem Grund wiederum scheint ein in FastAsy integrierter, einfach erweiterbarer Netzeditor ebenso wünschenswert wie ein zugehöriger Simulator für das Token-Game. Dafür ist aber offenbar eine graphische Benutzerschnittstelle (GUI) Voraussetzung. Nebenbei bemerkt hat die Einführung einer GUI natürlich auch den angenehmen Effekt, daß die Bedienung von FastAsy gerade für den gelegentlichen Benutzer (und um solche dürfte es sich in der Regel handeln) einfacher zugänglich wird. Da geplant ist, FastAsy etwa über das WWW interessierten Anwendern zur Verfügung zu stellen, gewinnt dieser Aspekt in der vorliegenden Version beträchtlich an Gewicht.

Wir gehen nun als erstes kurz auf die Idee asynchronen Testens ein und beschreiben die verwendeten Algorithmen in einer sehr abstrakten Darstellungsweise. Ein folgendes Kapitel führt am Beispiel FastAsy kurz aus, warum es sich bei dem Slogan, das objektorientierte Paradigma schütze vor der Notwendigkeit von Redesigns, allenfalls um ein verbreitetes Mißverständnis handelt. Nachdem wir kurz auf das Framework der Implementierung von FastAsy engegangen sind, werden wir nach und nach die zentralen Designentscheidungen entwickeln, zuerst in einem konzeptionellen, dann in einem eher technischen Teil, und zuletzt in einem Abschnitt, der sich der Gestaltung der GUI widmet. Ein Ausblick auf geplante Erweiterungen von FastAsy beschließt dann unsere Ausführungen.

2 Effizienztests

2.1 Vorbemerkungen

Im folgenden wollen wir versuchen, dem Leser auf einer sehr abstrakten Ebene sowohl die gemeinsamen Grundzüge als auch die Freiheitsgrade bei der Definiton unserer Effizienzbegriffe vor Augen zu führen.
Ausgangspunkt ist der Zeitbegriff, den man in das asynchrone System einführt. Bei Petri-Netzen bieten sich verschiedene „Vorgänge“ an, denen man eine Dauer zuordnen könnte. Dies könnten im einfachsten Fall das Schalten einer Transition, die Zeitdauer bis zur Aktivierung einer Transition oder auch das Fließen einer Marke über eine Kanten sein. Eng damit zusammen hängt die Entscheidung, welche Elemente des Netzes Träger der Zeitinformation sind, d.h. welche Elemente sozusagen Uhren mitführen. (Da es sich um asynchrone Systeme handelt, macht eine globale Zeitmessung schon per Definition keinen Sinn).
Damit nun aus solchen Zeitbegriffen eine Beschreibung zeitlichen Verhaltens werden kann, müssen den Systemen noch gewisse Restriktionen mitgegeben werden. Darunter verstehen wir Forderungen wie „jede Transition schaltet nach ihrer Aktivierung frühestens nach einer Zeiteinheit und innerhalb einer endlichen Zeitspanne“ oder auch „jede Transition schaltet nach ihrer Aktivierung innerhalb einer Zeiteinheit“. (In der Tat ist jede der beiden Aussagen für sich jeweils eine sinnvolle Annahme für ein asynchrones System, wobei allein die zweite erhoffen läßt, auf ein entscheidbares Effizienzkriterium zu führen.) Solche Restriktionen können global sein und damit implizit bleiben (wie etwa obige Beispiele), oder wir können individuelle Restriktionen als Inschriften entsprechender Elemente des Netzgraphen vorsehen.
Das Ergebnis dieser Definitionen wird in jedem Fall eine Erweiterung der Schaltregel sein, die neben dem funktionalen auch zeitliches Verhalten beschreibt. Im allgemeinen werden beide nicht orthogonal zueinander sein, siehe z.B. [Bih98], wo durch zeitliche Restriktionen unter Umständen funktionales Verhalten ausgeblendet werden kann. Diese erweiterte Verhaltensbeschreibung kombinieren wir nun jeweils mit einem Testbegriff. Dabei handelt es sich, stark vereinfacht, um eine (unendliche) Familie von Testnetzen, die von getesteten Netzen jeweils ein bestimmtes Verhalten innerhalb einer bestimmten Zeit fordern.
Ein Netz N_1 soll nun als effizienter als ein anderes N_2 gelten, falls N_1 mindestens alle Tests besteht, die auch N_2 besteht. Durch Herausgreifen geeigneter Testnetze ließ sich in vielen der untersuchten Settings zeigen, daß eine solche Testordnung äquivalent dazu charakterisiert werden kann, daß man die zeitbehafteten Sprache verfeinert und dann auf Inklusion testet. Diese verfeinerte Sprache meinen wir, wenn im folgenden allgemein von *Effizienzsemantik* die Rede ist. Um umständliche Formulierungen zu vermeiden, bezeichnen wir ihr zugehörige Objekte generell mit „*erweiterte* Markierung“, „...Schaltregel“, usw...
Wir erwarten also generell von einem Tool, das solche Effizienzvergleiche durchführt, folgende Schritte:

- Aufbau eines (nichtdeterministischen) Automaten, der die Effizienzsemantik repräsentiert, für jedes Netz
- Aufbau eines äquivalenten deterministischen Automaten
- Durchführung einer Vorwärtssimulation auf je zwei solchen Automaten

Wie bekannt ist, ist die Existenz einer Vorwärtssimulation auf deterministischen Automaten äquivalent zu einer Inklusion ihrer Sprachen. Im Fall eines Fehlschlags des Simulationsversuchs würden wir gerne einen weiteren Schritt durchführen:

- Ausgabe eines kritischen Elements, das nur in der einen Sprache enthalten ist

Dieses Element übernimmt die Aufgabe einer Begründung, wir erhalten also in der Tat nicht nur Aussagen der Form „N_1 ist schneller als N_2“, sondern sogar „N_1 ist schneller als N_2, weil N_2 folgendes langsame Verhalten hat ...“.

2.2 Die wichtigsten Algorithmen

Da im nachfolgenden Teil an den meisten Stellen nur noch der *Klassen*aspekt beleuchtet wird, wollen wir an dieser Stelle noch kurz die grundlegenden *Algorithmen* vorstellen, die bei den bisherigen Modellen zum Einsatz kommen und sich im Verhalten der Klassen wiederfinden.

2.2.1 Generierung des NDEA

Der Grundgedanke hinter diesem Algorithmus ist es, eine Liste von erweiterten Markierungen zu halten, von denen wir noch nicht alle Folgemarkierungen kennen. Am Anfang befindet sich in dieser Liste lediglich die Startmarkierung. Für jede Markierung, die wir aus dieser Liste nehmen, bestimmen wir alle möglichen Folgemarkierungen, tragen die Kanten zu diesen in unseren Graphen ein und stellen die neu erhaltenen Markierungen in die Liste. Wenn die Liste leer ist, haben wir alle erreichbaren Markierungen gefunden und der NDEA ist komplett aufgebaut.

2.2.2 Konstruktion des Potenzautomaten

Das Prinzip, um aus dem NDEA einen DEA mit gleicher Sprache zu erhalten, ist die bekannte Konstruktion des Potenzautomaten (siehe z.B. [HU94]).

2.2.3 Aufbau der Simulationsrelation

Eine (Vorwärts-)Simulation zwischen zwei deterministischen Automaten ist eine Relation zwischen den beiden Zustandsmengen, so daß die Startknoten in Relation stehen und für jedes Element (q,p) der Simulation und jeden Knoten q', den der erste Automat von q über eine Kante mit einer bestimmten Beschriftung erreichen kann, auch ein Knoten p' existiert, den der zweite Automat über eine gleichbeschriftete Kante erreichen kann, und (q',p') ebenfalls in Relation stehen.
Unser Algorithmus versucht nun, eine Simulation zu finden, indem eine Liste von noch nicht betrachteten Elementen der Simulation geführt wird, in der am Anfang nur das Paar der beiden Startzustände steht. Über obige induktive Definition ergänzen wir so lange Elemente in der Relation, wobei wir abgearbeitete Elemente aus der Liste entfernen und neu gefundene hinzufügen, bis die Liste leer ist. Tritt dabei die Situation auf, daß der zweite Automat einen bestimmten Übergang nicht durchführen kann, den der erste verlangt, so existiert keine Simulation und wir brechen die Konstruktion ab. Die bis dorthin konstruierte Relation wird uns dann beim Aufbau des Fehlerpfads nützlich sein. Zu diesem Zweck vermerken wir in Elementen der Simulation zusätzlich, welches Element der Vorgänger war.
Abschließend sei noch gesagt, daß zum Aufbau der Relation und dem Ableiten der Resultate eigentlich nur der zweite Automat unbedingt deterministisch sein muß. Der Algorithmus ist jedoch der Einfachkeit halber nur für zwei deterministische Automaten formuliert, denn wir sind ja immer an einem Test in beiden Richtungen interessiert. Wirklich aussagekräftig sind nämlich offenbar nur *echte* Inklusionen auf den RT-Semantiken, und um solche festzustellen, benötigen wir sowieso beidseitige Vergleiche.

2.2.4 Rekonstruktion des Fehlerpfads

Wir können aus dem Verweis auf ein Vorgängerelement in der Simulationsrelation leicht einen Fehlerpfad aufbauen, in dem wir vom Fehlerelement aus rückwärts diesen Verweisen folgen. Um daraus nun einen Pfad im NDEA zu erhalten, der zu diesem Fehlerpfad geführt haben könnte, müssen wir sowohl für jeden Potenzzustand geeignete Teilzustände finden als auch beim Übergang zum deterministischen Automaten verschwundene Lambdapfade expandieren. Auch dazu müssen wir wieder vom Ende der Sequenz her vorgehen:
Wir haben aus dem DEA eine Kante $e = Q_1 \rightarrow Q_2$ zwischen Potenzzuständen oder äquivalent dazu eine 'Kante' $\{q_1^1,...,q_1^n\} \rightarrow \{q_2^1,...,q_2^m\}$ zwischen Mengen von einfachen Zuständen. Desweiteren haben wir den Fehlerpfad schon rückwärts bis zu einem $q_2 \in \{q_2^1,...,q_2^m\}$ verfolgt. Gesucht ist nun ein $q_1 \in \{q_1^1,...,q_1^n\}$, so daß der Pfad $q_1 \rightarrow q_2$ bei der Konstruktion des Potenzautomaten zur Entstehung der Kante e beiträgt. Da bei der Bestimmung der Potenzzustände die gesamte Lambdahülle eines Einzelzustands eingeht, besteht $q_1 \rightarrow q_2$ im allgemeinen nicht nur aus einer einzelnen Kante. Nach Konvention beim Aufbau hat aber genau die 1. Kante im Weg $q_1 \rightarrow q_2$ eine Beschriftung $\neq \lambda$.

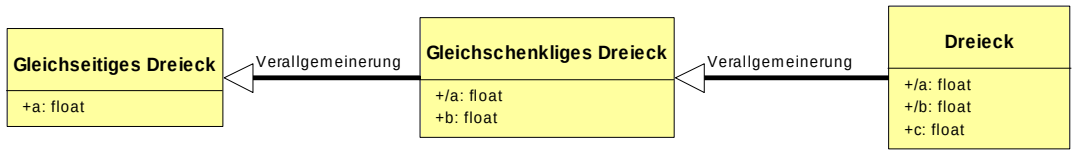
3 Notwendigkeit von Redesigns

Wir haben uns auf Grund der in der Einleitung formulierten Forderungen entschieden, statt einer Evolution des Designs ein komplettes Redesign durchzuführen. Nun war aber ja das Design der älteren FastAsy-Versionen bereits objektorientiert, siehe dazu [Bih98]; insofern könnte man die Forderung stellen, die Erweiterungen sollten sich möglichst auf der Basis des alten Quellcodes durchführen lassen – im Idealfall ja sogar ohne Veränderungen an diesem, sondern etwa durch Ableiten neuer Klassen mit den erweiterten Funktionalitäten. Es ist jedoch kein Geheimnis, daß solche Forderungen an das objektorientierte Paradigma an sich schon unrealistisch wären:

„Wie gut oder schlecht dieses Versprechen [nämlich die leichte Änderbarkeit] von objektorientiert entwickelten Programmen eingelöst wird, hängt nach unserer Erfahrung stark davon ab, bis zu welchem Grad künftige Änderungsanforderungen bereits in Analyse und Design abschätzbar waren.“ [BW99]

Das grundsätzliche Problem, das dabei auftritt, läßt sich folgendermaßen formulieren: Vererbungstechniken sind in der Praxis meist ungeeignet, im nachhinein eine Erhöhung des Abstraktionsniveaus zu erreichen. Dies liegt daran, daß in einer Programmiersprache wie C++ jedes Objekt einer abgeleiteten Klasse durch Rückwärtskonversion auch als Objekt jeder übergeordneten Klasse verwendet werden kann. Mit anderen Worten wird auf der formalen Ebene schon eine ISA-Semantik der Vererbung vorweggenommen.

Man sieht das Problem an einem Beispiel, das sich in manchen Büchern zur OOP exemplarisch für eine Art „Generalisierungsvererbung" findet:



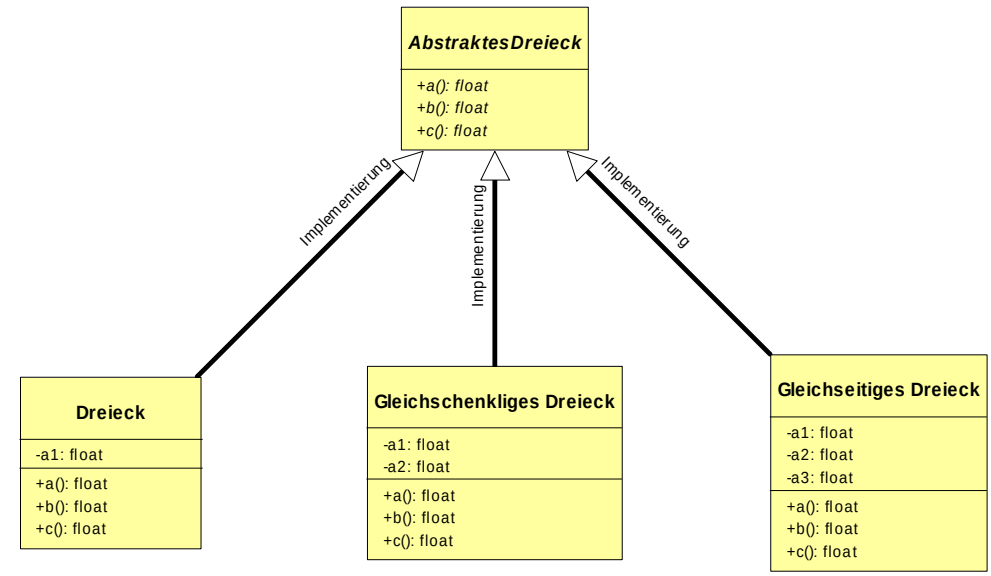
Entgegen der konzeptionellen Hierarchie

„Dreieck **ISA** gleichschenkliges Dreieck **ISA** gleichseitiges Dreieck“

wird hier auf der formalen Ebene vom gleichseitigen bis hinunter zum allgemeinen Dreieck vererbt. Informell ausgedrückt ist die Hoffnung also, durch Hinzunahme von Attributen eine größere Klasse von Objekten zu erfassen.

Wie man hier sieht, führt dies aber sofort zu formalen Schwierigkeiten, denn die Schnittstelle der ursprünglichen Klasse „Gleichseitiges Dreieck“ wurde nicht auf einem geeigneten Abstraktionsniveau definiert, so daß eine Aufwärtskonversion hier einfach zu einer fehlerhaften Interpretation des Objekts führt.

Unter Beibehaltung der Vorteile des obigen Modells (nämlich der Speicherplatzersparnis) hätte man stattdessen korrekterweise modellieren können:



Hier wurden die Attribute als *private* gekennzeichnet und damit aus der Schnittstelle herausgenommen. Die Aufgabe des Zugriffs übernehmen die Methoden a, b und c. Fehlerhafte Konversionen sind hier nicht möglich.

Die Alternative zu einer Weiterentwicklung der Klassenhierarchie durch Vererbung ist die *Transformation* von Klassen. Dabei werden Ausschnitte einer Klassenhierarchie, deren Abstraktion als nicht ausreichend identifiziert wurde, durch korrektheitserhaltende Umwandlungen in mehreren Schritten in eine Form überführt, die nach außen hin das gleiche leistet, von deren Abstraktionsniveau aus aber die gewünschten Änderungen erreichbar sind. Nachteile dieser Methode sind:

- Transformationen betreffen im allgemeinen nicht nur die Schnittstellen von Klassen, denn mitunter kann die mangelnde Abstraktion genau in den Schnittstellen begründet sein (s.o.). Die Umformungen lassen sich also nicht rein lokal durchführen, sondern betreffen mehrere Klassen auf einmal.
- Es existiert momentan kaum Unterstützung durch formale Methoden oder auch Werkzeuge, die die Korrektheit der Umformungen sicherstellen könnte, so daß diese in der Praxis „frei Hand“ durchgeführt werden. Es ist offensichtlich, daß dabei, gerade im Zusammenspiel mit dem ersten Punkt, oft die Integrität des Quelltextes leidet.

Diese Punkte lassen die Technik der Transformation bestenfalls für Projekte mit sehr einfach strukturierten Klassenhierarchien (etwa transaktionsorientierte Geschäftsapplikationen) interessant erscheinen.

Generell gibt es also keinen Ersatz dafür, schon in der ersten Designphase eines Evolutionszyklus einen ausreichend hohen Abstraktionsgrad zu etablieren. Allerdings ist aus naheliegenden Gründen bei einem Entwurf auch darauf zu achten, daß der Abstraktionsgrad nicht zu sehr ansteigt („over-engineering“):

- Abstraktion kostet Zeit in der Analyse-, Design- und Implementierungsphase.
- Übermäßige Abstraktion birgt das Risiko, daß das Ergebnis der Implementierung auf Grund zu hoher Komplexität unbeherrschbar wird.
- Höhere Abstraktion kostet zur Laufzeit durch Anwachsen des Overhead Effizienz.
- Abstraktion verträgt sich nicht immer mit „datennahen“ Optimierungen, so daß auch hier Effizienz verlorengehen kann. Die theoretische Möglichkeit, durch weitere Kapselung der betroffenen Stellen die Optimierungen objektorientiert „sauber“ unterbringen zu können, führt in der Praxis manchmal nur in verschärfter Form wieder auf die obigen Probleme.

Dies alles führt bei jedem Projekt zur Festlegung eines „Änderungshorizonts“, in den absehbare Änderungswünsche gemäß der Wahrscheinlichkeit ihrer späteren Realisierung einfließen sollten. Ein späteres Durchbrechen dieser Grenze wirkt sich fast immer in einem sprunghaften Anstieg des Aufwands bei der Fortentwicklung aus. Wir werden später noch sehen, wie diese Festlegung für die aktuelle Version von FastAsy aussieht.

4 Die Implementierungsumgebung

Ein komplettes Neudesign, an das sich ja notgedrungen eine Neuimplementierung anschließt, bietet immer auch die Chance, die bestehende Plattform des Projektes zu überdenken. Bei FastAsy haben sich allerdings in dieser Richtung die meisten Annahmen, von denen schon früher ausgegangen wurde, erfüllt, so daß nur wenig Notwendigkeit für Änderungen etwa gegenüber [Bih98] bestand. Wir fassen die wichtigsten Fakten zusammen:

4.1 Sprache

Als Programmiersprache kommt nach wie vor C++ zum Einsatz, da diese Sprache unserer Meinung nach immer noch am besten hervorragende Effizienz mit einer hinreichenden Unterstützung von objektorientierten Techniken vereinigt.

4.2 Bibliotheken

Da die Standardisierung der Standard Template Library (STL) in der letzten Zeit erfreuliche Fortschritte gemacht hat und insbesondere ja mittlerweile als Teil der ANSI-Spezifikation in vielen Compilern umgesetzt ist, haben wir uns entschieden, wann immer möglich die entsprechenden Teile der STL einer proprietären Implementierung von Containerklassen vorzuziehen.

4.3 Plattform

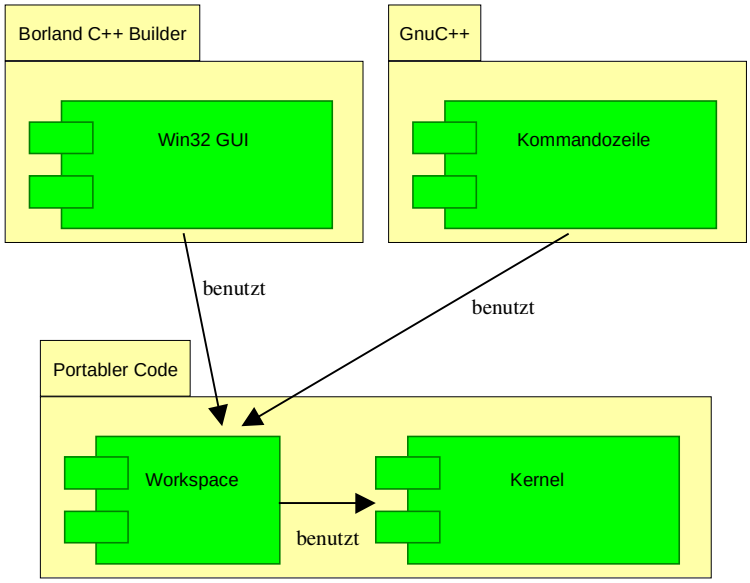
Im Gegensatz zu früheren Versionen erhält die Frage der Portierbarkeit durch die graphische Oberfläche nun eine neue Qualität. Schon allein aus dem Grund, daß für FastAsy nur sehr begrenzte personelle Ressourcen zur Verfügung stehen (die Realisierung von FastAsy ist immer noch ein 1-Mann-Projekt), mußte dabei jedoch ein Kompromiß eingegangen werden:
Die Quelltextmodule für die Kernfunktionalität von FastAsy wurden vollständig von der Benutzeroberfläche getrennt (s.u. zur Verteilungsstruktur). Bei diesen ersteren Bereichen des Quellcodes wurden konsequent nur ANSI-konforme Schreibweisen eingesetzt, parallel zur Implementierung häufige Portabilitätstests durchgeführt und in seltenen Fällen auch Präprozessordirektiven zur bedingten Übersetzung verwendet. Dadurch wurde in jeder Phase der Implementierung sichergestellt, daß eine korrekte Übersetzung auf allen verwendeten Compilern (s.u.) möglich blieb.
Die GUI hingegen wurde plattformspezifisch unter Verwednung der Visual Component Library (VCL) von Borland erstellt. Damit ist die GUI auf ein 32-Bit-Windows (Win95/98, WinNT oder Win2000) als Betriebssystem festgelegt.
Als Compiler kamen dabei zum Einsatz:

- Gnu-C++ in der Version egcs-2.91.57
- Sybase Power++ 2.1/2.5 Professional
- Borland C++-Builder 4.0 Professional Patch Level 2

Dabei schied allerdings im Laufe der Projektrealisierung der zweitgenannte Compiler aus und wurde durch das Borland-Produkt ersetzt, da sich herausstellte, daß er, im Gegensatz zu den Angaben des Herstellers, weit hinter den Anforderungen der ANSI-Spezifikation zurückblieb. Insbesondere die fehlende Fähigkeit, Default-Parameter bei Template-Argumenten zu verwenden, schränkte den sinnvollen Gebrauch der STL empfindlich ein.

4.4 Verteilungsstruktur

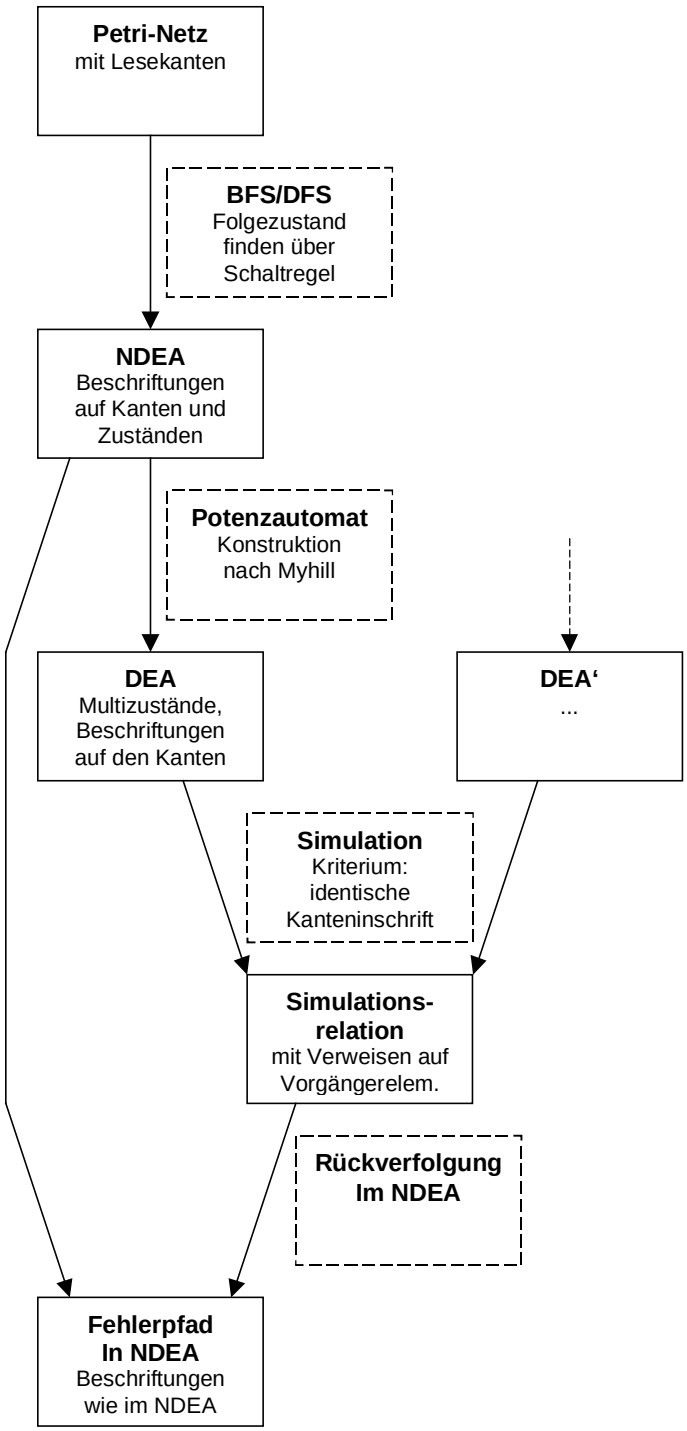
Bei der Realisierung der Schichtenteilung mußte ebenfalls aus Gründen des Aufwands darauf verzichtet werden, die Verteilung durch einen Mechanismus wie den (nicht unproblematischen) Einsatz von compilerübergreifendem dynamischem Linken oder gar COM/DCOM zu realisieren. Stattdessen findet die Anbindung tieferer Schichten tatsächlich durch Einbinden des Quelltexts statt. Durch die überraschend hohe ANSI-Konformität der Gnu- und Borland-Compiler sowie der entsprechenden STL-Implementierungen blieb dabei die Anwendung von bedingtem Code auf sehr wenige Stellen beschränkt.



Die oberen beiden Module beinhalten dabei den Code zur wirklichen Benutzerinteraktion, im Fall von *Win32 GUI* also Verwaltung von Fenstern, Controls, sowie die Verarbeitung von Windows-Botschaften.
Im Modul *Workspace* sind solche Aspekte der Benutzerschnittstelle gekapselt, die sich nicht für die Ausprägung als graphisches- oder Kommandozeileninterface unterscheiden. Auf die Vorteile dieser zusätzlichen Kapselungsschicht soll weiter unten noch die eingegangen werden.
Das Kommandozeileninterface wird momentan nicht in einer für den realen Anwendungsfall geeigneten Version unterstützt, sondern dient lediglich Testzwecken, um den Quelltext weiterhin in einer portablen Version halten (und die Portabilität auch testen) zu können.

5 Designentscheidungen

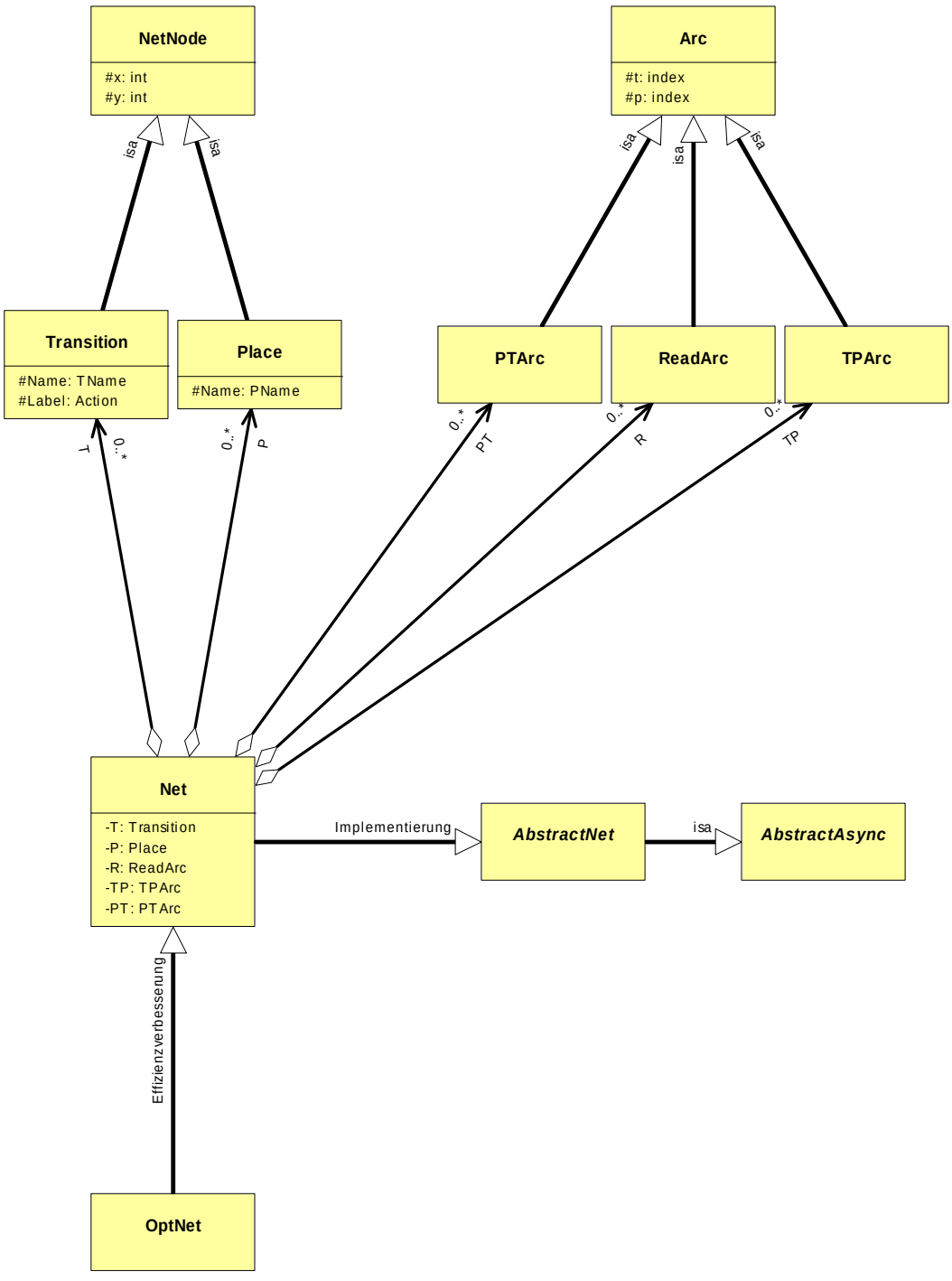
Wie wir weiter oben bereits gesehen haben, kommt es gerade bei einem Projekt, das von Haus aus auf Erweiterbarkeit angelegt sein soll, auf die passende Wahl des Abstraktionsniveaus an. Bei einem Projekt wie FastAsy, wo die genaue Form der Erweiterungen noch nicht einmal vorausgesagt werden kann, da sie sich zu einem hohen Maße aus der Weiterentwicklung der unterliegenden Theorie ergibt, wird diese Wahl zum Dreh- und Angelpunkt des Designs. Als Ausgangspunkt des Designs, von dem es zu abstrahieren gilt, nehmen wir die Grundidee von FastAsy, wie sie vorher schon beschrieben wurde:



Wir identifizieren nun in den folgenden Abschnitten Stellen, an denen mit hoher Wahrscheinlichkeit Erweiterungen notwendig werden. Wir bedienen uns dabei teilweise zur Veranschaulichung einer UML-Notation, wobei wir generell nur Ausschnitte aus den Klassen angeben, um nicht unnötig ins Detail zu gehen.

5.1.1 Petri-Netze

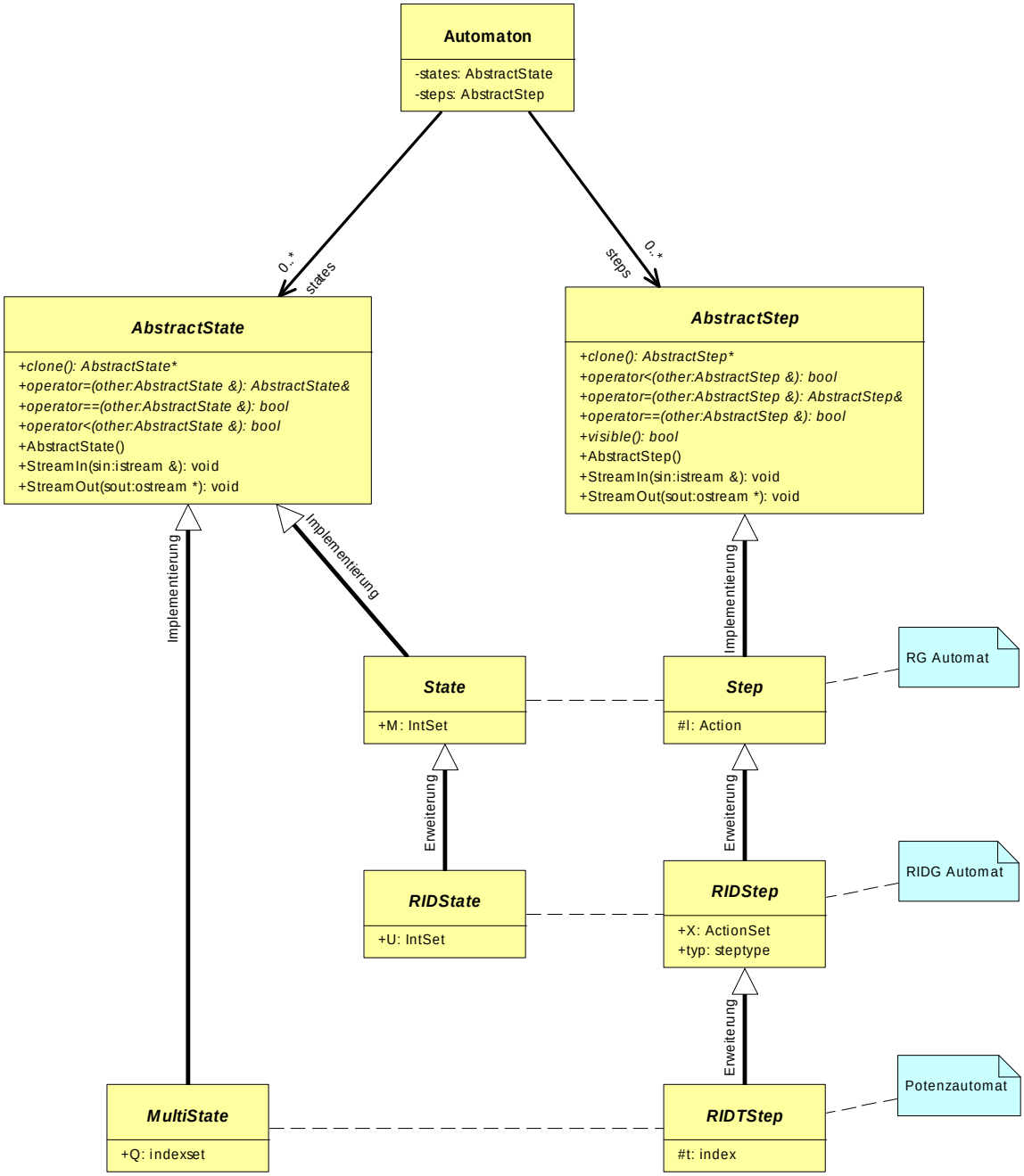
Wie weiter oben schon umrissen, werden mit Sicherheit verschiedene Netzklassen auftreten. Insbesondere können etwa zusätzliche Beschriftungen in der Form von Zeitinformationen hinzukommen. In einem ersten Abstraktionsschritt könnten wir also vorsehen, die verschiedenen Objekte eines Netzgraphen mit Inschriften zu versehen.
Allerdings scheint es (wie das Beispiel der Lesekanten zeigt) auch möglich – wenn auch unwahrscheinlicher – daß sich der Aufbau des Netzgraphen selbst strukturell ändert. Letztendlich wäre es in der letzten Abstraktionsstufe denkbar, daß man konkurrierende Modelle (etwa Automaten oder Terme von Prozessalgebren) untersuchen möchte.
Unsere Klassenhierarchie soll nun allen Möglichkeiten gerecht werden, wobei der Aufwand der wahrscheinlichsten Änderung natürlich am geringsten ausfallen soll. Betrachten wir dazu folgenden Auszug aus der UML-Dokumentation von FastAsy:



Dabei muß ein Interface von *AbstractAsync* lediglich durch die Standardmethoden gewährleisten, daß abgeleitete Klassen unter einem gemeinsamen Begriff angesprochen werden können. *AbstractNet* stellt dann bereits gewisse elementare Methoden zur Verfügung, etwa Bestimmung aller Transitionen, Stellen oder allgemeiner auch nur der Netzknoten. Sie sind in *AbstractNet* aber noch abstrakt. Interessant ist diese Schnittstelle vor allem, weil ein erweiterbarer Netzeditor sie später benutzen könnte.
Für bloße Erweiterungen der Inschriften genügt es nun, von den Klassen, die Elemente des Netzgraphen repräsentieren, weitere Klassen abzuleiten. Da die Netzklasse selbst nur Aufgaben, die sich auf den Netzgraphen beziehen, übernimmt, müssen hier auch keine Veränderungen durchgeführt werden.
Soll dagegen die Struktur des Netzes selbst erweitert werde, ist eine Ableitung direkt von *AbstractNet* (oder natürlich eine weitere Ableitung von Net) und *NetNode* bzw. *Arc* vorgesehen.
Man beachte, daß die in der Implementierung vorgenommene Ableitung von *Net* zu *OptNet* zum Zwecke der Effizienzverbesserung vorgenommen wird. *OptNet* behält das Interface von *Net* vollständig bei, implementiert jedoch eine Art Caching für Anfragen nach Pre- oder Postsets, indem es bereits angefragte Ergebnisse speichert und so mehrfache Berechnungen vermeidet.
Für den allgemeinsten Fall, der vorgesehen ist, muß direkt von *AbstractAsync* abgeleitet werden. Es mag an dieser Stelle verwundern, daß *AbstractAsync* keine Methoden zur Verfügung stellt, die in irgendeiner Art und Weise für asynchrone Systeme spezifisch wären. Dies liegt einfach daran, daß solche Dienste sowieso nur an einer Stelle in Anspruch genommen werden: Bei der Erstellung des NDEA. Wir wollen aber von der Erstellung des NDEA sowieso abstrahieren, indem wir die Regeln zu seinem Aufbau auslagern. (Wir werden dieses ausgelagerte Verhalten weiter unten einen Generator nennen.) Ein solcher Generator ist jedoch immer spezifisch für einen konkretes Effizienzmodell und damit auch spezifisch für die konkrete Spezialisierung von *AbstractAsync*.

5.1.2 Der verallgemeinerte Automat

Wir stellen fest, daß sich mit Einführung einer neuen Netzklasse und/oder Schaltregel die Art der mit dem NDEA assoziierten Daten, also die Inschriften der Kanten und Zustände, ändern kann. Keinesfalls wollen wir aber für jeden dieser Fälle eine neue Klasse für den Automaten schreiben müssen, so daß wir festhalten können: Wir müssen nicht nur von den Regeln zur Konstruktion, sondern auch von den Beschriftungen des NDEA abstrahieren. (Betrachten wir dann allerdings unser ursprüngliches Schema, stellen wir fest, daß wir mit dieser Abstraktionsstufe auch schon die Erstellung des DEA und seine Inhalte subsumiert haben, die Myhill-Konstruktion also ebenfalls eine Spezialisierung eines Generators ist. Wir werden dies später ausnutzen.)
Nun sollte die Lösung in diesem Falle allerdings nicht daraus bestehen, nur die Schnittstelle eines Automaten als abstrakte Klasse zu spezifizieren, und dann doch wieder für jede Art Automat „das Rad neu zu erfinden“, denn dies wäre bei späteren Erweiterungen wohl eher eine Einschränkung als ein Vorteil.
Stattdessen modellieren wir all diese Automaten mit einer einzigen, festbleibenden Klasse. Es besteht keine Notwendigkeit, von dieser Klasse abzuleiten, denn die notwendige Flexibilität wird anders erreicht: Der Automat selbst braucht ja keinerlei Kenntnis von den Daten zu haben, die in seinen Kanten und Zuständen stehen. Er ist stattdessen nur für die Verwaltung der Struktur zuständig. Dazu führen wir zwei abstrakte Klassen ein, von der sich später die spezielleren Versionen von Zuständen bzw. Kanten ableiten. Dafür reicht ein fast minimales Interface aus: Diese Objekte müssen lediglich kopiert und auf Gleichheit geprüft werden können, außerdem noch einen Test auf kleiner-als enthalten, denn diesen benutzt die STL ggf., um Objekte zu sortieren. Hinzu kommen Methoden zur Ein- und Ausgabe, wobei wir weiter unten noch genauer beschreiben werden, wie der Automat im einzelnen Zugriff auf die spezialisierten E/A-Operationen erhält. Wie stellen außerdem noch fest, daß allen Kanten in unserem Setting noch eine weitere Gemeinsamkeit zu eigen ist: Sie modellieren ein bestimmtes Verhalten des asynchronen Systems, das untersucht werden soll, und insofern ist es schon auf dieser Abstraktionsstufe sinnvoll, internes (d.h. unsichtbares) und externes (d.h. sichtbares) Verhalten zu trennen. Dies geschieht durch eine weitere abstrakte Methode, die wir zum Interface von *AbstractStep* hinzufügen.
Sehen wir uns hierzu wiederum das UML-Schema an, indem dies dargestellt ist, zusammen mit den Spezialisierungen der Klassen, die genau der am Anfang des Abschnitts angegebenen Grundfunktionalität entsprechen. Die internen Strukturen des Automaten zur Verwaltung der Beziehung zwischen Zuständen und Kanten wurden dabei der Klarheit halber unterdrückt. Diese Verwaltung erfolgt unter Zuhilfenahme der STL und ist völlig geradlinig durch Listenstrukturen realisiert.

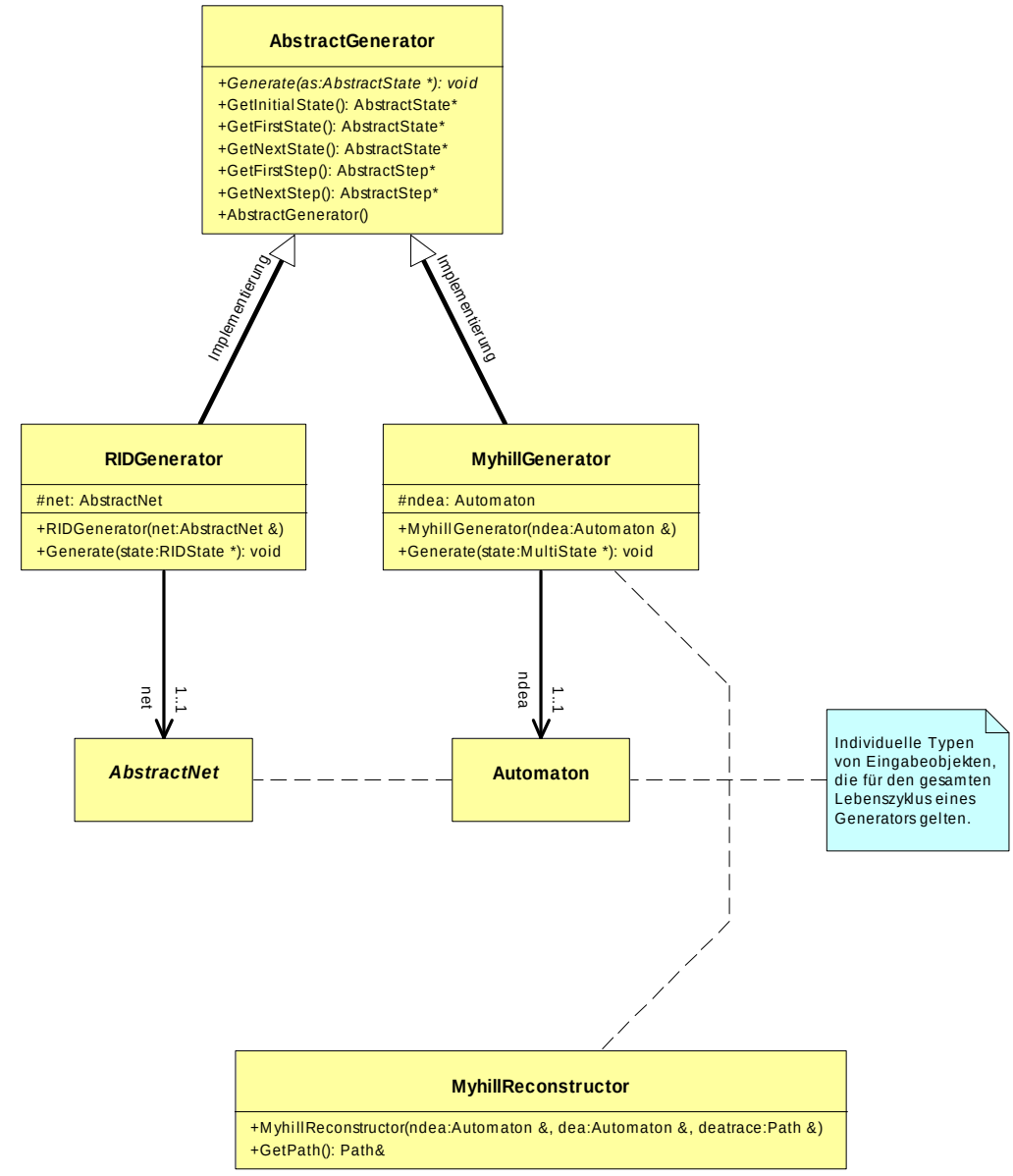


Dem Leser wird nun allerdings nicht entgangen sein, daß der Automat nach unserer bisherigen Beschreibung zwar vieles kann, verblüffenderweise eines jedoch nicht: Er kann weder neue Zustände noch neue Kanten erzeugen, denn woher soll der Automat wissen, welchen Konstruktor er aufrufen soll, wo wir ihn doch auf den Umgang mit den abstrakten Klassen beschränkt haben. Man wäre nun vielleicht versucht, dem Automaten zwei abstrakte Methoden zum Generieren der jeweils passenden Objekte mitzugeben, die dann in abgeleiteten, instanzierbaren Automatenklassen definiert werden. Die Verwendung des Generatorenkonzepts aus dem folgenden Abschnitt liefert uns allerdings noch eine einfachere Lösung.

5.1.3 Generatoren im Detail

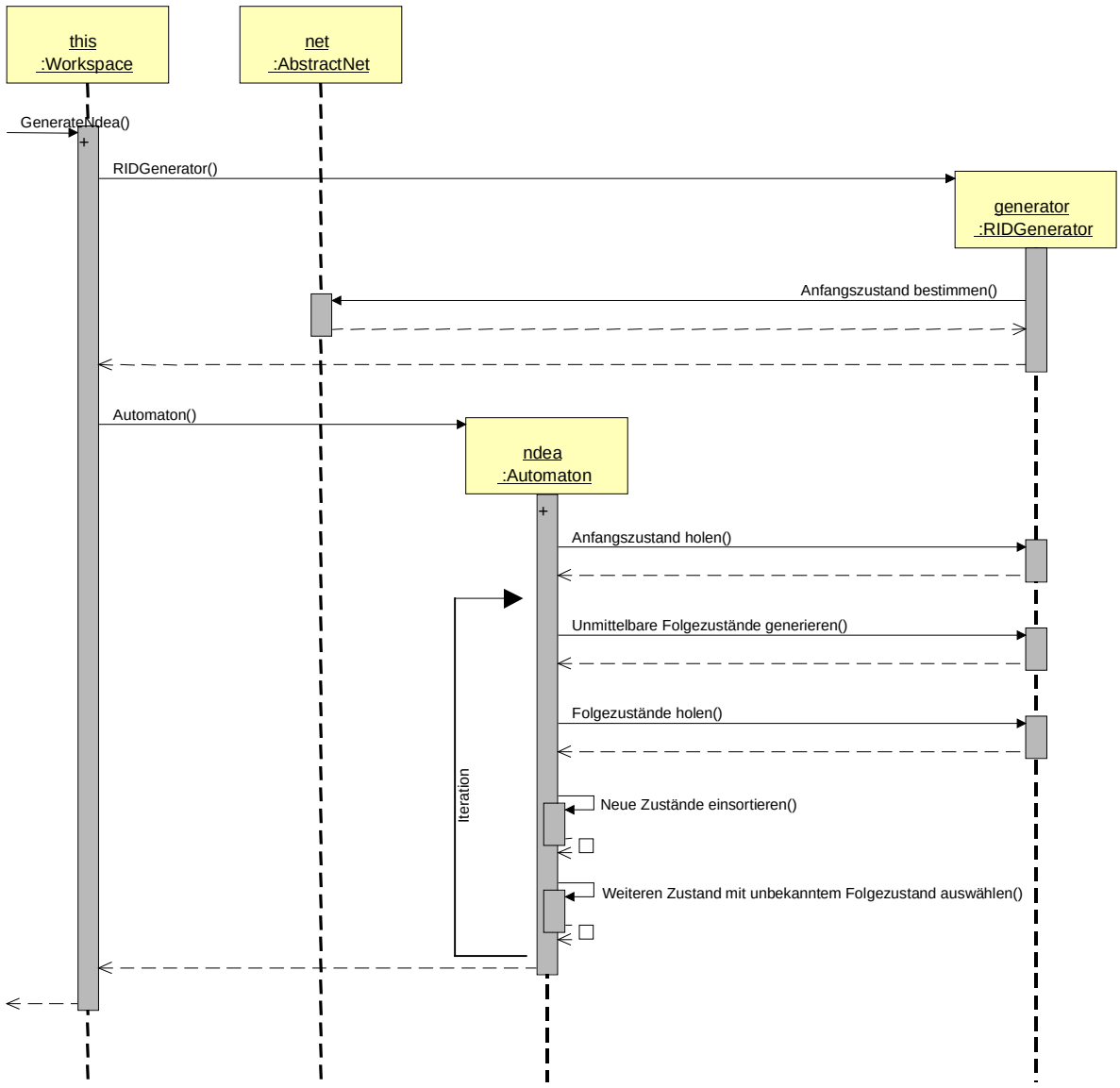
Wir haben bereits entschieden, die Vorschriften zum Erstellen von verschiedenen Automaten in Generatoren-Klassen zu kapseln. Diese Generatoren wiederum besitzen aber genaue Kenntnis von den Daten, die sie in den Zuständen und Kanten speichern sollen. Also liegt es nahe, diese Generatoren die entsprechenden Objekte erzeugen (und deren Zustand initialisieren) zu lassen. Zu beachten ist, daß diese erzeugten Objekte im allgemeinen eine längere Lebensdauer haben als ihr Erzeuger, denn sie gehen nach ihrer Erzeugung in den Besitz des Automaten über. Konstrukte wie diese, die Objekte speziellerer Typen erzeugen und einer allgemeineren Verarbeitung zuführen, nennt der OOP-Jargon bezeichnenderweise auch „Fabriken“.

Wir geben zunächst wieder ein Klassendiagramm an, das die Schnittstelle eines Generators zeigt, zusammen mit Spezialisierungen für die diskutierte Erstellung eines erweiterten Erreichbarkeitsgraphen sowie für die Potenzautomatenkonstruktion. Eine weitere Klasse names *MyhillReconstructor* sei hier der Vollständigkeit halber schon einmal erwähnt. Sie ist eng mit *MyhillGenerator* assoziiert und wird uns dazu dienen, aus Pfaden im Potenzautomaten wieder die ursprünglichen Pfade im NDEA zu rekonstruieren.
Man sieht hier, was im Abschnitt über Netzklassen schon vorweggenommen wurde, nämlich daß die Schnittstelle des *AbstractGenerator* nur auf der Abstraktionsebene des Automaten, also ohne Kenntnis über den Typ der Vorlage für die Erstellungsvorschrift, spezifiziert ist. Erst die Spezialisierungen verlangen im Konstruktor die Übergabe eines speziellen Vorlagentyps, etwa eines Netzes.
Zu erwähnen ist noch, daß die *GetFirst/NextState/Step*-Methoden nicht nur ein gemeinsames Interface aufweisen, sondern tatsächlich bereits in *AbstractGenerator* implementiert sind. Diese Methoden verwendet der Automat, um nach erfolgtem *Generate()* die Folgezustände und die Kanten dahin nacheinander anzufordern (s.u.). Die (internen) Datenstrukturen, in denen ein Generator die Folgezustände zwischenspeichert, sind nämlich schon in *AbstractGenerator* vorhanden und werden von den jeweiligen Spezialisierungen nur noch gefüllt, was es hier ermöglicht, durch eine gemeinsame Implementierung den Aufwand zu verringern.



Nun haben wir uns noch die Frage zu stellen, wie das Zusammenspiel zwischen Automat und Generator aussehen soll. Die einfachste Spielart wäre wohl, einem Generator den Automaten, den er füllen soll und das Objekt, das die Vorlage dazu ist, bekannt zu machen. Dabei hätte man jedoch immer noch einen höheren Aufwand bei der Implementierung eines konkreten Generators, als einem vielleicht lieb ist, denn man hat noch nicht ausgenutzt, daß das Vorgehen zur Erzeugung im Prinzip immer dasselbe ist: Von einem Anfangszustand

ausgehend werden so lange für alle Zustände, deren Folgezustände noch nicht vollständig bekannt sind, eben diese Folgezustände ermittelt und neu eingefügt, bis der volle Automat erstellt ist. (Sicherzustellen, daß dieses Verfahren ein endliches Ergebnis liefert und daher terminiert, ist Sache der unterliegenden Theorie.) Also brauchen Generatoren keine komplette Vorschrift zum Aufbau eines Automaten, sondern lediglich eine induktive Regel, nach der sich aus einem gegebenen Zustand alle Folgezustände (und natürlich die Kanten dorthin) berechnen. In der Tat liegen die Resultate, die die Theorie liefern, auch genau in dieser Form vor, nämlich als Schaltregeln.
Aufgabe des Automaten (und somit nur ein einziges Mal zu implementieren) ist es nun, den Generator zunächst mit dem Anfangszustand aufzurufen, die zurückgegebenen Folgezustände samt Kanten zu übernehmen (und, am Rande bemerkt, Duplikate von Zuständen zu verwerfen) und wie oben beschrieben bis zur Abbruchbedingung fortzufahren.
Ein UML-Sequenzdiagramm soll diesen Zusammenhang noch einmal verdeutlichen, hier in der Spezialisierung für den Aufbau des NDEA:



Auf ein weiteres Detail in diesem Schema sei noch hingewiesen: Woher kennt der Automat den Anfangszustand, wenn er doch nach oben gesagtem mit Beschriftungen seiner Objekte gar nichts anzufangen weiß? Die Antwort mag überraschen, aber diesen Zustand teilt ihm wiederum der Generator mit. Bei näherem Hinsehen scheint es auch nur vernünftig, dem Generator diese zweite Aufgabe zu übertragen, denn es handelt sich ja dabei um die Modellierung einer induktiven Definition, und zu dieser gehört nun einmal auch der Induktionsanfang. Auch technisch gesehen sind mit dieser Aufgabenverteilung keine Probleme zu erwarten, denn der Generator muß mit dem zu Grunde liegenden Petri-Netz (oder dem allgemeineren Objekt) schließlich bei der Bestimmung der

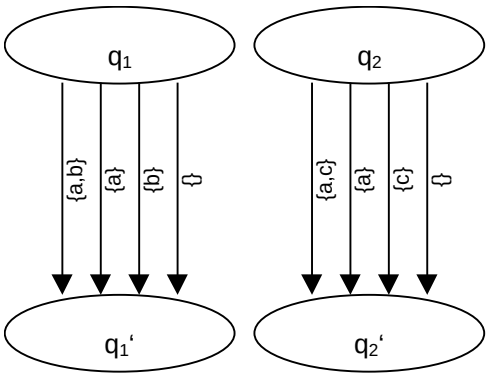
Folgezustände schon in allen Details umgehen können, also sollte ihm die Bestimmung des Anfangszustands nicht schwerfallen*.

5.1.4 Einbettung der Potenzautomatenkonstruktion

Dankenswerter Weise liegt auch der Algorithmus für die Konstruktion des Potenzautomaten nach Myhill genau in Form einer iterativen Vorschrift vor, welche nämlich besagt: Ein Multizustand Q‘ ist dann Nachfolger eines Multizustands Q unter der Kantenbeschriftung ε, wenn Q‘ nicht leer und genau die Menge der Nachfolger eines $q \in Q$ unter ε. Dies ist sehr hilfreich, da mit dieser Regel somit auch für unsere verfeinerte Modellierung des Generators kein Ausnahmefall vorliegt.
Es stellt sich nun noch die Frage, ob sich auch für den DEA Erweiterungen finden, die ausnutzen, daß von der Automatenkonstruktion abstrahiert wird. Es sei vorweggenommen, daß sich sehr wohl eine Anwendung finden läßt, nämlich für eine konzeptionelle Optimierung der Darstellung des DEA.
Um diesen Anwendungsfall darzulegen, läßt es sich allerdings nicht vermeiden, ein wenig weiter auszuholen. Wir wollen dazu den konkreten Fall derjenigen erweiterten Schaltregel betrachten, die im ursprünglichen FastAsy implementiert war und die wir hier in der für Petri-Netze üblichen Notation angeben:

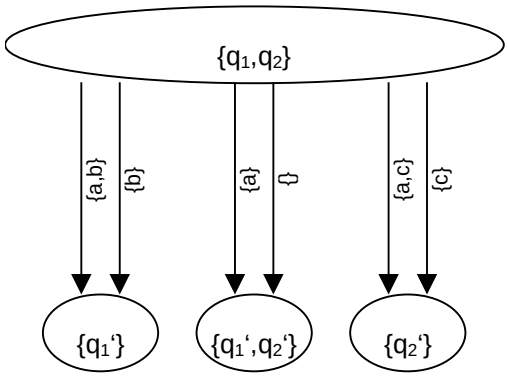
> Sei N ein Petri-Netz mit *Anfangsmarkierung* M_N, *Transitionen* T und einer *Beschriftung* $l:T \to \Sigma$, wobei Σ ein Alphabet von *Aktionen* ist. Wir nehmen o.B.d.A. Σ als endlich an.
> Eine *erweiterte Markierung* (M,U) besteht aus einer Markierung M und einer Menge U von *dringenden* Transitionen. Die *erweiterte Anfangsmarkierung* von N ist $(M_N, \{ t \mid M_N[t\rangle \})$.
> Es gilt $(M,U)\ [\varepsilon\rangle_r\ (M',U')$, falls einer der folgenden Fälle eintritt:
> 1. $\varepsilon = t \in T,\ M[t\rangle M',\ U' = U - \{ t' \mid {}^\bullet t \cap ({}^\bullet t' \cup {}^\wedge t') \neq \emptyset \}$
> 2. $\varepsilon = X \subseteq \Sigma,\ M = M',\ U' = \{ t \mid M[t\rangle \},\ \forall t \in U: l(t) \notin X \cup \{\lambda\}$
>
> Dabei nennen wir solche Mengen X *Verweigerungsmengen*. Wir erweitern dies auf Schaltfolgen und durch l(X):=X auf Sequenzen von Aktionen, die sogenannten *Refusal-traces*.

Ohne tiefer auf die Bedeutung der Schaltregel eingehen zu wollen, sieht man ganz lokal an der Definition von U‘ in 2., daß, wann immer $ID\ [X\rangle_r\ ID'$ gilt, für $Y \subseteq X$ auch $ID\ [Y\rangle_r\ ID'$ gelten muß.
Durch diesen Untermengenabschluß steigt die Zahl der Kanten im NDEA natürlich enorm an, so daß die Idee naheliegt, diesen Untermengenabschluß implizit zu lassen und nur durch ihr maximales Element zu repräsentieren. Die Umsetzung dieser Idee bereitet auf der Ebene des NDEA noch keine Schwierigkeiten, denn dort liegt diese Idee ganz gekapselt der entsprechenden Generator-Klasse.
Bei der Erstellung des Potenzautomaten wird aber der normale Algorithmus von Myhill unbrauchbar, denn dieser erkennt nun im allgemeinen nicht mehr, wann er eine durch eine (implizite) Untermenge beschriftete Kante mit einer anderen zusammenfassen müßte. Ein simples Beispiel mag dies verdeutlichen. Nehmen wir zunächst einmal folgenden Ausschnitt eines NDEA an:

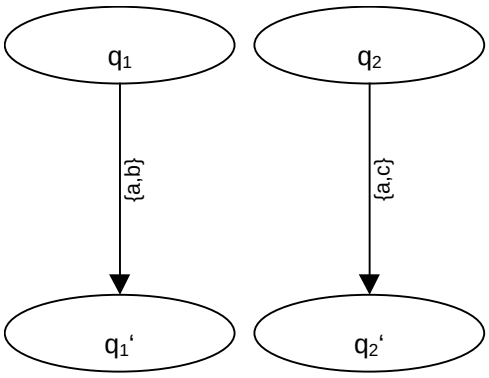


Nehmen wir weiterhin an, daß es im zugehörigen Potenzautomaten einen Multizustand $\{q_1,q_2\}$ gibt, dann liefert der Algorithmus von Myhill die Folgezustände:

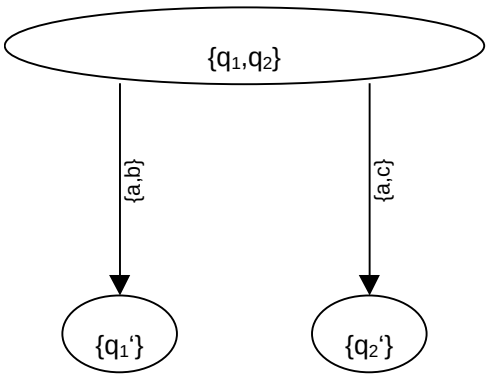
* Es sei die grundsätzliche Bemerkung erlaubt, daß ich eine derartig blumige Diktion in einem solchen Kontext nicht nur für legitim halte, sondern es als herausragenden Vorzug der objektorientierten Vorgehensweise sehe, daß solche Beschreibungen sich zuweilen als höchst hilfreich bei Analyse und Design erweisen.



Nun betrachten wir einen NDEA, bei dessen Erzeugung genau obige Optimierung angewandt wurde:



Der normale Algorithmus von Myhill liefert uns nun allerdings ein falsches Ergebnis, denn er interpretiert die Kantenbeschriftungen ja unverändert und entscheidet wegen {a,b} ≠ {a,c}, daß beide Kanten nichts miteinander zu tun haben:



Richtig wäre also nicht die ursprüngliche Vorschrift:

„Erstelle für gleichbeschriftete Kanten im NDEA eine (ebenso beschriftete) Kante im DEA."

sondern:

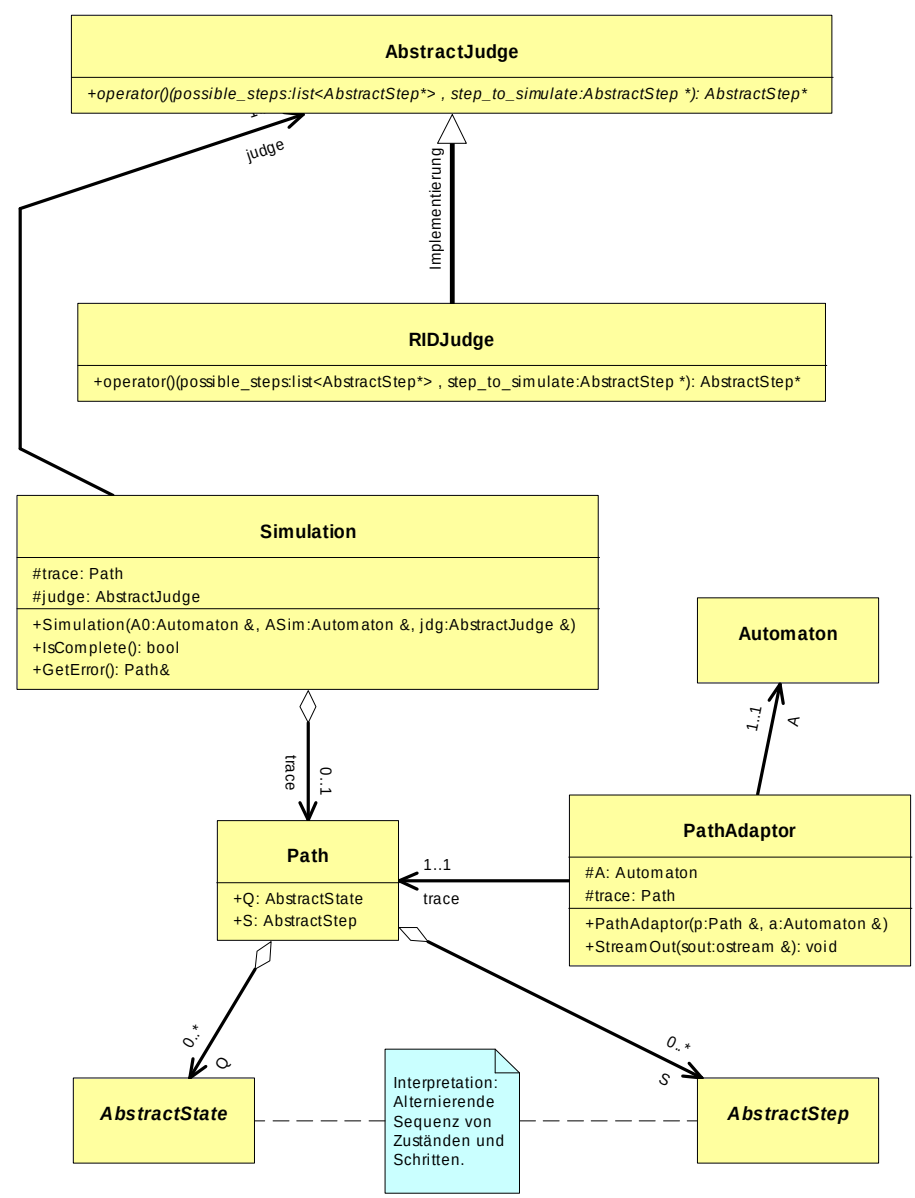
„Erstelle für mit Mengen $X_1,...,X_n$ beschriftete Kanten im NDEA für jede Indexmenge $\{i1,...,im\} \subseteq \{1,...,n\}$ eine mit der Menge $X_{i1} \cap ... \cap X_{im}$ beschriftete Kante."

Wir sehen natürlich schnell, daß unser Generatoren-Konzept ohne weiteres auch eine solche Spezialisierung zuläßt. Der Grund, warum dieser Fall hier doch etwas ausführlicher dargestellt ist, ist der, daß wir für genau diese Anwendung auch eine Verallgemeinerung des Simulationsbegriffs benötigen werden.

5.1.5 Durchführung der Simulation

Sehen wir uns in diesem Stadium des Designs nun die Vorschrift an, um die Simulationsrelation zu erzeugen, so sehen wir uns zunächst in unseren Überlegungen die Modellierung der vorangegangenen Objekte betreffend

bestätigt: Die Formulierung der Vorwärtssimulation ist nämlich möglich, ohne daß dazu noch irgendwelche spezielleren Schnittstellen als die von *Automaton* und *AbstractState/Step* verwendet werden müßten.
Allerdings müssen wir uns hier trotzdem wiederum die Frage stellen, ob diese Beschreibung wirklich schon ausreicht, um nach Möglichkeit für spätere Erweiterungen gerüstet zu sein. Wie oben schon angedeutet, sehen wir am Beispiel der kurz umrissenen Optimierung des Potenzautomaten, daß dem nicht so ist. Hier benötigen wir tatsächlich die Möglichkeit, beim Aufbau der Simulation durch eine Vorschrift gewissermaßen „von außerhalb" der Simulationsklasse nicht nur zu entscheiden, ob eine Kante mit Inschrift l durch eine mit Inschrift l' simuliert werden kann, sondern wir müssen sogar für gegebenes l aus einer mit $l_1,...l_n$ beschrifteten Menge von Kanten diejenige heraussuchen, die zur Simulation verwendet werden soll (nämlich die maximale „passende". Details hierzu werden wir in einem anderen Zusammenhang veröffentlichen.).
Dies führt uns auf folgende Verallgemeinerung: Wir führen eine abstrakte Schnittstelle mit einer einzigen Methode ein, deren Aufgabe es sein wird, für eine gegebene Kante im zu simulierenden Automaten und eine Liste von möglichen Kanten im simulierenden Automaten diejenige Kante herauszufinden, die die passende im Sinne der jeweiligen Simulation ist (bzw. anzuzeigen, falls keine solche existiert). Die Implementierung dieser Klasse für den ursprünglichen Simulationsbegriff etwa sucht in der Liste lediglich nach einer (bzw. *der*, da es sich ja um deterministische Automaten handelt) Kante, die identisch zur gegebenen beschriftet ist.
Insgesamt stellen sich die relevanten Klassen dann so dar:



Schlägt eine Simulation fehl, so wird durch Rückverfolgung der Elemente in der Simulation (deren Darstellung z.Z. durch einen *map*-Container der STL geschieht, obwohl später ein Container mit einem speziellen Hashing-Verfahren eingesetzt werden soll) der Fehlerpfad (im DEA) aufgebaut und in einem *Path*-Objekt gespeichert.

PathAdaptor verbindet lediglich ein *Path*-Objekt mit seinem Automaten. Die dann folgende Rückverfolgung des Pfads liefert ein weiteres *Path*-Objekt, das sich diesmal allerdings auf den NDEA bezieht.

5.1.6 Rückverfolgung des Fehlerpfads

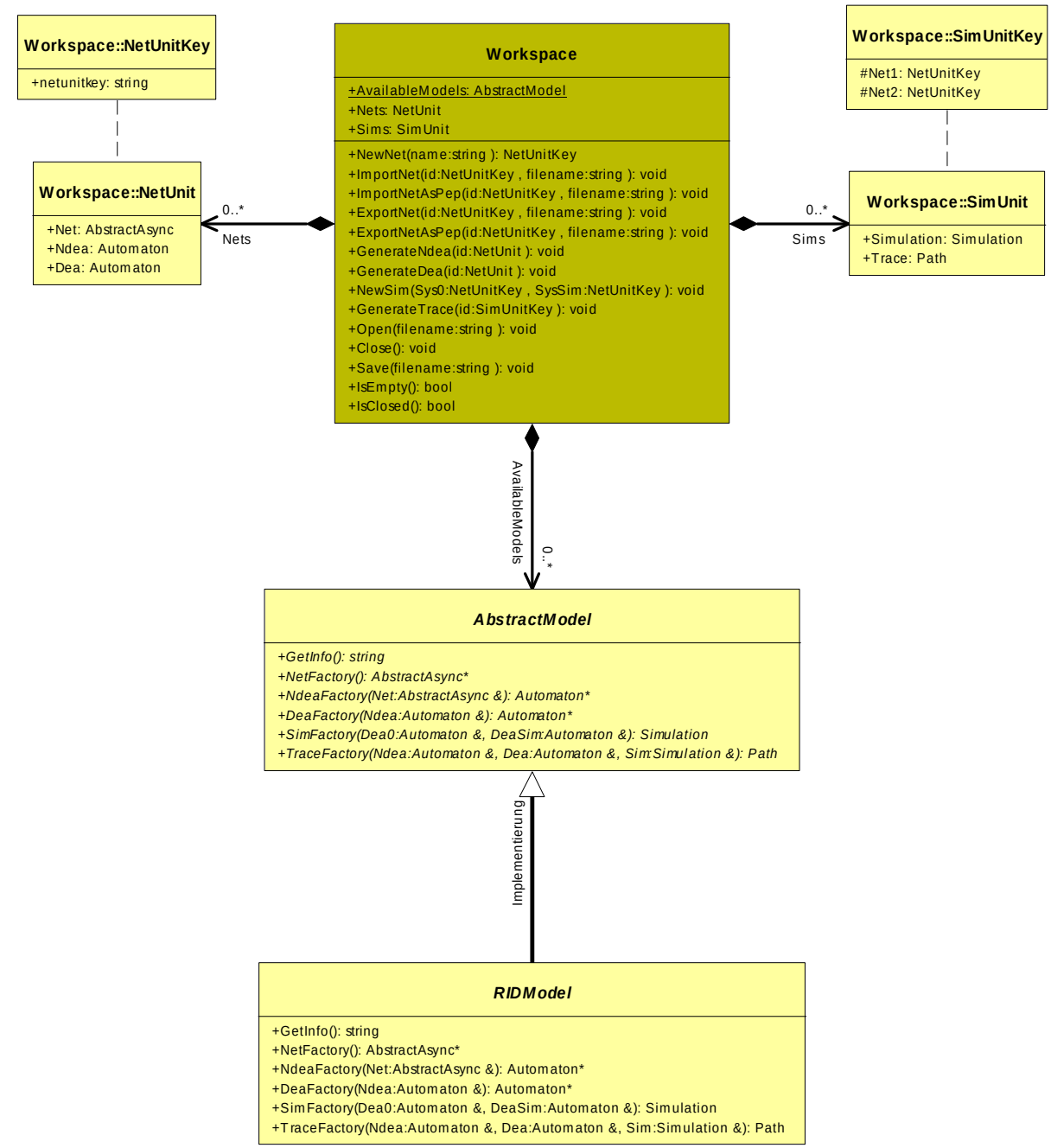
Die Rekonstruktion des Fehlerpfads im ursprünglichen System sei hier nur der Vollständigkeit halber erwähnt, da dies in erster Linie eine Frage der effizienten Implementierung darstellt und vom Klassendesign her eher uninteressant ist. Konzeptuell muß die Komponente zur Rückverfolgung sowieso Kenntnis von allen relevanten Objekten (d.h. Ausgangsobjekt, NDEA und DEA) haben. (Siehe hierzu im Diagramm zu den Generatoren die Klasse *MyhillReconstructor*. Für jeden Generator, der zur Konstruktion eines Potenzautomaten herangezogen wird, muß naheliegenderweise auch eine solche Klasse realisiert werden.)

5.1.7 Einbindung konkreter Modelle durch die Klasse Workspace

Wir haben in den vorangegangenen Abschnitten eine Vielzahl an Möglichkeiten beschrieben, durch Erweiterung der Klassenhierarchie die Funktionalität von FastAsy auf neue Problemklassen auszudehnen. Es stellt sich dann aber die Frage, wie man die Klassengruppen, die jeweils zu einem Effizienzmodell gehören, geschickt in den Quelltext aufnimmt. Die Einbindung soll dabei möglichst wenig Änderungen am Quelltext erfordern. Sieht man nun eine einfache Architektur mit nur zwei Schichten vor, bei der die Benutzeroberfläche direkt die Dienste des Kernels in Anspruch nimmt, wie man es bei kleineren Applikationen oft sieht, so treten sich in unserem Fall gleich mehrere Nachteile auf:

- Änderungen müssen in einer Komponente vorgenommen werden, die nicht portabel ist, müssen also für jede Plattform konsistent gehalten werden.
- Änderungen an der Schnittstelle können nicht zentral stattfinden, da die Dienste über das gesamte User-Interface verteilt aufgerufen werden.

Stattdessen ziehen wir, wie schon im Abschnitt über die Verteilungsstruktur dargelegt, eine zusätzliche Schicht ein, die die Rolle eines Vermittlers zwischen Benutzerinterface und Kernel übernimmt. Ein Blick auf das folgende Klassendiagramm zeigt die Organisation dieser Schicht und die Anbindung von konkreten Modellen:



Es sei kurz darauf hingewiesen, daß *Workspace* ein *individuelle Klasse* darstellt, also sowohl die Aspekte einer Klasse als auch eines Objekts hat. Wir sehen am angegebenen (öffentlichen) Interface von Workspace, daß die angebotenen Dienste sehr elementar sind, wie etwa das Laden und Speichern einzelner Objekte oder des ganzen Arbeitsbereichs oder aber die verschiedenen Berechnungen. Die verschiedenen Objekte werden dabei in sogenannten Units verwaltet, wobei *NetUnit* die eigentliche Beschreibung der Netze und ggf. die daraus berechneten Automaten beinhaltet, während in *SimUnit* die (evtl. nur partielle) Simulationsrelation und ggf. ein Fehlerpfad abgelegt werden.
Um nun die angebotenen Dienste für jedes Modell korrekt ausführen zu können, enthält *Workspace* eine Liste *AvailableModels*, in der alle von *AbstractModel* abgeleiteten Klassen aufgeführt sind (im wesentlichen also die Extension der Klasse *AbstractModel*, da deren Abkömmlinge ebenfalls individuelle Klassen sind). Momentan wird diese Liste im Konstruktor von Workspace explizit erstellt, so daß neue Modelle dort von Hand registriert werden müssen. In einer der nächsten Quelltextüberarbeitungen wird dies jedoch durch eine automatische Registrierung jedes *AbstractModel*-Objekts bei der Instanzierung ersetzt. Wenn *Workspace* nun einen Dienst wie die Berechnung eines Automaten ausführt, so beauftragt es den entsprechenden Abkömmling von *AbstractModel*, ein entsprechend spezialisiertes Objekt zu erzeugen und die Berechnung durchzuführen. Dies wird in der Regel innerhalb von *AbstractModel* durch einen einfachen Konstruktoraufruf erreicht, denn dieser führt dann bereits die Berechnung durch. Man hat jedoch, als eine Art letzte Rückversicherung gegenüber

unvorhersehbaren Erweiterungswünschen, hier auch die Möglichkeit, komplexeres Verhalten, etwa das Erzeugen von spezifischen Zwischenobjekten, einzubringen. Im üblichen Anwendungsfall jedoch handelt es sich bei von *AbstractModel* abgeleiteteten Klassen wieder um nichts anderes als eine „Fabrik“ (siehe Abschnitt übere Generatoren).
Die Vorgeheinsweise für die Einbindung eines neuen Modells die folgende: Zunächst müssen für jeden Aspekt des Modells, der noch nicht durch vorhandene Klassen abgedeckt werden kann, neue Spezialisierungen derjenigen Klassen oder ggf. auch ganz neue Klassen zur Verfügung gestellt werden, so daß der funktionale Aspekt abgedeckt ist. Dann wird von *AbstractModel* spezialisiert, so daß die jeweiligen Aufrufe die vom Modell vorgegebenen Objektklassen liefern. Man beachte, daß alle Arbeiten bis hierher lediglich den modellspezifischen, neuen Code betreffen. Abschließend muß das neue Modell noch dem Workspace bekannt gemacht werden, momentan wie gesagt durch einen Eintrag im Konstruktor, später jedoch durch einfaches Instanzieren der Spezialisierung an einer beliebigen Stelle – in der Regel also in dem Modul, das das neue Modell enthält. (Damit dann ein Benutzer mit dem neuen Modell auch etwas anzufangen weiß, implementiert der Programmierer die abstrakte Methode *GetInfo()*, so daß diese eine Zeichenkette mit einer knappen Beschreibung des Modells liefert. Die Oberfläche bietet dem Benutzer dann in einem Auswahldialog das neue Modell unter dieser Beschreibung an.)

6 Technische Details

6.1 Ein-/Ausgabe und abstrakte Klassen

6.1.1 Die Stream-Operatoren

Viele der Klassen in FastAsy sind von abstrakten Klassen abgeleitet und werden in ihrem Lebenszyklus die meiste Zeit nur über diese geerbten Schnittstellen angesprochen. Ein grundsätzliches Problem, das dabei immer wieder auftritt, ist es, bei einer E/A-Operation die passende Variante der jeweiligen Operation aufzurufen. Die übliche Art, die E/A für eine Klasse K zur Verfügung zu stellen, sind die folgenden Vereinbarungen:

```
ostream& operator<< (ostream& , const K&);
istream& operator>> (istream& , K&);
```

Da es sich hierbei jedoch nicht um Methoden von K handelt (dies ist grundsätzlich nicht möglich, da es sich durch die Schreibweise `stream_objekt << K` ja immer nur um eine Methode von `stream_objekt` handeln könnte), sondern um bloße Funktionen, kann keine späte Bindung stattfinden. Es werden also für abgeleitete Objekte die Funktionen für die Basisklasse K aufgerufen, selbst wenn hinter der Referenz auf K zur Laufzeit ein Objekt einer abgeleiteten Klasse steht.
Eine sehr unschöne Möglichkeit, dies zu umgehen, wäre natürlich, in der Definition obiger Funktionen sozusagen per Hand die Laufzeit-Typinformation (RTTI) auszuwerten und das Objekt in der entsprechenden Form auszugeben. Allerdings würde es dann jedes Ableiten einer neuen Klasse von K nötig machen, diese Funktionen zu ändern, was nicht gerade als wartungsfreundlich gelten kann.
Stattdessen gehen wir einen kleinen Umweg: Dem Interface jeder abstrakten Klasse, für deren Nachkommen E/A eine Rolle spielt, fügen wir zwei abstrakte Methoden hinzu:

```
virtual void StreamOut(ostream&) const = 0;
virtual void StreamIn(istream&) = 0;
```

Da es sich hierbei um virtuelle Methoden der Klasse handelt, sind sie sensibel für RTTI und es wird zur Laufzeit jeweils die richtige Version aufgerufen. Um aber nicht auf die gewohnte Schreibweise verzichten zu müssen, implementieren wir die obigen Prototypen der >>- und <<-Operatoren jeweils einfach durch einen Aufruf von StreamIn bzw. StreamOut.

6.1.2 Ein E/A-Adapter für Automaton

In Zusammenhang mit einer Klasse wie Automaton, die eigentlich nur Verknüpfungen bestimmter Daten verwaltet und von der Art dieser Daten keine Kenntnis hat, ist die Frage, wie bei E/A-Operationen jeweils das korrekte Format sichergestellt wird.
Dabei haben wir das Problem der Ausgabe ja im vorangehenden Abschnitt bereits gelöst, denn ein Automat, der aufgefordert wird, seinen Inhalt auszugeben, ruft einfach für seine Kanten und Zustände die virtuellen Ausgabefunktionen auf. Diese wählen dann in Abhängigkeit von der jeweiligen Spezialisierung das richtige Format aus.
Anders jedoch sieht es bei der Eingabe aus: Dort muß ja, um überhaupt eine solche Methode verwenden zu können, zuerst ein Konstruktor aufgerufen werden. Die Klasse *Automaton* ist jedoch offensichtlich zu generell, um den genauen Konstruktor zu kennen, den sie aufrufen müßte. Deswegen wird hier zunächst der Weg beschritten, die StreamIn-Operation zu verallgemeinern, so daß sie folgende Signatur hat:

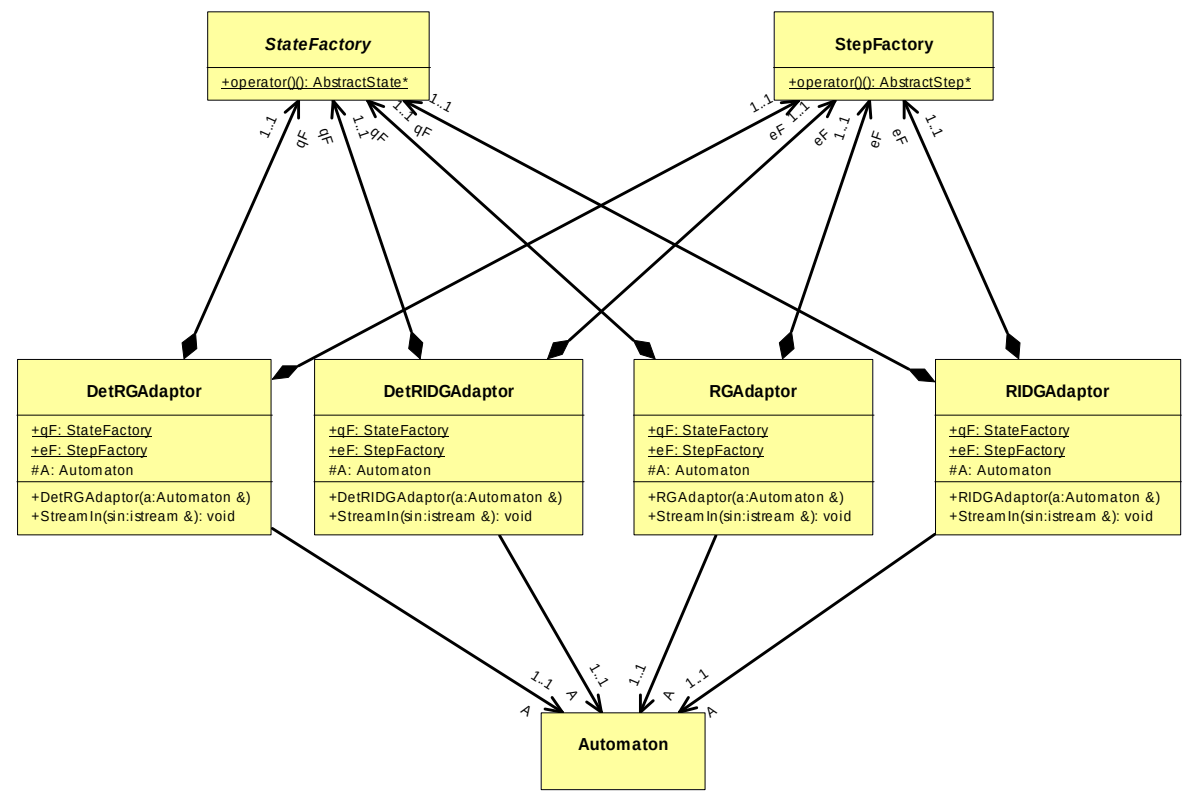
```
typedef AbstractState* StateFactory(void);
typedef AbstractStep* StepFactory(void);
virtual void StreamIn(istream&, StateFactory*, StepFactory*);
```

Damit können wir also der Eingabe-Operation Funktionen mitteilen, die sie zu Hilfe nehmen soll, um neue Kanten und Zustände zu erzeugen. (Wir erinnern uns, daß ähnlich auch beim Aufbau von Automatenobjekten vorgegangen wurde.)
Ein Schönheitsfehler an dieser Lösung ist jedoch noch folgender: Im späteren Programmtext möchte man an den Stellen, an denen solche Eingabe-Operationen eingesetzt werden, nach Möglichkeit nichts von solchen Details sehen. Die erste Möglichkeit, diese Details zu kapseln, besteht darin, in neuen Spezialisierungen von *Automaton* jeweils die StreamIn-Methode mit der alten Signatur durch Einsetzen der (dann bekannten) Argumente für *State-/StepFactory* in den allgemeineren Aufruf einzubetten.

Wir gehen aber hier einen anderen Weg, der, wie wir im folgenden Abschnitt noch sehen werden, noch gewisse Vorteile bietet: Wir definieren uns neue Klassen, die als sogenannte E/A-Adapter dienen. (Wenn im folgenden einfach von Adaptern die Rede sein wird, so darf der Begriff doch nicht mit dem Konzept von Klassenadaptern verwechselt werden, die in der OOP dazu eingesetzt werden, Dienste von Klassen nach außen mit einem anderen Interface zu versehen – etwa, um Kompatibilität mit bestehendem Quelltext zu gewährleisten.)
Ein solcher Adapter ist allgemein eine Klasse, die nach außen hin die üblichen Methoden zur Ein- und/oder Ausgabe zur Verfügung stellt, dabei aber selbst keine Daten speichert, sondern diese sofort an ein Host-Objekt weitergibt. Dies wird derart realisiert, daß dem Konstruktor des Adapters das Host-Objekt als Referenz übergeben wird. Dadurch entsteht beim Aufruf sehr geringer notationeller Aufwand, da das Objekt nur anonym instanziert werden muß, also durch einen Aufruf der folgenden Art:

```
cout << AdapterKlasse( HostKlassenObjekt );
```

Im unserem konkreten Fall ist das Zusammenspiel von Adapter und Automat noch ein wenig subtiler, denn das Vorgehen zum Einlesen eines Automaten ist ja, abgesehen vom Format der Inschriften, immer dasselbe und dies soll (wie oben beschrieben) auch ausgenutzt werden. Also muß ein Adapter für *Automaton* in seiner *StreamIn()*-Methode nur das *StreamIn()* von *Automaton* aufrufen und diesem seine eigenen Implementierungen von *State-* und *StepFactory()* übergeben. Diese Implementierungen wiederum bestehen nur im Aufruf des new-Operators für die entsprechenden Spezialisierungen von *AbstractState* und *AbstractStep*.
Das folgende Klassendiagramm stellt (wieder am Beispiel der Spezialisierungen für die ursprünglich FastAsy-Funktionalität) die implementierten Adapterklassen dar. Man beachte, daß im Rahmen von UML hier *State-/StepFactory* als Funktionalobjekte dargestellt sind, während sie als bloße Funktionszeiger implementiert sind.

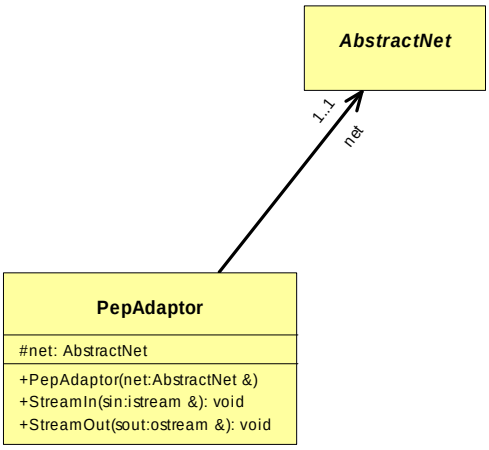


Man könnte vielleicht noch erwarten, daß für die verschiedenen Adapter eine gemeinsame Basisklasse existiert. Es gibt jedoch keinen Grund, eine solche wirklich zu implementieren, obwohl sie konzeptionell völlig natürlich erscheint, denn an keiner Stelle der Implementierung müssen die Adapter über ein gemeinsames Interface angesprochen werden.

6.1.3 Die E/A-Adapter von AbstractNet

Noch in einem anderen Zusammenhang werden Adapter bei der Ein-/Ausgabe benutzt, und zwar bei den Petri-Netzen selbst. Dort nutzen wir ihre zusätzliche Flexibilität aus, um Petri-Netze in verschiedenen Formaten, einem proprietären und dem PEP-Format, ein- und ausgeben zu können, ohne die (umfangreichen) Export- und Import-Routinen für PEP-Files in einer Netzklasse unterbringen zu müssen. (Für das FastAsy-eigene Format

wollen wir aber keinen Adapter implementieren, denn dies soll ja eigentlich das natürliche E/A-Verhalten der Klasse sein.)



Im Gegensatz zu oben sind hier die Adapter für die Ein- *und* Ausgabe relevant, wobei beide Verhalten sich jeweils eine Adapterklasse teilen. Es sei darauf hingewiesen, daß dies unter Umständen dann problematisch ist, wenn const-Referenzen auf Netzobjekte bei der Ausgabe benutzt werden sollen. Dann nämlich müßten zwei Konstruktoren, für const- und für allgemeine Objekte, vorgesehen werden. Die *StreamIn()*-Methode wäre dann nur für einen Aufruf aus einer mit einer nicht-const-Referenz erzeugten Instanz legitim und müßte anderenfalls einen Laufzeitfehler auslösen. Eine Alternative wäre das Aufteilen in getrennte Ein- und Ausgabeadapter.

6.2 Das Idiom „ptr“

In Zusammenhang mit der STL taucht ein Problem immer wieder auf, wenn größere Objekte in STL-Containern verwaltet werden sollen: Die STL ist in natürlicher Weise nur für die Verwaltung von Werten ausgelegt, nicht für Referenzen oder Verweise.
Würde man Zeiger auf Objekte an die STL übergeben, würden etwa zwei identische Kopien eines Objekts nicht als gleich erkannt, denn die Semantik des ==-Operators ist bei Zeigern ja ein Test auf Gleichheit der Speicheraddresse, nicht des Inhalts des adressierten Speicherbereichs.
Noch heikler ist die Lage bei Referenzen: Hier erzeugt schon die Templateinstanzierung Fehler, denn sie bezieht sich ja auf die rein syntaktische Ebene, und dadurch würde etwa das Erzeugen eines neuen Objekts durch einen Konstruktoraufruf für einen Referenztyp instanziert, was natürlich widersinnig ist.
Diese Beschränkung auf Werte kann sehr hinderlich beim Gebrauch der STL sein, denn das Einreihen eines Objekts in einen Container resultiert damit immer im physischen Kopieren des Objekts. Was wir suchen, ist also ein Zeigertyp mit Referenzsemantik des Vergleichsoperators. Nun läßt C++ ja bei elementaren Datentypen kein Überschreiben von Operatoren zu, so daß dieser Weg verstellt ist.
Es ist aber natürlich möglich (und bei ernsthaftem Gebrauch der STL unumgänglich), sich einen neuen Templatetyp zu definieren, der die beschriebenen Anforderungen erfüllt. Da dies in der Tat ein denkbar elementares Klassentemplate darstellt, wollen wir es kurz und prägnant mit *ptr* bezeichnen. Aufgrund der herausragenden Bedeutung soll hier einmal seine Implementierung in voller Länge angegeben werden:

```
template <class T>
class ptr
{
      private:
      T* pointer;

      public:
      // Konstruktoren
      explicit ptr(T& t) : pointer(&t) {}
      ptr(T* p=NULL) : pointer(p) {}
      ptr(const ptr<T>& rp) : pointer(rp.pointer) {}

      // Nicht-manipulierende Operatoren
      bool operator==(const ptr<T>& rp) const {return
            pointer==*rp.pointer;}
      bool operator<(const ptr<T>& rp)  const {return
            *pointer<*rp.pointer;}
      bool operator==(const T* p) const {return *pointer==*p;}
      bool operator<(const T* p)  const {return *pointer<*p;}
      T& operator*() const {return *pointer;}
      operator T*() const {return pointer;}
};
```

Es ist mir nicht verständlich, warum eine solche Vereinbarung nicht fester Bestandteil der STL ist.

6.3 Darstellung von Mengen

Wie man an den kurzen Abrissen der Algorithmen, die in FastAsy verwendet werden, erahnen kann, spielt die Modellierung von Mengen eine zentrale Rolle. Typisch bei diesen speziellen Mengen ist es, daß sie immer Teilmengen einer endlichen, während der Lebensdauer des Objekts unveränderten Grundgesamtheit sind, wie z.B. Aktionsmengen oder Transitionsmengen. Wir wollen uns dabei zunächst auf Mengen von natürlichen Zahlen beschränken, zu denen all diese Mengen trivialerweise bijektiv sind, später soll jedoch noch davon die Rede sein, wie die Klasse *RegSet* diese Beschränkung wieder aufhebt.
Zwar bietet die STL natürlich Möglichkeiten, solche Mengen zu modellieren, da Operationen darauf jedoch sehr oft in den inneren Schleifen der Algorithmen stattfinden, ist es angebracht, sich über die Effizienz ein wenig mehr Gedanken zu machen. Typisches Anforderungsprofil ist wie gesagt, daß die maximale Anzahl von Elementen (d.h. die Größe der Grundgesamtheit) wenn nicht konstant, so doch nur sehr selten Änderungen unterworfen ist, wohingegen sehr häufig Operationen auf ganzen Mengen, wie z.B. Schnitte und Vereinigungen, vorkommen. Eine Implementierung als eine Form von Array ist also der als Liste klar vorzuziehen. Da wir uns weiter auf natürliche Zahlen beschränken, ist eine Modellierung als Bitvektor die effizienteste Lösung. Diese bietet folgende Vorteile:

- Mengenoperationen können schnell ausgeführt werden, da sie an den meisten Stellen durch Byte-, Word-, oder DWord-Operationen statt durch Bit-Operationen implementiert werden. Ein hochoptimierender Compiler sollte den entsprechenden Code sogar in die entsprechenden String-Befehle, die moderne Prozessoren bieten, umsetzen können.
- Zugriff auf einzelne Elemente erhält man schnell durch zwei Shift-Operationen, eine And-Verknüpfung und eine Indirektion: Sei n das gesuchte Element, dann erhält man aus den unteren 3 Bits von n durch 1<<(n&7) die Zugriffsmaske und aus den restlichen Bits von n den Offset (n>>3), den wir zur Basisadresse des Datenfelds addieren müssen.
- Man gewinnt ohne weiteres Zutun eine totale Ordnung auf den Mengen, nämlich die Interpretation des Bitvektors als Binärzahl. Dies ist vor allem dann nützlich, wenn wir etwa Mengen in Suchbäumen speichern wollen.

Leider bietet die STL (besser gesagt: die Schnittmenge der verwendeten STL-Implementierungen) momentan keine passende Modellierung von Bitfeldern an, die auch portabel wäre. Auch wenn man davon ausgehen kann, daß dieses Manko in absehbarer Zeit behoben sein wird, zwingt uns dies doch dazu, den entsprechenden Entwurf bis zur untersten Detailebene durchzuführen.

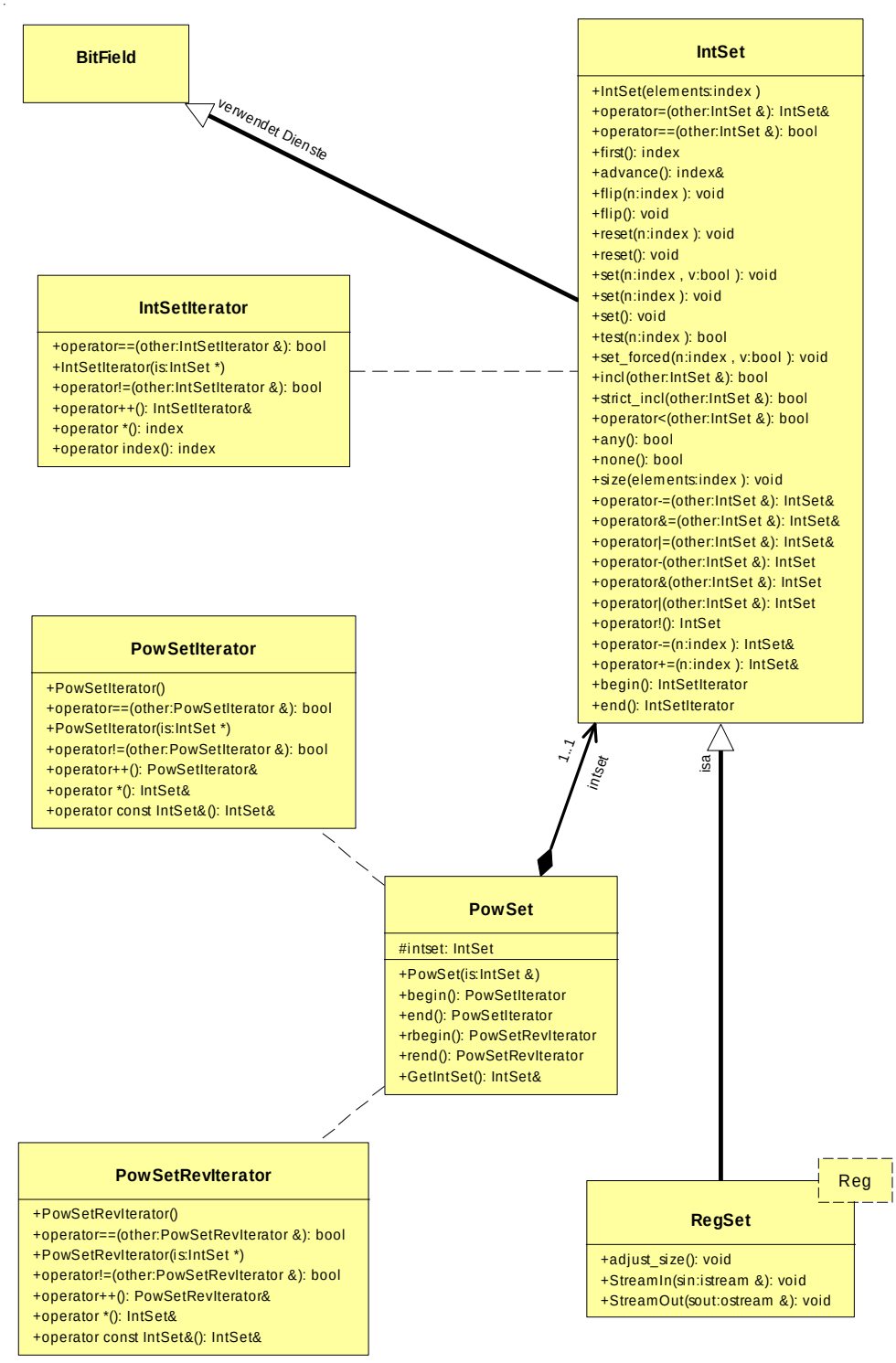
6.3.1 BitField

Dabei haben wir zunächst eine Klasse *BitField* implementiert, die alle nötigen Methoden bereitstellt, um komfortabel mit Bit-Arrays umgehen zu können. Dazu zählen auch die logischen Operationen, auf die sich später unsere Mengenoperationen abstützen sollen. Es wurde bei der Implementierung wie gesagt besonderer Wert darauf gelegt, wann immer möglich die Tatsache auszunutzen, daß es für einen Prozessor ja natürlich ist,

mit einem Zugriff mehrere Bits auf einmal zu bearbeiten. Ansonsten enthält die Realisierung der Klasse *BitField* keine Besonderheiten, so daß wir nicht weiter in die Tiefe gehen wollen.

6.3.2 IntSet

Die Klasse *IntSet* soll nun die Dienste von *BitField* verwenden, um unsere Mengen zu modellieren. Um Raum für low-level-Optimierungen zu lassen, geschieht die Einbindung von *BitField* nicht etwa als Attribut, sondern durch Vererbung, so daß wir nicht etwa durch *friend*-Deklarationen erst den Zugriffsschutz umgehen müssen. Da nicht zu erwarten ist, daß sich die Implementierung und insbesondere das Speicherlayout von *BitField* in einem späteren Entwurszyklus noch ändert, erscheint ist an dieser Stelle vertretbar, *IntSet* direkten Zugriff auf die Datenelemente zu gewähren.
Beim Entwurf von *IntSet* wurde darauf geachtet, daß Interface konform zu dem der STL-Container zu gestalten, auch wenn es sich im Gegensatz zu diesen natürlich bei *IntSet* gar nicht um ein Template handelt. Um dies zu verdeutlichen, sei im folgenden Diagramm für *IntSet* einmal die gesamte öffentliche Schnittstelle angegeben:



Es sei auf die kleine Fußangel hingewiesen, daß es im Kontext der STL nicht ratsam ist, den <-Operator als Test auf Mengeninklusion zu definieren, denn STL-Container benutzen diesen Operator zur Sortierung und erfordern daher tatsächlich einen totalen Ordnungsbegriff. Zwar kann bei der Templateinstanzierung bei allen verwendeten STL-Implementierungen optional eine abweichende Methode angegeben werden, die diesen Test durchführt, doch muß dies dann eben bei jeder Templateinstanzierung erneut erfolgen. Die Chance, daß dies irgendwann vergessen wird, darf im Umfeld einer größeren Klassenhierarchie recht hoch angesetzt werden. In Anbetracht der Tatsache, daß ein solcher Fehler später typischerweise nur sehr schwer zu entlarven ist, wählt man wohl besser das geringere Übel und benennt die Teilmengenbeziehung dann mit *incl()* bzw. *strict_incl()*.

6.3.3 Iteratoren

Ebenfalls nach STL-Vorbild gestaltet ist das Interface von *IntSetIterator*. Unter einem Iterator verstehen wir die Verallgemeinerung einer Schleifenvariable, wobei eben nicht über ein Intervall natürlicher Zahlen, sondern über die Elemente in einem Container iteriert wird. Dazu stellen Iteratoren folgende Operationen zur Verfügung:

- Test auf Gleichheit/Ungleichheit
- Iterieren
- Dereferenzieren

Nun möchten wir aber an einigen Stellen in FastAsy nicht über alle Elemente, sondern über alle Teilmengen iterieren. Dazu benötigen wir offensichtlich eine weitere Iteratorenklasse *PowSetIterator*. Wollen wir nun aber ein STL-ähnliches Erscheinungbild gewährleisten, so ist die Frage, wie man ein *PowSetIterator*-Objekt erhält, denn nach STL-Notation erhält man für eine Containerklasse C mit *C::begin()* und *C::end()* Exemplare von Vorwärtsiteratoren die jeweils auf das erste bzw. hinter (!) das letzte Element zeigen, mit *C::rbegin()* und *C::rend()* entsprechende Rückwärtsiteratoren.

6.3.4 PowSet

Um nun dieses vertraute Schema beizubehalten, führen wir die Klasse *PowSet* ein. In ihrem Konstruktor wird ihr die Referenz auf ein *IntSet*-Objekt übergeben. Das *PowSet*-Objekt repräsentiert nun während seiner gesamten Lebenszeit die Potenzmenge der ursprünglichen Menge, wobei diese natürlich nicht materialisiert wird. Der einzige Zweck von *PowSet* ist es, die passenden Iteratoren zur Verfügung zu stellen.

6.3.5 RegSet und Register

Wie weiter oben angedeutet, stehen hinter den natürlichen Zahlen, die *IntSet* verarbeitet, konzeptuell natürlich allgemeinere Objekte. Speziell gilt dies in FastAsy momentan für: Aktionsbeschriftungen, Transitionen und Stellen. Dabei werden diese Objekte vom Benutzer ja nicht durch eine Nummer benannt, sondern durch einen eindeutigen Namen. Besonders wichtig ist dies bei den Aktionsbeschriftungen, wo ja dieser Name von den Algorithmen auch wirklich zum Test auf Gleichheit verwendet wird, und zwar – und hier beginnen die Komplikationen – auch objektübergreifend, also etwa die Aktionen zweier verschiedener Netzobjekte (aber natürlich der gleichen Netzklasse).
Die simple Lösung, nämlich eine Aktion auch einfach mit einem Zeichenkettenattribut *Name* zu vereinbaren, bringt inakzeptable Nachteile mit sich: Nicht nur würden wir in den inneren Schleifen der Algorithmen ständig Zeit damit verschwenden, langwierig Zeichenketten auf Gleichheit zu prüfen, wir hätten auch kaum eine Möglichkeit, Mengen solcher Objekte mit einem effizienten Mengenmodell wie *IntSet* zu modellieren.
Was wir benötigen, ist eine Art globaler Nachschlagetabelle, in der alle schon zur Benennung eines oder mehrerer Objekte (die Rede ist hier von Objekten des Lösungs- nicht des Problembereichs, also tatsächlich Instanzierungen von Klassen) herangezogenen Namen. Dabei möchten wir aber notationellen Mehraufwand grundsätzlich vermeiden, denn man vermag sich vorzustellen, wie es um die Lesbarkeit von Quellcode bestellt ist, in dem ein Schritt

„Teste, ob die Aktionsmenge A in B enthalten ist“

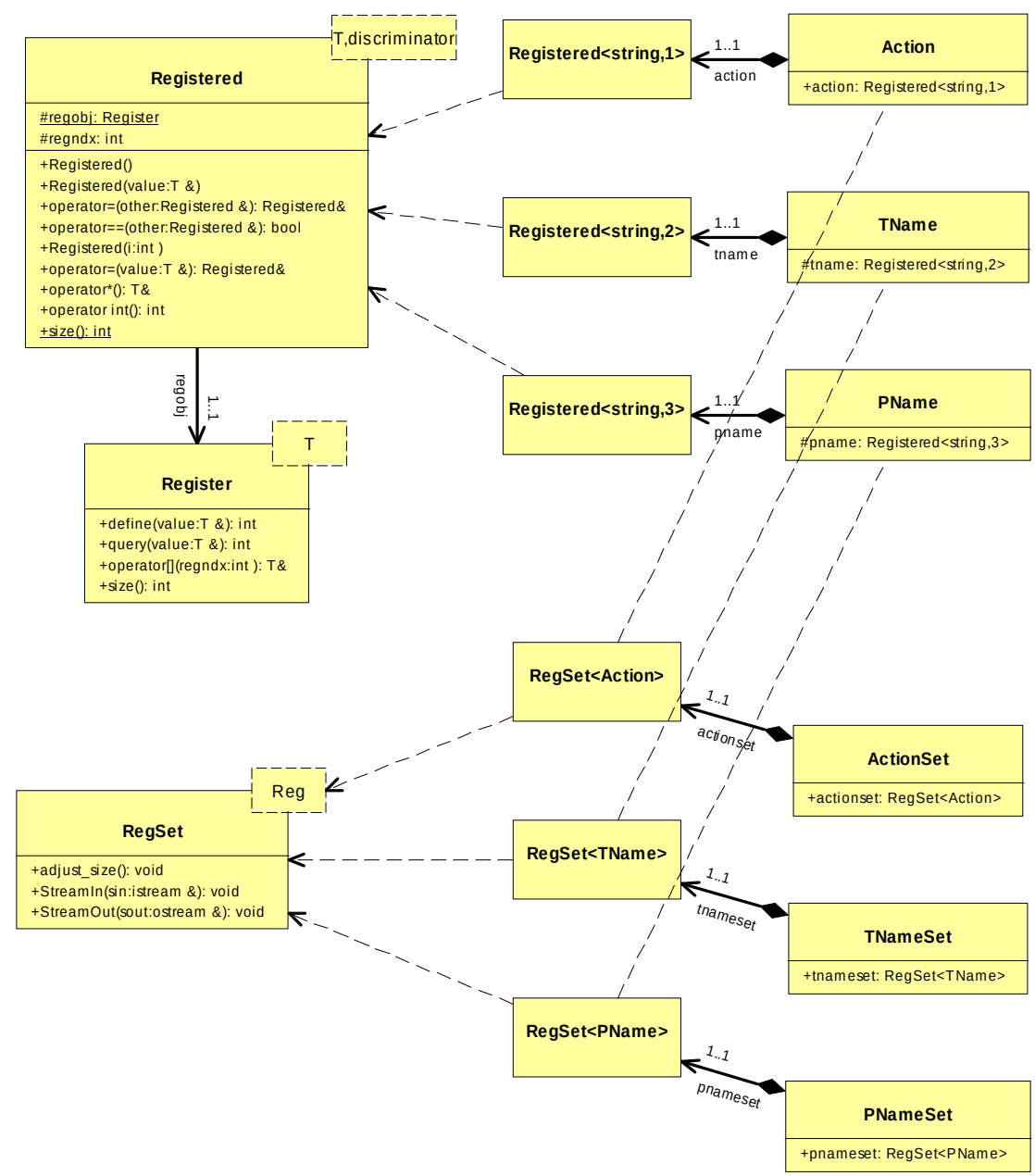
ausartet zu

„Für jedes Element a aus A schlage die Benennung l(a) nach und prüfe, ob es in B ein Element b gibt, so daß b in der Tabelle mit l(a) beschriftet ist.“

(Wir wollen dem Leser die Angabe von echtem Quellcode für dieses „Horrorszenario“ ersparen.)
Stattdessen benötigen wir einen Mechanismus, der dies vor weiter oben liegendem Quellcode versteckt. Beginnen wir also damit, daß wir eine Klasse *Register* definieren, die die Rolle der Nachschlagetabelle übernehmen soll. Das Verhalten, das wir von dieser Klassen fordern, ist nicht sonderlich komplex: Wir möchten neue Benennungen bekannt machen, die Indizes von Benennungen und die Benennungen zu gegebenen Indizes erfragen. Dies alles können Container der STL auch, und nachdem wir noch feststellen, daß die Suche nach Indizes von gegebenen Benennungen im Gegensatz zur umgekehrten Richtung im Hinblick auf die Effizienz eine sehr untergeordnete Rolle spielt, entscheiden wir uns für eine Implementierung durch das *vector*-Template der STL. Obwohl in FastAsy bisher nur Spezialisierungen für Zeichenketten existieren, möchten wir den Typ trotzdem noch nicht genau festlegen und geben den Templateparameter von vector *T* in der Klasse *Register* einfach nach oben weiter.
Als nächstes definieren wir die Klasse *Registered*, die sich einerseits nach außen in Bezug auf Zuweisungen und Wert wie der Typ *T* (hier also eine Zeichenkette) selbst verhalten soll, andererseits aber die Aufgabe hat, implizit die gesamte Verwaltung zu übernehmen.
Ein *Registered*-Objekt soll also bei einer Zuweisung:

- bei seinem assoziierten *Register*-Objekt anfragen, ob diese Bennenung bereits existiert,
- falls ja, sich den Index verschaffen,
- falls nein, die neue Benennung bekannt machen.

Bei einer Konversion nach *T* soll das Objekt natürlich die Benennung liefern. Bei einem Vergleich aber ist dies eben *nicht* notwendig, denn dabei brauchen ja nur die Indizes verglichen werden. Daraus sollte auch schon klar geworden sein, daß wir gar nicht vorhaben, die Benennungen in *Registered* abzulegen. Stattdessen speichen wir die (im Vergleich zu den Indizes womöglich sehr großen) Benennungen nur einmal in der *Register*-Tabelle.
Um nun auch verschiedene Tabellen (d.h. für Aktions-, Transitions- und Stellennamen je eine) verwalten zu können, fügen wir dem Template *Registered* einen künstlichen zweiten Templateparameter hinzu. *Registered* enthält nämlich sein *Register*-Objekt als Klassenvariable, und durch den zusätzlichen Parameter handelt es sich dann bei den Klassen *Action*, *TName* und *PName* jeweils um verschiedene (Template)-Instanzierungen, die folglich jede einen eigenen Satz Klassenvariablen besitzen. (*Action*, *TName* und *PName* sind in der Implementierung als typedefs realisiert, was das UML-Schema leider nicht 1:1 wiedergeben kann.)

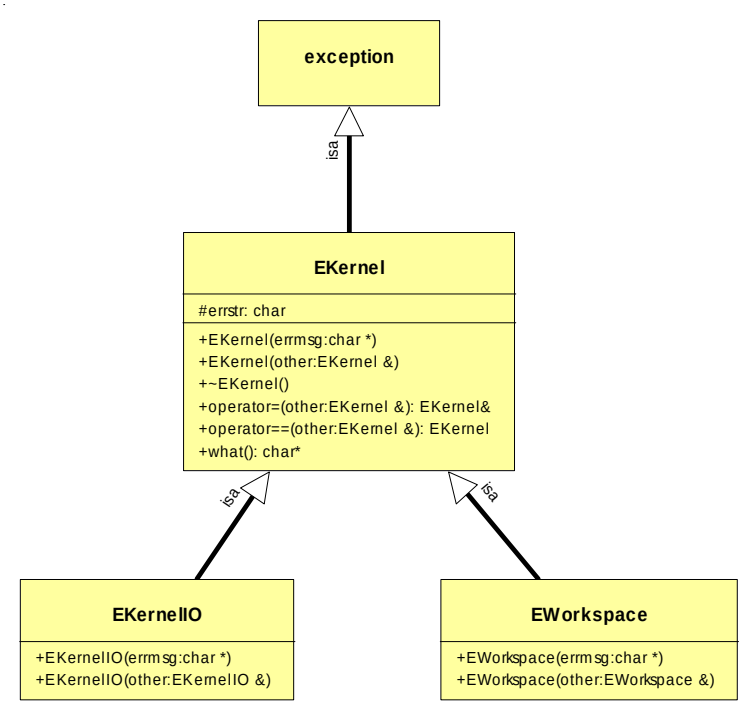


Das Klassentemplate *RegSet* schließlich leistet nun die versprochene Aufgabe, benannte Objekte in *IntSet*-Mengen speichern zu können, was nun auch nicht mehr verwundern wird, da wir doch in *Registered* schon nur mehr natürliche Zahlen abgelegt haben. Wir müssen also lediglich noch dafür sorgen, daß die Ein- und Ausgabe unter Verwendung der Benennungen statt der Indizes erfolgt. *RegSet* wird dazu lediglich unter Angabe einer *Registered*-Instanzierung selbst instanziert, womit die Ein- und Ausgabe dann jeweils mit dem Inhalt der

betreffenden *Register*-Tabelle übersetzt wird. Kenntnis von der passenden Tabelle hat *RegSet* natürlich von *Registered*, denn dies hat die Tabelle ja als Klassenattribut.

6.4 Fehlerbehandlung

Die strukturierte Behandlung von Laufzeitfehlern erfolgt, wie in C++ mittlerweile üblich, durch den *throw()*-Mechanismus. Dazu werden von der Basisklasse *exception* spezifische Fehlerklassen abgeleitet:



Dabei sind die Fehlerklassen wie folgt verteilt:

- Interne Laufzeitfehler des Kernels, die durch Bugs in FastAsy verursacht und durch Zusicherungen erkannt werden, gehören keiner spezielleren Klasse an und sind Instanzen von *EKernel*. Sie werden bis an die Oberfläche weitergegeben und brechen den Programmlauf ab, da nach ihrem Auftreten das Programm sowieso kein wohldefiniertes Verhalten mehr hat.
- *EKernelIO* wird geworfen, wenn bei einer Ein- oder Ausgabeoperation Fehler erkannt werden. Typische Vertreter dieser Klasse sind Formatfehler bei Eingaben oder Gerätefehler. Sie sollten bereits von *Workspace* abgefangen werden, von wo aus ggf. auch eine Wiederholung einer fehlgeschlagenen Operation eingeleitet werden kann.
- Ein Fehler der Klasse *EWorkspace* bedeutet, daß ein Auftrag der Benutzeroberfläche nicht ausgeführt werden konnte. Dabei kann es sich auch um das Resultat eines *EKernelIO*-Fehlers handeln, den Workspace nicht behandeln konnte.

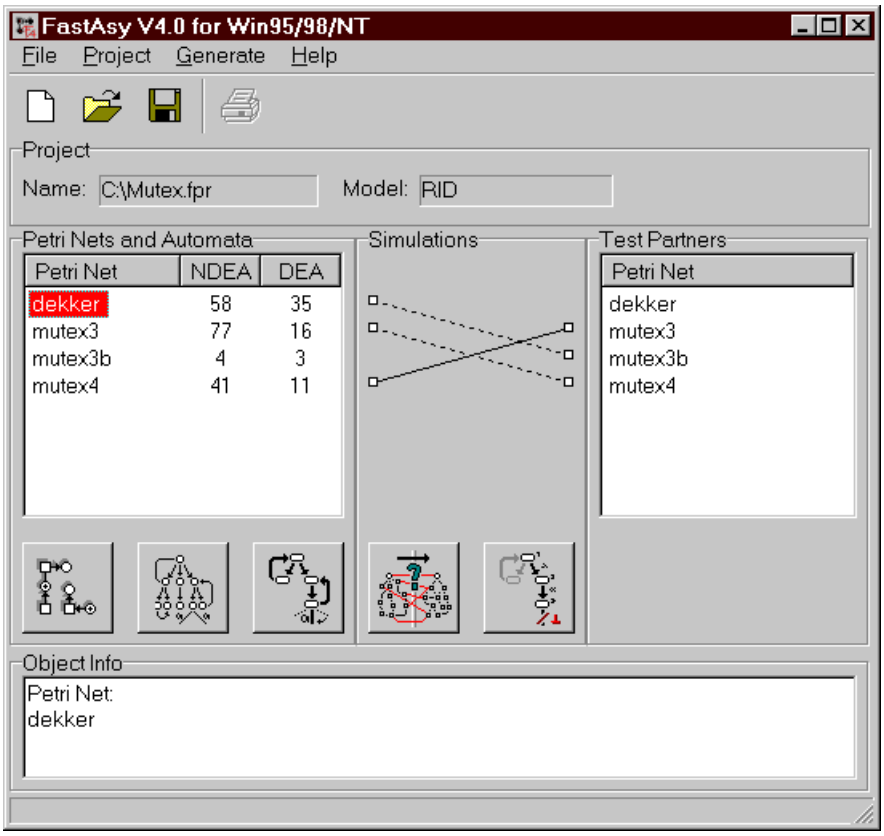
Dieses Modul stellt übrigens eine der extrem seltenen Situationen dar, in der bedingte Compilierung eingesetzt werden mußte, um den Code portabel zu halten. Der Borland C++-Builder läßt nämlich entgegen der ANSI-Spezifikation nicht zu, daß *EKernel* von *exception* abgeleitet wird. Dies ist dadurch bedingt, daß die Fehlerbehandlung beim C++-Builder sowohl einen STL- als auch einen VCL-Aspekt hat.

7 Die graphische Oberfläche

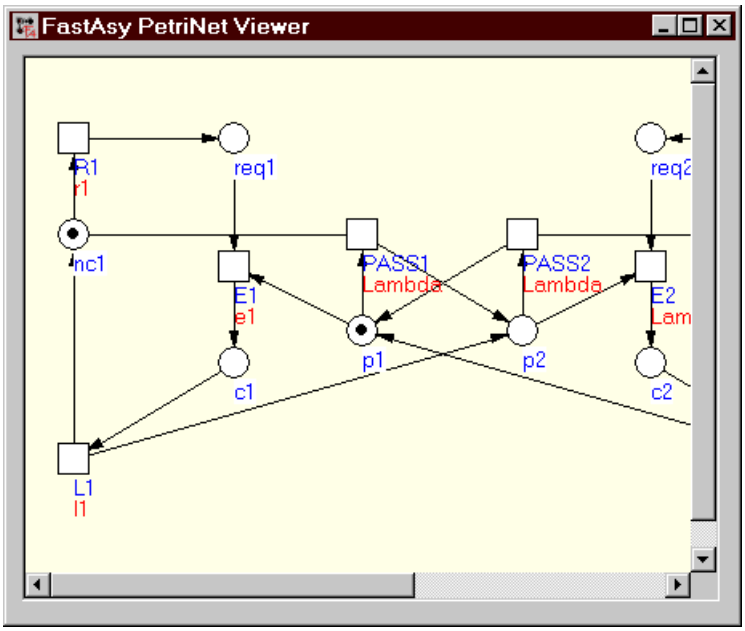
Im Bereich Petri-Netze existiert eine Vielzahl von Tools, die im universitären Umfeld entstanden sind, und die mit einer GUI ausgestattet sind. Dabei fällt in den allermeisten Fällen auf, daß das Design der Benutzerschnittstelle auf Grundlage der Philosophie „form follows function“ erfolgte, worunter wir verstehen, daß das Design des Benutzerinterface sich sehr stark an den interne Erfordernisse der Applikation anlehnt. Dies ist vom Design- und Implementierungsaufwand her sicher sehr ökonomisch; außerdem werden Anwender, die mit diesen internen Abläufen vertraut sind, gut mit dieser Art Oberfläche zurechtkommen. Probleme wird allerdings derjenige Benutzer haben, der an der Entstehung des Tools unbeteiligt war und es nur selten benutzt oder nur kurzzeitig zu Evaluierungszwecken installiert.
Da wir, wie schon erwähnt, planen, FastAsy zur „Illustration“ von am Lehrstuhl entstandenen theoretischen Arbeiten über WWW allgemein verfügbar zu machen, liegen uns eben auch gelegentliche Benutzer am Herzen. Deshalb wird bei FastAsy versucht, die GUI vom Look-and-Feel an die von Standardsoftware vorgegebenen Konventionen anzupassen, bzw. dort, wo diese Konventionen fehlen, die Bedienung so intuitiv und „windows-like“ wie möglich zu gestalten, selbst wenn dies an vielen Stellen Mehraufwand bedeutet.

7.1 MDI vs. SDI

Die erste Entscheidung, die dabei zu fällen war, war die zwischen Multi Document Interface (MDI) und Single Document Interface (SDI). MDI ist dabei das Konzept, das typischerweise von gängigen Textverarbeitungs-, Spreadsheet- oder Zeichenprogrammen benutzt wird: Die Anwendung wird über ein einzelnes Hauptfenster bedient, das alle Controls zur Interaktion trägt. Die verschiedenen Objekte (Dokumente) kommen dadurch ins Spiel, daß diese als verschiedene untergeordnete Fenster innerhalb des Hauptfensters dargestellt werden. Die untergeordneten Fenster können das Hauptfenster dabei nicht verlassen, und wenn das Hauptfenster minimiert, geschlossen, usw. wird, so erfaßt dies auch alle Dokumentfenster.
Nun mag eine solche Oberfläche für FastAsy ja zunächst durchaus sinnvoll erscheinen, in der dann etwa alle im Speicher befindlichen Netze als untergeordnete Fenster dargestellt werden. Direkt von einem Netz abhängige Objekte, also NDEA und DEA, wären ebenfalls mit dessen Fenster assoziiert und könnten etwa über Punkte in einem Menü *Ansicht* umgeschaltet werden (man vergleiche hierzu diverse Office-Software). Dabei vergißt man aber, daß weitere zentrale Objekte nicht in dieses Schema passen: Wir sind ja prinzipiell an den Vergleichen zwischen den Netzen interessiert, und diese würden sich dann ja auf mehrere Fenster beziehen. Also müßte man diese Vergleiche selbst auch wieder in untergeordneten Fenstern darstellen, womit aber erstens die Homogenität des MDI durchbrochen wäre (dadurch müßte das Hauptfenster etwa jedesmal, wenn ein Fenster eines anderen Objekttyps aktiviert würde, mit einem Schlag den Großteil seiner Kontrollelemente dem neuen Kontext entsprechen anpassen); zweitens ginge recht schnell die Übersicht verloren – man halte sich vor Augen, daß die Anzahl der Simulationen, wenn man an einer kompletten Vergleichsserie aller Netze im Speicher interessiert ist, quadratisch mit der Anzahl dieser Netze ansteigt.
Stattdessen scheint es also angebracht, genau eine Übersicht über mögliche und vorhandene Simulationen zum zentralen Objekt der Benutzeroberfläche zu machen. Wir haben dies realisiert, indem wir das Hauptfenster links und rechts mit zwei Listen mit versehen haben, in denen beide Male alle Netze im Speicher aufgeführt werden. Dabei ist links zusätzlich die Knotenzahl der abhängigen Automaten angezeigt. In der Mitte des Fensters gibt uns eine graphische Anzeige darüber Auskunft, welche Simulationsversuche bereits durchgeführt wurden und ob diese jeweils eine partielle (gestrichelte Linie) oder eine vollständige (durchgezogene Linie) Simulationsrelation als Ergebnis hatten:



Detailiertere Kontextinformationen über selektierte Objekte kann sich der Benutzer in einem darunter liegenden Textfeld ausgeben lassen.
Konsequenterweise beziehen sich die Befehle im Menü *File* auf die Gesamtheit von Netzen, Automaten und Simulationen, zusammen mit dem zugeordneten Modell. Befehle zum Import- und Export einzelner solcher Objekte befinden sich dagegen im Menü *Project*, man vergleiche hierzu die Diktion verschiedener IDE-Software. Über ein weiteres Menü *Generate* schließlich lassen sich die verschiedenen Berechnungen einzelnen oder als ganzes anstoßen.
Für Befehle im *File*-Menü existiert oben am Fenster ein Toolbar mit den üblichen Piktogrammen für die Standardoperationen. Die Schaltflächen unter der linken Liste bzw. unter der Anzeige der Simulationen hingegen ändern ihr Verhalten in Abhängigkeit vom Kontext: Ist beispielsweise kein Netzobjekt selektiert, geht die Schaltfläche mit dem Netz-Bildchen davon aus, daß der Benutzer ein neues Netz erstellen möchte, ist dagegen ein Netzobjekt selektiert, wird dieses dem Benutzer angezeigt. Hat ein selektiertes Netzobjekt noch keinen zugehörigen NDEA und der Benutzer betätigt die Schaltfläche *NDEA* (die zweite von links – zur Laufzeit wird diese natürlich über die üblichen Tooltip-Texte benannt), so wird der NDEA berechnet, ist ein solcher jedoch schon vorhanden, wird er wiederum angezeigt, usw...
Wenn wir nun immer davon gesprochen haben, ein Netz oder Automat würde angezeigt, ist damit gemeint, daß tatsächlich ein weiteres (nicht-modales !) Fenster mit dem Objekt geöffnet wird, z.B.:



Wollte man nun wirklich noch auf ein MDI-Erscheinungsbild hinaus, so könnte man im Hauptfenster einen größeren Bereich vorsehen, in dem diese Objekte dann als untergeordnete Fenster erscheinen. Damit würde das Hauptfenster dann allerdings etwas schwerfällig, so daß der Benutzer in einer recht hohen Auflösung arbeiten muß, um komfortabel damit arbeiten zu können. Dann allerdings hätte diese Lösung womöglich durchaus einen gewissen Reiz, auch wenn damit, in abgeschwächter Form, wieder die Homogenität der MDI-Fenster verletzt wäre.

7.2 Benutzerkomfort

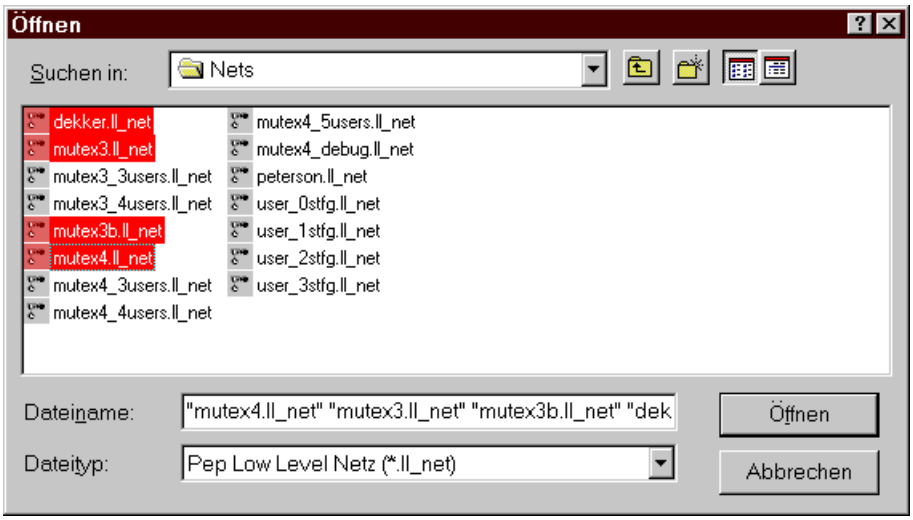
Hier soll kurz exemplarisch auf einige Details eingegangen werden, die der Benutzer (wohl mit gutem Recht) intuitiv erwartet, die aber immer etwas zusätzlichen Aufwand bei der Verwirklichung nach sich ziehen.

7.2.1 Sinnvolle Fensterskalierung

Eine Grundanforderung die Ergonomie einer Oberfläche betreffend ist es, Fenster durch Ziehen am Rahmen in ihrer Größe anpassen zu können. Ohne dieses Merkmal kann der Benutzer bei verschiedenen Auflösungen den ihm zur Verfügung stehenden Raum auf seinem Bildschirm nicht sinnvol nutzen.
Bei Inhalten wie Text oder auch Graphik ist die Semantik einer Größenänderung meist trivial, es wird einfach der angezeigte Ausschnitt des Objekts größer oder kleiner. Bei einem Fenster mit vielen Controls wie unserem Hauptfenster hingegen lohnt es sich doch, in diesen Punkt etwas Arbeit zu investieren. So sollte sich z.B. primär der Listenbereich mit der Größe des Fensters ändern, da aber die linke Liste weitergehende Informationen enthält, sollte eine Verbreiterung des Fensters nur diese linke Liste betreffen, denn dadurch werden evtl. weitere Listenspalten sichtbar.

7.2.2 Mehrfachauswahl

Ebenfalls ein unauffälliges, aber häufig benötiges Merkmal ist die Unterstützung einer Mehrfachauswahl etwa beim Öffnen von Dateien mit Petri-Netzen zum Import derselben. Beim Start eines neuen Projekts etwa, das auf schon vorhandenen Netzen Vergleiche durchführen soll, will der Benutzer typischerweise eine ganze Reihe von Netzen einlesen. Es ist einfach ärgerlich – nicht mehr und nicht weniger – , wenn diese Prozedur für jede Datei einzeln aufgerufen werden muß.



Analog gilt dies für jede Operation, die der Benutzer mit einer gewissen Wahrscheinlichkeit auf Gruppen gleicher Objekte gemeinsam durchführen möchte.
Eine weitere Dimension des Problems kommt übrigens hinzu, wenn die in Frage stehenden Operationen auch noch zeitaufwendig sind, denn dann möchte der Benutzer diese Operationen möglicherweise lieber einmal anstoßen und seinen Rechner danach für einige Minuten oder Stunden vergessen, statt in Intervallen immer wieder die nächste zeitintensive Aufgabe zu starten.

7.2.3 Threads

Es soll zum Abschluß dieses Abschnitts nun der Fairness halber nicht verschwiegen werden, daß eine wünschenswerte Eigenschaft einer Oberfläche sich in FastAsy einfach nicht mehr mit vertretbarem Aufwand implementieren ließ, nämlich eine permanente Reaktivität. Darunter verstehen wir die Eigenschaft einer Benutzerschnittstelle, auch während andauernder zeitintensiver Operationen (typischerweise etwa Datenbankabfragen, hier aber die Berechnungen von Automaten oder Simulationen) nicht zu blockieren. Dazu müssen solche Operationen in eigenen Threads stattfinden, die vom Hauptthread der Oberfläche angestoßen werden. Sinnvoll wäre dieses Feature vor allem deshalb, weil damit versehentlich angestoßene Berechnungen vorzeitig abgebrochen werden könnten, ohne die ganze Applikation beenden zu müssen und im schlimmsten Fall dadurch nicht gespeicherte Daten zu verlieren.

8 Geplante Erweiterungen

Im folgenden möchten wir noch kurz unsere Vorstellungen darlegen, in welche Richtung FastAsy weiter fortentwickelt werden soll. Der wichtigste Punkt ist dabei natürlich ganz klar die Implementierung der jeweiligen Modelle bzw. Semantiken.

Eine wünschenswerte Erweiterung eher handwerklicher Natur wäre die Implementierung eines komfortablen Netzeditors. Mit komfortabel ist damit nicht nur gemeint, daß er sich in Bezug auf Look-and-Feel konform zu den de-facto-Konventionen verhält, die professionelle graphische Editoren aus allen Bereichen seit langem etabliert haben, sondern auch, daß dem Anwender fortgeschrittene Werkzeuge bei seiner Arbeit zur Verfügung stehen, wie beispielsweise:

- Netzkomponenten können nicht nur durch Cut&Paste, sondern auch über synchrone oder asynchrone Komposition zu größeren Netzen kombiniert werden.
- Lokale, sehr elementare und vor allem parametrisierte Standardkonstruktionen können mit wenigen Mausklicks eingefügt werden, etwa Stellenarrays zur Darstellung von Variablenwerten, oder einfache Puffer.
- Kanten, Stellen und Transitionen können per Mausklick mit Elongationen (Delays) versehen werden.
- Umbenennungen von Stellen/Transitionen können systematisch durch Suchen&Ersetzen von regulären Ausdrücken stattfinden.

Dabei sollte der Netzeditor natürlich der Möglichkeit von Erweiterungen von *NetNode*- und *Arc*-Klassen Rechnung tragen. Da der Netzeditor ja naturgemäß sehr nahe an der GUI liegt, wird es sinnvoll sein, zu diesem Zweck wieder ein Zwischenschicht wie *Workspace* einzuziehen.

Neben einer Möglichkeit, Netze zu bearbeiten, soll der Editor aber auch die Funktion eines Simulators übernehmen. Dies heißt zunächst, daß der Benutzer für eingegebene Netze schrittweise das Token-Game durchlaufen und sich so zumindest ein grobes Bild davon machen kann, ob das Netzverhalten seinen Vorstellungen entspricht. Weiter soll der Simulator aber auch die Ergebnisse von Effizienzvergleichen visualisieren können, indem er dem Benutzer die jeweilige Sequenz, die „langsames“ Verhalten repräsentiert, am Netz selbst vor Augen führt.

Ferner wäre es noch wünschenswert, auch die Zwischenergebnisse, also die Automaten, in einer intuitiveren Form als einer Liste von Zuständen und Kanten zu präsentieren. Bedauerlicherweise sind diese Objekte selbst bei einfachen Netzen viel zu groß, um an eine graphische Darstellung der ganzen Automaten zu denken – mehrere tausend Zustände im NDEA sind eher die Regel als die Ausnahme. Wir denken deshalb eher an eine Art Browser, der den Automaten ausschnittsweise darstellt und in dem der Benutzer sich navigierend bewegen kann.

Anhang A: Literaturverzeichnis

[Bih98] E. Bihler. Effizienzvergleich bei verteilten Systemen - eine Theorie und ein Werkzeug Diplomarbeit an der Universität Augsburg, 1998

[BW99] G. Bannert, M. Weitzel. Objektorientierter Softwareentwurf mit UML Addison Wesley Longman, 1999

[BV98] E. Bihler and W. Vogler. Efficiency of Token-Passing MUTEX-Solutions - Some Experiments. In J. Desel and others, editors, *Applications and Theory of Petri Nets*, Lect. Notes Comp. Sci. 1420, 185-204, Springer, 1998.

[HU94] J. E. Hopcroft, J. D. Ullman. Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie, 3. Auflage, Addison-Wesley, 1994

[JV96] L. Jenner and W. Vogler. Fast asynchronous systems in dense time. In F. Meyer auf der Heide and B. Monien, editors, *Automata, Languages and Programming ICALP'96*, Lect. Notes Comp. Sci. 1099, 75-86. Springer, 1996

[PEP] PEP Homepage. http://www.informatik.uni-hildesheim.de/~pep/

[Vog95] W. Vogler. Faster Asynchronous Systems, Report Nr. 317, Universität Augsburg, 1995

[Vog96] W. Vogler. Efficiency of Asynchronous Systems and Read Arcs in Petri Nets, Report Nr. 352, Universität Augsburg, 1996