Inhaltsverzeichnis

1. Vorwort

Moderne Flugsicherung hat das Ziel, Luftfahrzeuge effizient und sicher vom Startflughafen bis zum Ziel zu begleiten. Das stetige Wachstum des Luftverkehrs hat eine Verdichtung des Flugraums zur Folge, eine effektive Regelung des Flugbetriebs wird dadurch notwendig. Dabei konkurriert der Sicherheitsaspekt mit ökonomischen und ökologischen Zielen.

Gegenstand dieser Arbeit ist die Entwicklung einer *domänenspezifischen Sprache* zur Modellierung des Regelwerks der Flugsicherung im Bereich *Air Traffic Flow Management* und die Realisierung einer *Abflugplanungskomponente* zur Optimierung der Abflugzeitenbestimmung. Die Arbeit beschreibt die Phasen eines Projekts im Bereich Flugsicherung von der Anforderungsanalyse bis zum operativen Betrieb bei dem Kunden.

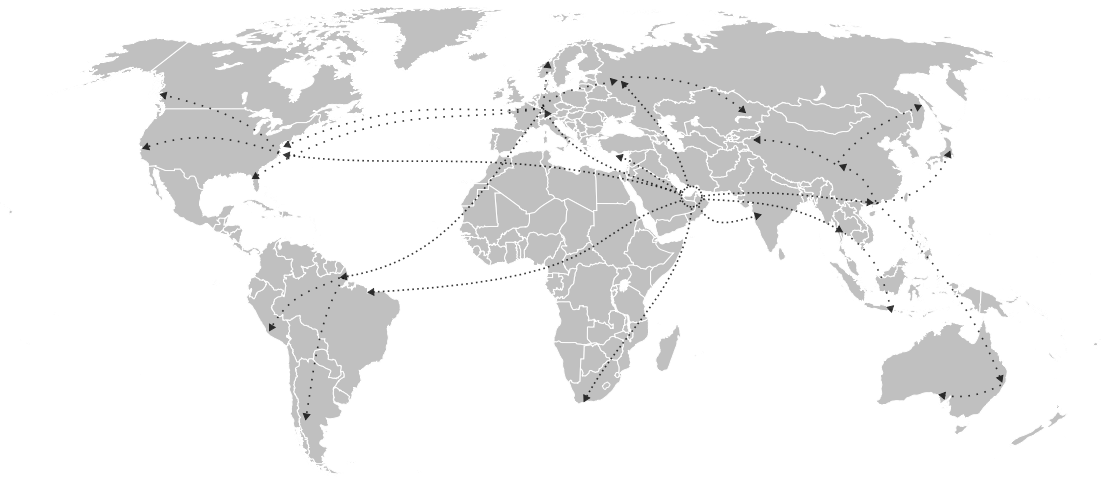


Abbildung 1.1.: Abstraktion der Luftverkehrsrouten ausgehend von den Vereinigten Arabischen Emiraten. *Quelle (Weltkarte): Wikipedia*

1.1. Danksagungen

Dieses Buch ist als Dokumentation einer Projektarbeit während meiner Tätigkeit bei der Comsoft GmbH entstanden. An dem Projekt waren einige Mitarbeiter der Comsoft GmbH und Prof. Dr. Mayer als Betreuer von der Hochschule Offenburg beteiligt.

Als Erstes danke ich Herrn Pitz, als Leiter der Abteilung Centre Solutions hat er die Arbeit an diesem Kundenprojekt mit der GCAA ermöglicht. Ich danke auch den beteiligten Mitarbeitern der GCAA, die konstruktiven Erläuterungen der Verkehrsflussproblematik waren für die Spezifikation der Modellierungssprache äußerst hilfreich.

Ein besonderer Dank geht an Prof. Dr. Erwin Mayer von der Hochschule Offenburg und Dr. Stephan Schulz von der Comsoft GmbH für die tatkräftige Unterstützung bei der Einarbeitung in die Domäne Flugsicherung, der Bildung einer strukturierten Arbeitsweise und für die zusätzliche Motivation, die manchmal notwendig gewesen war.

Nils Hilt hat mich bei der Entwicklung der Abflugplanungskomponente und der Einarbeitung in die PRISMA-Architektur begleitet. Als Entwickler des DFLOW-Displays hat Wenzel Svojanovsky die Planungskomponente für den Benutzer zugänglich gemacht. Hiermit danke ich beiden, ebenso den Mitarbeitern des Testteams, welche die Komponenten auf ihre Zuverlässigkeit geprüft haben.

2. Einleitung

Das Ziel der Flugsicherung ist es den Flugverkehr sicher und entsprechend internationalen und regionalen Richtlinien vom Startflughafen bis zur sicheren Landung am Zielflughafen zu begleiten.

Die Sicherheit im Flugverkehr wird in erster Linie durch die Staffelung erreicht. Als Grundlage dazu bestehen in jedem Flugraum Wegpunkte, welche die Flugverkehrsrouten bilden. Der Verkehr wird auf diesen Routen zeitlich, vertikal und horizontal separiert. Zur Festlegung der Separationsabstände gibt es Regeln, welche von den Flugzeugeigenschaften und gegebenen Umweltbedingungen abhängen.

Oftmals werden an den Übergabewegpunkten von einem Fluginformationsgebiet in das angrenzende maximale Verkehrsflussdichten gesetzt. Diese gebietsübergreifenden Regelungen dienen dazu den Regulierungsaufwand bei der Übernahme von Flügen in das kontrollierte Gebiet zu minimieren.

Bei der Bewältigung von Überkapazitäten werden u.a. die Luftfahrzeuge angewiesen bestimmte Flugmanöver auszuführen, diese sind z.B. *Hold*[1] und *Dog Tail*[2]. Eine weitere Möglichkeit ist die Anpassung der Fluggeschwindigkeit als Regulierungsmaßnahme. Der Nachteil solcher Eingriffe in den Flugverlauf sind die entstehende Mehrkosten für die Flugbetreiber und mögliche Wartezeiten und Verspätungen für die Passagiere. So sind laut [Men04] bis zu 23% aller in Europa registrierten Verspätungen im Flugverkehr auf das Air Traffic Flow Management zurückzuführen, ein Indiz dafür, dass hier großes Optimierungspotential besteht (Abb. 2.1).

[1] Ein Flugmanöver, wobei eine vollständige Ellipse geflogen wird
[2] Ein Flugmanöver, das vertikal betrachtet die Form eines ”n”bildet

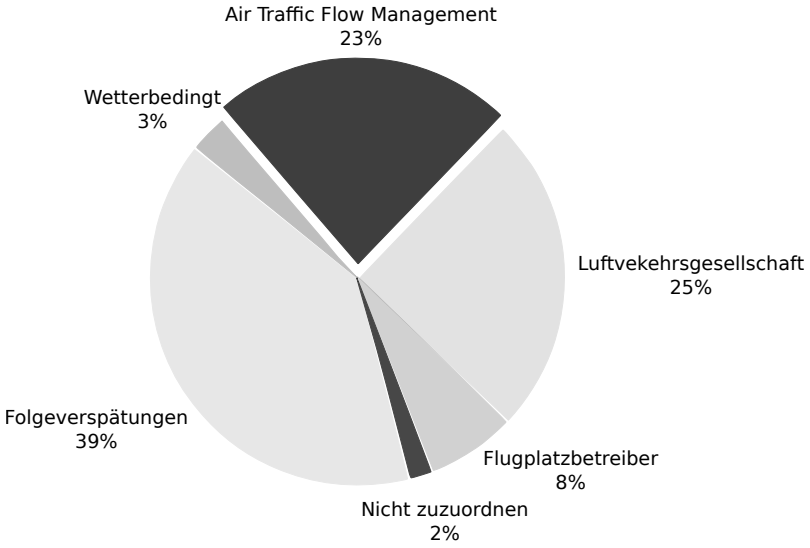


Abbildung 2.1.: Ursachen für Abflugverspätungen in Europa. *Quelle: [Men04]*

2.1. Zielsetzung

Um die Notwendigkeit der regulatorischen Maßnahmen zu reduzieren, sollen bei der Abflugplanung die Verkehrsflussregelungen berücksichtigt werden. Bereits vor Abflug sollen die Flugpläne der registrierten Starter analysiert und der Startzeitpunkt entsprechend des späteren Flugverlaufs gesetzt werden.

Diese Funktion übernimmt die Abflugplanungskomponente. Die Komponente berechnet den optimalen Abflugzeitpunkt für individuelle Flüge, basierend auf den konfigurierbaren Regeln formuliert in einer Domänensprache. Anhand des Regelwerks werden die Flüge nach passenden Mustern sortiert und an den relevanten Wegpunkten gestaffelt.

Zur Modellierung der dynamischen Luftraumbeschränkungen wird eine domänenspezifische Sprache entwickelt. Sie bildet die Schnittstelle zwischen der Planungskomponente und dem Verkehrsflussmanagement. Die Sprache soll eine redundanzfreie Konfiguration der Flugplanmuster und Verkehrsflussbeschränkungen bieten, die der Planer selbst verwalten kann.

Das erste Ziel der Arbeit war es, eine Modellierungssprache für den Bereich des Verkehrsflussmanagements und der allgemeinen Flugsicherung zu erstellen. Als zweiter Teil der Arbeit galt es eine Abflugplanungskomponente zu entwickeln und in eine bereits bestehende Flugsicherungsarchitektur zu integrieren. Die Komponente soll die

Domänensprache als Konfigurationssprache akzeptieren. Die Berechnungen der Abflugplanungskomponente basieren auf dem Modell der Luftraumbeschränkungen.

2.2. Aufbau der Arbeit

Das erste Kapitel beinhaltet Danksagungen für alle, die an der Fertigstellung des Projekts mitgewirkt haben, oder diejenigen, die Unterstützung für die Ausarbeitung dieser Arbeit leisteten. Darauf folgenden bietet das zweite Kapitel eine kurze Einleitung über die Arbeit und das Unternehmen, in dem das hier beschriebene Projekt stattgefunden hat.

Das Grundlagenkapitel widmet sich der Thematik Flugsicherung, der Softwarearchitektur, in der die Neuentwicklungen zu integrieren waren, der Einführung in die Entwicklung von Compiler, den relevanten Sicherheitsfragen für das Projekt und abschließend der Marktanalyse über Produkte mit ähnlicher Funktionalität. Die Absicht dieses Kapitels ist die Schaffung einer Wissensbasis, auf der die folgenden Kapiteln implizit aufbauen und so eine effiziente Abhandlung ermöglicht wird. Ebenso soll die Notwendigkeit der im Rahmen dieser Arbeit entwickelten Komponenten verdeutlicht werden, der Produktvergleich dient als Übersicht über bestehende Systeme und deren Funktionsumfang.

Die Kapitel vier bis sieben protokollieren die Entwicklungsphasen eines Projekts nach dem Prozessmodell V-Modell XT. Die Anforderungsanalyse bildet die erste Phase, basierend auf deren Ergebnissen wird der Entwurf der Komponenten entwickelt. Nach Fertigstellung des Entwurfs werden die Komponenten realisiert und abschließend geprüft. Die Verifikation dient dabei nicht nur der Bestätigung der Richtigkeit im Sinne einer Softwareprüfung, sondern bewertet auch die tatsächliche Leistung des Systems im operativen Betrieb und daraus folgende Auswirkungen auf den Luftverkehrsbetrieb.

Das letzte Kapitel ist einer Zusammenfassung der gesammelten Erfahrungen und bietet einen Ausblick in mögliche Weiterentwicklung der präsentierten Lösung.

2.3. Comsoft GmbH

Die Comsoft GmbH ist ein mittelständisches Unternehmen mit Sitz in Karlsruhe. Als Systemhaus mit Beratungskompetenz hat sich Comsoft im Bereich Flugsicherung auf internationaler Ebene etabliert.

Durch projektgetriebene Arbeitsweise bewahrt sich Comsoft die nötige Flexibilität um kundenorientierte Lösungen zu entwickeln. Als zertifiziertes Unternehmen mit über 20 Jahren Erfahrung im Flugsicherungssektor entwickelt Comsoft hauptsächlich für das europäische und asiatische Ausland, oder wie im Falle der Abflugplanungskomponente für die Vereinigten Arabischen Emirate.

3. Grundlagen

3.1. Flugsicherung

Die Hauptaufgabe der Flugsicherung ist die Vermeidung von Kollisionen zwischen den Luftfahrzeugen. Dazu wird eine räumliche Trennung zwischen allen Verkehrsteilnehmern erwirkt. Im Sichtflugbetrieb stellt diese Aufgabe nur in Ausnahmesituationen ein Problem dar und kann auf die Piloten übertragen werden. Im Instrumentenflugbetrieb jedoch ist der Pilot auf die Anweisungen der Fluglotsen und seine Navigationsinstrumente angewiesen.

Um die Sicherheit der Luftfahrzeuge zu gewährleisten gibt es eine Reihe von Verfahren zur Staffelung des Flugverkehrs.

3.1.1. Luftraumorganisation

Die Strukturierung des Luftraums bildet die Grundlage, auf der Flugsicherungsmechanismen arbeiten. Die Struktur soll die Koordination von Flugbewegungen ermöglichen und die Häufigkeit von Eingriffen in den Flugverlauf reduzieren. Außerdem ist die Auflösung und somit die Genauigkeit von Überwachungseinrichtungen wie der Primär- und Sekundärradare beschränkt, dies soll durch intelligente Organisation des Flugverkehrs kompensiert werden.

In den folgenden Abschnitten wird die Basis der Luftraumorganisation erläutert, [Men04].

Fluginformationsgebiet

Der internationale Luftraum ist in Fluginformationsgebiete unterteilt. Es sind geographische Sektoren, die einen Zuständigkeitsbereich zur Regelung des Flugverkehrs definieren. Die Fluginformationsgebiete selbst können in mehreren Kontrollbezirken organisiert sein. Diese Gebietshierarchie ermöglicht eine effektive Vergabe von Zuständigkeiten

an unabhängige, jedoch vernetzte Organe, die den Flugverkehr innerhalb ihrer Bereiche sichern.

Gerade bei internationalen Gebietsgrenzen spielen sog. *Coordination Exit/Entry Points* eine wichtige Rolle, diese markieren Wegpunkte für die Übergabe des Flugverkehrs von einem FIR[1] in den angrenzenden Kontrollsektor.

Routensystem

Ähnlich dem Straßenverkehrsnetz hat auch der Luftraum fest definierte Flugrouten – die sog. *Air Traffic Service Routes* – kurz ATS-Routes. Der Flugverkehr hat diesen Routen mit einer bestimmten Genauigkeit zu folgen.

Die ATS-Routen ermöglichen erst eine Sicherung des Flugraums mit hohen Verkehrsdichten. Eine ergänzende Routenführung findet im Nahbereich von Flughäfen statt, hier existieren zusätzlich die *Standard Arrival Routes* und die *Standard Instrument Departure Routes*. Die *Standard Arrival Routes* – kurz STAR – führen die Luftfahrzeuge von den ATS-Routen zum Flugplatz, während die *Standard Instrument Departure Routes* – kurz SID – die Wegpunkte von Flugplatz zu den ATS-Routen setzten.

Flugflächensystem

Zur Vermeidung von Kollisionen auf dem Routennetz und zur Ausnutzung der technischen Möglichkeiten eines Luftfahrzeugs – der Navigation auf verschiedenen Höhen – wird eine Höhenseparation durchgeführt.

Für jeden Routenabschnitt werden hierfür mehrere Flughöhen festgesetzt, auf denen die Luftfahrzeuge navigieren können. Die Angabe der Flughöhe wird in Flight Levels – kurz FL – angegeben, wobei ein FL 100 ft barometrischer Höhe entspricht, umgerechnet ca. 30,48 Meter.

3.1.2. Staffelungsverfahren

Die Staffelung bildet eine Reihe von Verfahren, die zur Separation von Luftfahrzeugen verwendet werden, [Men04]. Es gibt besondere Staffelungsverfahren für Landeanflüge und Startvorgänge, da diese Arbeit jedoch nur die Separation im Streckenflug – also

[1] Flight Information Region – das Fluginformationsgebiet ist ein Luftraumsektor, in dem Fluginformations- und Alarmdienste angeboten werden

nach Abschluss von Start- und Landemanövern – gewährleisten soll, werden diese nicht weiter erläutert.

Längsstaffelung

Im Folgenden wird lediglich die Längsstaffelung nach Zeit beschrieben, da nur diese Form der Längsstaffelung für das Projekt relevant war.

Bei der Längsstaffelung nach Zeit werden Mindestabstände zwischen Luftfahrzeugen erwirkt, indem man minimale Zeitabstände zwischen zwei aufeinander folgenden Flügen an bestimmten Wegpunkten durchsetzt. Dabei wird bei der Festlegung der minimalen Zeitabstände eine Klassifizierung nach Verkehrsrichtung etabliert:

- Verkehr in gleicher Richtung
- Kreuzender Verkehr
- Gegenverkehr

Bei Verkehr in gleicher Richtung gilt ein Mindestzeitabstand von 10 Minuten, 5 Minuten bei großem Geschwindigkeitsvorteil des voraus fliegenden Luftfahrzeugs. Wechselt ein Luftfahrzeug die Flugfläche und kreuzt dabei die Flugbahn eines anderen Flugobjekts, spricht man von kreuzendem Verkehr. Hier gilt ein Mindestabstand von 5 Minuten zum Zeitpunkt des Kreuzen. Bei Gegenverkehr gilt es spätestens 10 Minuten vor dem prognostizierten Zusammentreffen, die Durchführung einer Höhenstaffelung einzuleiten. Nach dem Überflug kann die Höhenstaffelung ab 5 Flugminuten Abstand aufgehoben werden.

Höhenstaffelung

Für die vertikale Separation wird der Luftraum in zwei Bereiche geteilt, wobei jeweils verschiedene Separationsregeln gelten. Für den Luftraum unter Flugfläche 290 gilt ein Mindestvertikalabstand von 1000 ft.

Da die Genauigkeit der Barometertechnik zur Bestimmung der Flughöhe durch den Luftdruck in größeren Höhen abnimmt, lag die historische minimale Vertikalseparation für Flüge oberhalb der Flugfläche 290 bis Flugfläche 410 bei 2000 ft. Mit dem Fortschritt der Technik wurden die Geräte präziser und auch in großen Höhen verlässlicher. Seit 2005 gilt deshalb in den meisten Lufträumen der westlichen Hemisphäre auch in

den Höhen oberhalb der Flugfläche 290 ein vertikaler Mindestabstand zwischen zwei Luftfahrzeugen von 1000 ft. Die Voraussetzung hierfür ist eine technische Ausrüstung, die festgelegte Kriterien der Genauigkeit und Robustheit durch Redundanz erfüllt. Die Regelung nennt sich *Reduced Vertical Separation Minima* – kurz RVSM – oder ins Deutsche übersetzt *reduzierte Vertikalstaffelung*.

Slot

Für die sichere Durchführung der Flugsicherung ist es notwendig, die vergebenen Zeiten möglichst genau einzuhalten. Die Dynamik des Luftverkehrs verhindert jedoch zumeist eine sekundengenaue Präzision, durch die Addition minimaler Abweichungen entsteht Potential für Gefahrensituationen. Um diesem Potential entgegenzuwirken, werden in der Flugsicherung stets Zeitfenster vergeben. Die Zeitfenster – auch *Slots* genannt – bieten minimalen Spielraum für die Piloten und Fluglotsen. Werden die vergebenen Zeiten nicht eingehalten, so gilt es lediglich im Rahmen eines Zeitfensters zu bleiben, um die Sicherheit des Flugverkehrs nicht zu beeinträchtigen.

Solche Slots werden für die Start- und Landemanöver vom Flughafen und der zuständigen Flugsicherungsbehörde vergeben und für die Überflüge über den gesetzten Wegpunkten von den Lotsen. Werden Slot-Zeiten verletzt, so muss dieser Zustand in die Aufmerksamkeit des zuständigen Lotsen gebracht werden, es besteht möglicherweise Handlungsbedarf um die Sicherheit zu erhalten.

3.1.3. GCAA

Die *General Civil Aviation Authority* (GCAA) ist die Flugsicherungsbehörde der *Vereinigten Arabischen Emirate*. In Zusammenarbeit mit den lokalen Autoritäten hat die GCAA die Sicherung des Flugraums über Abu Dhabi, Dubai, Sharjah, Ajman, Umm al-Quwain, Ras al-Khhaimah und Fujairah als Aufgabe. Auch die regelkonforme Weiterleitung des Flugverkehrs in angrenzende Kontrollgebiete gehört zu den Zuständigkeiten der Behörde.

Die GCAA ist der Kunde, in dessen Auftrag die Abflugplanungskomponente entwickelt wurde. Zur Abflugregulierung müssen die zuständigen Fluglotsen der Flughäfen im GCAA-kontrollieren Gebiet jeden Starter registrieren. Beim Anmelden eines Starters hat die GCAA die Möglichkeit, Einfluss auf die Startzeit des Luftfahrzeugs zu nehmen. Die Bewilligung einer Startzeit soll im Sinne der Sicherheit, somit also der Einhaltung

der Staffelungsrichtlinien und eventuellen Flussdichtenbeschränkungen, stattfinden.

In Zusammenarbeit mit einem Berater der Behörde und einigen GCAA-Fluglotsen wurde die Sachlage der Flussdichtenbeschränkungen analysiert.

Beschränkungen auf Verkehrsflussdichten

Der Flugverkehr kann an geeigneten Wegpunkten reguliert werden. Das Setzen von maximalen Flussdichten bedeutet eine Begrenzung der Anzahl an Überflügen innerhalb einer Zeitspanne. Aus der Flussdichtenbeschränkung kann eine Durchflussfrequenz abgeleitet werden.

Das Ziel der Beschränkungen ist die Sicherung des Flugverkehrs über Gebieten, die aufgrund ihrer geographischen Eigenschaften nicht durch Radar überwacht werden. Im Falle des GCAA-FIR sind besonders die nordwestlichen und östlichen Routen von solchen Beschränkungen betroffen. In nordwestlicher Richtung liegen ausgedehnte Wüstengebiete, die kaum Radarüberdeckung aufweisen, da der Betrieb von Radaranlagen dort sehr kostspielig und wartungsintensiv ist. In östlicher Richtung führen die Flugrouten über den Indischen Ozean, auch hier ist zum heutigen Stand der Technik keine flächendeckende Radarüberwachung möglich. Das Abb. 3.1 veranschaulicht die Gefahrenzonen im Umfeld von den GCAA-kontrollierten Gebiet.

Die Analyse der verschiedenen Flussdichtenregelungen der GCAA hat ergeben, dass

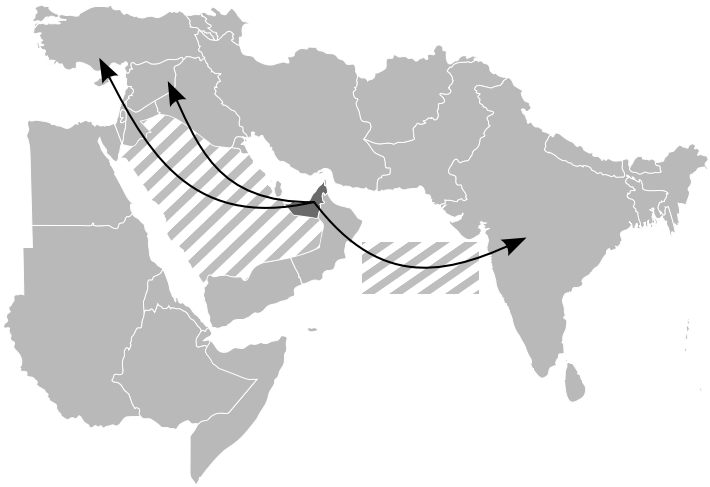


Abbildung 3.1.: GCAA FIR und umliegende Gebiete. Die gestreiften Gebiete sind größtenteils nicht von Radar überwacht. *Quelle (Weltkarte): Wikipedia*

man zusätzlich zu den regulären Beschränkungen auf einem Wegpunkt auch drei Sonderfälle betrachten muss:

- **Kombinierte Wegpunkte**
  Aufgrund eines Zusammenführens mehrerer Flugrouten im späteren Flugverlauf, gilt es bereits im kontrollierten Gebiet mehrere Wegpunkte einem abstrakten Durchflusswegpunkt – *Flow Point* genannt – zuzuordnen. Ein Flow Point wird bei der Verteilung der Zeiten als ein logischer Wegpunkt betrachtet, auch wenn die explizite Überflugzeiten dem entsprechenden Wegpunkt zugeordnet werden.

- **Routenverlauf**
  Nicht alle Routen von einem Wegpunkt müssen über das restriktive Gebiet führen. Manche Flugrouten bieten Abzweigungen außerhalb des FIR an, die in andere Gebiete ohne Restriktionen führen. Hier gilt es den den nachfolgenden Routenverlauf zu betrachten um eine positive Zuweisung zu einem Flow Point zu leisten.

- **Wegpunkte außerhalb des kontrollierten Gebiets**
  Eine besondere Regelung der GCAA fordert eine Separation auf einem Wegpunkt, der außerhalb des kontrollierten FIR liegt. Hier gilt es durch die Prognose der Überflugzeit an dem besagten Wegpunkt eine Separation an den kontrollierten Exit Points zu leisten.

Die Besonderheiten der Flow Point-Zuweisung erhöht die Komplexität der Komponenten, die eine Automatisierung der Zuordnung und Startzeitvergabe realisieren sollen. Die Einteilung der Regelungen in Äquivalenzklassen ermöglicht eine genauere Analyse der Anforderung an ein System. Sind alle Klassen identifiziert, muss die Unterstützung für alle Regelungen nicht explizit in die Anforderungen aufgenommen werden und später verifiziert werden. Die Recherche der Luftraumbeschränkungen ist die Basis der Analyse im Abschnitt 4.1.4.

Manuelle Abflugplanung

Vor der Einführung der DFLOW-Komponente, wurde der Abflugzeitpunkt manuell durch das Personal der GCAA bestimmt. Die Vorgabe für die Planer bestand darin, eine maximale Frequenz an Startern pro Flugroute nicht zu überschreiten. Hierbei war eine Berücksichtigung der Eigenschaften des Luftfahrzeugs, wie z.B. dessen Ausrüstung, Reisegeschwindigkeit und Flughöhe, nicht möglich. Die Berechnungsvektoren für eine Optimierung sind für einen Menschen zu umfangreich, die Berücksichtigung aller Eigenschaften – dazu zählt auch die aktuelle Slot-Belegung an den jeweiligen Wegpunkten

einer Route – ist bei manueller Bearbeitung nicht in einer praktikablen Zeit möglich.

Die höchste Priorität bei der Vergabe von Startzeiten gilt der Wahrung der Sicherheit. Durch die manuelle Vergabe der Abflugzeiten entsteht eine ineffektive Nutzung der Luftraumkapazitäten. Dies hat negative Folgen für die Wirtschaftlichkeit der Flugsicherung, insbesondere im Fall der GCAA und des enormen Wachstums des Luftverkehrs in deren kontrollierten Region.

3.2. PRISMA-Architektur

PRISMA von der Comsoft GmbH ist eine modulare Lösung zur integrierten Flugsicherungsautomation. Ein PRISMA-System ist eine kundenspezifische Kombination und Konfiguration von Komponenten. Alle PRISMA-Komponenten sind außerdem als eigenständige Module erhältlich und somit in andere Umgebungen integrierbar.

Die Hauptkomponenten von PRISMA sind:

- **CWP** – ***Controller Working Position***
  Die CWP ist das Display für die Lotsen und die Lotsenaufsicht. Es bildet das Kernstück des PRISMA-Frontends und bietet ein interaktives Luftlagebild über den kontrollierten Sektor. Alle Komponenten des PRISMA werden in diesem Display integriert, so liefert das SDPS die Tracks[2], das FDPS die Flugpläne während die Konfliktwarnungen des Safety Net im Display visualisiert und mit einem Audiosignal versehen werden.

- **AWP** – ***Assistent Working Position***
  Die AWP ist das Display für die Assistent Controller – auch Planer genannt. Das Display bietet eine strategische Ansicht, geeignet zur Vorverarbeitung der Flugplandaten, Zuweisung von Flugplänen zu Luftfahrzeugen und Vergabe von Abflugslots. Zusätzlich stellt es Werkzeuge bereit zur Bildung und Darstellung von Prognosen über das Verkehrsaufkommen.

- **FDPS** – ***Flight Plan Data Processing System***
  Das FDPS ist der Bearbeitung von Flugplandaten dediziert. Es werden u.a. dynamische Flugplanprofile erstellt, Überflugzeiten prognostiziert und Vorhersagen über das Verkehrsaufkommen entwickelt.

[2]Der Flugverlauf eines Luftfahrzeugs basierend auf der Integration verschiedener Überwachungsdaten

- **SDPS – *Surveillance Data Processing System***
  Das SDPS bietet die Integration von mehreren heterogenen Datenquellen der Luftraumüberwachung an. Das System unterstützt Primär- und Sekundärradare, Mode-S, ADS-B/C und Multilateration als Datenquellen.

- **Safety Net**
  Zur Unterstützung der Fluglotsen validieren die sog. Safety Nets die Flugsicherheit. Die Prozesse suchen den Flugraum nach möglichen Gefahrenquellen ab und melden potentielle Verstöße gegen die Regelungen der Flugsicherung.

- **Daten-, Audio- und Displayaufzeichnung**
  Eine sorgfältige Protokollierung der Datenströme ist die Grundlage der Nachvollziehbarkeit von Fehlerereignissen und der Nachvollziehbarkeit von Benutzerentscheidungen. PRISMA bietet hierfür eine Reihe von Werkzeugen an, von der automatisierten Aufzeichnung aller Datenbanktransaktionen bis zur synchronisierten Funkverkehr- und Displayaufzeichnungen.

3.2.1. DMAP

Eine Besonderheit in der PRISMA-Architektur stellt das Modell der Datenkommunikation dar. PRISMA besitzt eine verteilte Datenbank, die im höchsten Maße die Robustheit der beteiligten Systeme und die Sicherheit der Daten garantiert.

Die DMAP ist eine sichere Kommunikationsschicht auf der UDP[3]/IP-Schicht zur Realisierung von Transaktionen auf den persistenten Datensätzen. Die Datensätze sind in semantischen Gruppen organisiert, wie z.B. die `SFPL` als Datensatztyp für sie *System Flight Plans* oder `FPATH` für die Routenzusatzinformationen.

Jede Komponente von PRISMA, die mit der DMAP interagiert, hält stets lokale Kopien der relevanten Datensätze, Änderungen werden durch das DMAP-Protokoll zur *primären Datenbank* und von dort zu allen verteilten Abonnenten propagiert. Jegliche Interaktion zwischen PRISMA-Komponenten wird durch diese Schicht realisiert, der datengetriebene Ansatz ermöglicht eine hohe Dynamik in der Entwicklung und Integration neuer Komponenten und minimiert die Notwendigkeit feste Schnittstellen zu pflegen.

[3]User Datagram Protocol – ein verbindungsloses, nachrichtenbasiertes Internetprotokoll der Transportschicht

Dieses Modell bietet zudem einen hohen Grad an Redundanz der Daten. Die lokalen Kopien der Komponenten befähigen diese auch bei Abbruch der Verbindung zu dem Server weiterhin operativ zu bleiben, Benutzereingaben zu verarbeiten und bei Wiederaufnahme der Verbindung die Transaktionen fortzusetzen.

3.3. Compilerbau

Zur effektiven Modellierung der Luftraumbeschränkungen soll eine domänennahe Konfigurationssprache entwickelt werden. Für diesen Zweck galt es einen Compiler zu entwickeln, der die Sprache akzeptiert und das Modell der Luftraumbeschränkungen in ein geeignetes Format zur dynamischen Weiterverarbeitung überführt. Dieser Abschnitt ist die Zusammenfassung der Recherche von Techniken der Compilerentwicklung und der vorhandenen Technologien.

Die Übersetzung eines Programms von einer Sprache in eine andere Sprache wird von einem Compiler durchgeführt. Es gibt verschiedene Gründe für die Notwendigkeit einer Übersetzung, die Gemeinsamkeit liegt in der Zugänglichkeit und erhöhten Effizienz, die dadurch für die Entwicklung erreicht wird.

Eine Programmiersprache soll die Kommunikation zwischen Mensch und Computer erleichtern. Die Sprache ist an die menschlichen Bedürfnisse angepasst und erlaubt im besten Fall eine Abstraktion über das darunter liegende System. Das Übersetzen des Programms in ein Programm in Maschinensprache ermöglicht letztendlich das Ausführen des Programms auf einem Computer.

In manchen Situationen bietet es sich an, eine weitere plattformunabhängige Schicht dazwischen zu setzen, die sog. Virtuelle Maschine. Diese Schicht bedient sich der Schnittstellen des Systems und bietet dafür eine abstrahierte Laufzeitumgebung an. Eine Virtuelle Maschine kann somit plattformunabhängigen Bytecode auf verschiedenen System ausführen.

Eine wichtige Rolle eines Compilers ist es, Fehler im Programmcode zu berichten, die während der Übersetzung entdeckt wurden. Zur effektiven Auflösung von auftretenden Fehlern gilt es eine möglichst genaue Beschreibung und Lokalität des Fehlers zu liefern. Zur Reduzierung von wiederholten Übersetzungsvorgängen ist es hilfreich

möglichst viele Fehler in einem Durchlauf zu erkennen. Hierfür ist es notwendig Richtlinien zu setzen, die eine Fortsetzung des Übersetzungsvorgangs ermöglichen und das Entstehen von Seiteneffekten durch vorhergegangene Fehler vermeiden.

3.3.1. Compilerarchitekturen

Nach [ALSU07] fordern verschiedene Anwendungsgebiete bestimmte Eigenschaften von einem Übersetzer. Der traditionelle Compiler übersetzt Quellcode von einer Programmiersprache in ein lauffähiges Programm. Das Resultat ist eine ausführbare Datei, welches bestimmte Eingaben erwartet, auf denen es die programmierten Funktionen ausführt. So ein Programm ist systemabhängig (Abb. 3.2).

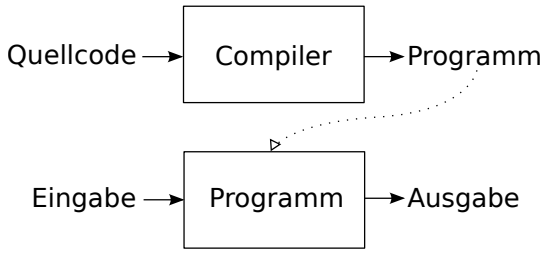


Abbildung 3.2.: Ein Compiler. *Quelle: [ALSU07]*

Einen anderen Ansatz bietet der Interpreter (Abb. 3.3), welcher sowohl den Quellcode als auch die Eingabeparameter erhält und daraus direkt die Ausgabe produziert. Der Übersetzungsprozess passiert iterativ für jeden Befehl, was Optimierungsmaßnahmen schwierig macht. Ein Interpreter wird eingesetzt, wenn ein interaktives Werkzeug gebraucht wird und die Berechnung der Ausgabe nicht zeitkritisch ist.

Abbildung 3.3.: Ein Interpreter. *Quelle: [ALSU07]*

Will man jedoch ein systemunabhängiges Programm erstellen, welches eventuell sogar auf das jeweilige Zielsystem optimiert werden soll, lässt sich ein sog. hybrider Compiler samt virtueller Maschine einsetzten (Abb. 3.4).

In diesem Fall übersetzt der Compiler den Quellcode in ein sog. *Intermediate Program*. Das Resultat ist ein systemunabhängiger Bytecode, welcher als Eingabeprogramm für die virtuelle Maschine genutzt wird. Die virtuelle Maschine ist nun in der Lage, systemabhängige Optimierungen am Programm durchzuführen. Zusammen mit den Eingabeparameter, führt die virtuelle Maschine die Berechnungen durch und erzeugt somit die Ausgabe.

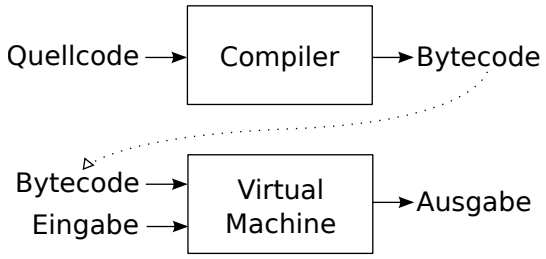


Abbildung 3.4.: Ein hybrider Compiler. *Quelle: [ALSU07]*

3.3.2. Werkzeugunterstützung

In der Analysephase eines Übersetzers wird der Eingabecode analysiert und auf Fehler überprüft, um aus dem validen Code die Überführung in das Back-End zu gewährleisten. Das Front-End eines Compilers wird auch Parser genannt. Die Geschichte der Compiler-Entwicklung hat seit dem ersten Compiler im Jahre 1952, welches von Grace Hopper entwickelt wurde, die später auch an der Entwicklung von *COBOL* beteiligt war, enorme Fortschritte gemacht. Mittlerweile stehen für die Entwicklung eines Übersetzers eine Vielzahl an Werkzeugen zur Verfügung, die jede Phase des Compilers abdecken.

Für die Entwicklung des Compilers für die Konfigurationssprache wurden verschiedene Technologien evaluiert:

- LLVM[4]
- `lex`
- `yacc`
- `flex/flex++`
- `bison/bison++`

[4]Low Level Virtual Machine – eine modulare Compilerarchitektur mit optimiertem Übersetzungskonzept

LLVM

LLVM ist eine Compiler-Infrastruktur zur Optimierung von Programmen, die in beliebigen Sprachen erstellt werden können. In einigen Anwendungsgebieten – wie z.B. in der Entwicklung von Objective-C-Programmen – soll es den `GCC`[5] ersetzen. Der Vorteil von *LLVM* liegt in der modernen Architektur des Frameworks und den fortgeschrittenen Optimierungsmechanismen. Auch die Integration mit anderen Werkzeugen, wie z.B. einer IDE[6], sollen mit *LLVM* erleichtert werden.

`lex` und `flex`

Die Erstellung der lexikalischen Analyse ist der erste Schritt in einer Parser-Entwicklung. `lex` bietet eine Möglichkeit die Analyse mit Hilfe einer Konfiguration zu automatisieren. Das Resultat ist ein Scanner in der Programmiersprache C.

`flex` ist eine unter *GNU Public License* entwickeltes Kommandozeilen-Tool, welches die Funktionen des `lex` implementiert und darüber hinaus Erweiterungen bietet. `flex++` bietet eine Reihe von C++-Klassen an, die eine Integration in einen C++-Parser erleichtern sollen. Auch der generierte Code ist dann in C++ verfasst.

`yacc` und `bison`

Dieses Kommandozeilenwerkzeug soll die Parser-Generierung automatisieren. Der Name steht für *Yet Another Compiler-Compiler*, was jedoch irreführend sein kann, denn es handelt sich definitiv nicht um einen Compiler-Generator. Mit einer Konfigurationsnotation wird die Syntax einer Sprache definiert, `yacc` generiert daraus C-Code, der die Syntaxanalyse realisiert. `bison` ist ein Werkzeug, welches die Funktionalität des `yacc` kopiert, jedoch unter der *GNU Public License* erhältlich ist. `bison++` bildet das C++-Pendant und bietet eine Integration mit `flex++` an.

3.4. Sicherheitsfaktoren

Systeme im Einsatz zur Flugsicherung müssen bestimmte Richtlinien der Sicherheit erfüllen. U.a. werden diese Richtlinien durch den *IEC 61508 Standard* festgelegt, der die *Funktionale Sicherheit sicherheitsbezogener elektrischer/elektronischer/programmierbar elektronischer Systeme* definiert.

[5] GNU Compiler Collection – eine Compilersammlung für u.a. C, C++ und Java unter GPL-Lizenz
[6] Integrated Development Environment – eine integrierte Entwicklungsumgebung

Als Planungskomponente erfüllt das DFLOW-System zwar keine unmittelbaren sicherheitsbezogenen Aufgaben, jedoch sollen die Richtlinien die Grundlage bilden für eine reibungslose Integration in das sicherheitskritische PRISMA-System.

3.5. Produktvergleich

Auf dem Markt befinden sich eine Reihe von Systemen, die das Abflugmanagement und Funktionen des *Air Traffic Flow Managements* automatisieren. Die Systeme bieten eine große Vielfalt an Ansätzen zur Bewältigung der Aufgaben an, exemplarisch werden drei Systeme kurz erläutert.

3.5.1. CFMU

Das *Central Flow Management Unit* – oder auch CFMU – ist ein koordiniertes, zentrales System zur Regelung des Luftverkehrsflusses im europäischen Raum. Das System wird von EUROCONTROL[7] betrieben, ausführliche Dokumentation ist online unter `www.cfmu.eurocontrol.int` erhältlich.

CFMU bietet eine Architektur basierend auf Web Services. Es werden eine Reihe von Diensten zur Vorbereitung und Übermittlung von Flugplänen, zur Visualisierung von Routen und Flugräumen und zur Erstellung von statistischen Reports angeboten. Es gibt eine große Bandbreite an Werkzeugen zur Interaktion mit diesen Diensten.

CFMU ist ein für den europäischen Flugraum konzipiertes System, die Funktionen realisieren somit nur die europäischen Regelungen. Es gibt keine Möglichkeit die Dienste für einen anderen Flugraum zu konfigurieren und die Systeme auf einem abgeschlossenen System, unabhängig von dem Internet, zu betreiben.

3.5.2. PATS Departure Manager

Ein weiteres von EUROCONTROL in Auftrag gegebenes System ist der *PATS Departure Manager*. Im Rahmen eines *PHARE*[8]-Projekts wurde ein Abflugmanager entwickelt, der europäischen Standards entspricht. Die Gewichtung der Aufgaben des *PATS Departure Manager* liegt bei der Sicherung des Abflugverkehrs unter Betrachtung der

[7] Die europäische Organisation für Flugsicherung
[8] Programme for Harmonised Air Traffic Management Research in EUROCONTROL

Trajektorie der Luftfahrzeuge während der Startmanöver [DME99]. Mit Hilfe der *PATS Conflict Probe* wird eine optimale und konfliktfreie Auflösung des startenden Flugverkehrs realisiert, während der *PATS Trajectory Predictor* die Trajektorieberechnungen aller Alternativen durchführt. Hierbei bilden die SIDs-Wegpunkte die Route zur Integration der Starter in die Reiseverkehrsrouten.

Neben einer Integration mit einem Arrival Manager zur Regelung von sog. Mixed Mode Airways[9] realisiert ein Display die Interaktion mit den Komponenten.

Der *PATS Departure Manager* realisiert laut [DME99] 2.3.3 keine Optimierung des Flugverlaufs hinsichtlich der weiterführenden Route und deren Verkehrsflussdichten. Auch die Separation nach Flugfläche ist nicht Bestandteil der Funktionalität.

3.5.3. Departure Manager Frankfurt

Eine weitere Entwicklung im Bereich Abflugmanager stellt der *darts4D DMAN* von *Delair* dar. Das System soll verbindliche Aussagen über die Startzeit, bzw. die sog. *Off-Block Time*, unter Berücksichtigung aller Prozesse und Richtlinien des Flughafens machen. Der Departure Manager wurde im April 2007 in Frankfurt in Betrieb genommen und aufgrund von Beeinträchtigung des Abflugmanagements wenige Monate später wieder aus dem System genommen [Klu07].

Interessant hierbei sind die genannten Gründe für den Fehlschlag: eine unausreichende Einbindung der Endanwender und Domänenexperten in den Entwicklungsprozess sollen laut [Klu07] die Hauptgründe für die nicht optimale Konzeption gewesen sein. Ein weiterer Grund werden die komplexen Abhängigkeiten der vielen Parameter sein, die zum erfolgreichen Mikromanagement eines Abflugs in die Berechnung aufgenommen werden. Die Eingangsparameter sind zudem durch äußere und menschliche Einflüsse höchst dynamisch, was die Zuverlässigkeit der Prognosen reduzieren kann.

[9] Startbahnen, die sowohl für Start- als auch Landeverkehr freigegeben sind

4. Anforderungsanalyse

Die Grundlage der Anforderungsanalyse bestand aus den Benutzeranforderungen, die zusammengefasst in einem Dokument von der GCAA bereitgestellt wurden. Diese Anforderungen wurden schrittweise mit Hilfe von sog. User-Stories in einen Kontext gebracht um dadurch etwaige Uneindeutigkeiten zu beseitigen.

Für jede Benutzeranforderung wurde mindestens eine Anforderung an das System entwickelt. Dabei wurden entstehende Anforderungsabhängigkeiten durch das Extrahieren einer unabhängigen Anforderung aufgelöst. Bei der Formulierung der Anforderungen wurde strikt zwischen Anforderung an die Sprache, an das verarbeitende System und an das Benutzer-Interface getrennt. Diese Trennung ermöglichte im weiteren Verlauf die Erarbeitung der Anforderungen an die Komponenten, insbesondere an die Syntax der Konfigurationssprache.

4.1. Modellierung der Luftraumbeschränkungen

Jeder Luftraum besitzt dynamische Beschränkungen auf die Verteilung des Luftverkehrs. Die Beschränkungen dienen maßgeblich zur Gewährleistung der Sicherheit des Luftraums unter Berücksichtigung der geografischen, politischen und technischen Merkmalen der ATS-Routen. Die Dynamik der Regelungen wird durch die Abhängigkeit von Tageszeiten, den Eigenschaften eines Luftfahrzeugs und dessen Flugplan und den vorherrschenden Luftraumbedingungen bestimmt.

Die Modellierung der Luftraumbeschränkungen soll durch eine Konfigurationssprache realisiert werden. Zur Festlegung der Merkmale für die Konfigurationssprache wurden folgende Aspekte berücksichtigt:

- Benutzerqualifikation
- Sicherheitsfaktoren

- Domänenkontext (Flugplandaten, Flussdichtenregelungen)

4.1.1. Benutzerqualifikation

Die Sprache soll eine domänennahe Konfigurationsmöglichkeit bieten, dabei Redundanz und Komplexität in der Syntax vermeiden. Der typische Benutzer der Sprache hat eine Ausbildung im Bereich Flugsicherung und ist technisch versiert. Anhand dieser Voraussetzungen entstanden Prototypen für die Syntax der Sprache, die vom Entwicklungsteam evaluiert wurden.

Die Analyse ergab eine deklarative Sprache, die explizite Syntaxelemente aufweist. Eine deklarative Sprache beschreibt das formale Modell der Berechnung. Im Gegensatz dazu wird beim imperativen Programmieren der Berechnungsalgorithmus in einzelnen zustandsverändernden Anweisungen verfasst. Zur Vereinfachung kann man sagen, dass der deklarative Ansatz eine zielorientierte Kodierung bedeutet, während beim imperativen Programmieren der Lösungsweg, also der Algorithmus kodiert wird. Die zielorientierte Kodierung bedeutet eine Abstraktion vom darunter liegenden Datenmodell und Berechnungsweg, dies ermöglicht den Einsatz der Sprache in dem Domänenumfeld unter Berücksichtigung der Benutzerqualifikationen.

Als Vorlagen wurden die Logikprogrammiersprache Prolog für die Regeldefinitionen und die Datenbanksprache SQL[1] für die Definition der Regelbedingungen genommen. Gerade SQL eignet sich in Hinsicht der Nähe zur natürlichen Sprache und der expliziten Ausdrücke für diesen Einsatz.

4.1.2. Sicherheitsfaktoren

Die Konfiguration der Abflugplanungskomponente hat Einfluss auf die Verarbeitungsprozesse und ist somit weit möglichst auf Korrektheit zu prüfen. Die Prüfung der Konfiguration soll in der Initialisierungsphase des Systems stattfinden, um die sichere Operation eines Systems zu garantieren, [Com05]. Bei statisch typisierten Sprachen werden bereits beim Übersetzen die Datentypen festgelegt und somit überprüfbar gemacht. Dies ermöglicht es, eine große Anzahl an möglichen Fehlern in der Konfiguration zu erkennen, bei denen ungültige Kombinationen von Operation und Datentyp beste-

[1] Structured Query Language – Sprache zur Definition, Manipulation und Abfrage von Daten in relationalen Datenbanken

hen.

Jede Funktionalität eines Systems muss stets überwacht werden. Bei Fehlverhalten soll eine Wiederaufnahme der Funktionalität gewährleistet werden, ohne die Datenintegrität oder andere Prozesse des Systems zu gefährden.

Zur Sicherung der Daten wird eine strikte Trennung von Daten und informationsverarbeitenden Prozessen durchgeführt. Jede Transaktion wird geprüft, protokolliert und redundant propagiert. In der Flugsicherung ist die redundante Auslegung von Komponenten Kernbestandteil bei der Sicherung der Verfügbarkeit eines Systems. Die letzte Option bei dem Auflösen eines Systemkonflikts liegt in der Abschaltung und Neuinitialisierung eines Systems, während die redundante Auslegung weiterhin die Funktionalität sicherstellt.

Statusmeldungen

Durch das *Watchdog*-Protokoll ist es möglich die Vitalität einer Komponente zu überwachen. In periodischen Abständen wird dabei von einem Überwachungsprozess eine Anfrage an alle Komponenten gestartet, die innerhalb einer bestimmten Zeit beantwortet werden muss. Die Antwort kann detaillierte Informationen über den Status einer Komponente enthalten, oder lediglich deren Aktivität bestätigen.

Bei Überschreitung des Zeitlimits für die Antwort wird von einer Fehlfunktion der Komponente ausgegangen. Wird das Zeitlimit wiederholt in Folge überschritten, können Prozesse wie z.B. der Neustart der Komponente zur Wiederherstellung der Vitalität in die Wege geleitet werden. Die Wiederaufnahme der Funktionalität soll in jeden Fall möglich sein.

Redundanz

Um eine erfolgreiche Migration der Aktivitäten von einem System auf ein Ersatzsystem zu gewährleisten, muss die Komponente dafür konzipiert sein. Die Komponente soll auf flüchtige Datenhaltung verzichten, um den Verlust bei einer Migration in Grenzen zu halten. Es soll auf externe Ressourcen wie z.B. File Handles verzichtet werden, da die Freigabe solcher Ressourcen systemabhängig ist und somit nicht garantiert werden kann.

Sowohl das Reinitialisieren als auch die Migration eines Prozesses gibt folgende Richtlinien für die Entwicklung der Komponenten vor:

- Kein flüchtigen Daten
- Verarbeitungszeiten minimieren
- Transaktionsmodell zur Persistenzschicht
- Keine externen Abhängigkeiten zur Laufzeit

Die PRISMA-Architektur bietet ein Datenprotokoll zur persistenten Datenhaltung mit der `DMAP`-Komponente an. `DMAP` ist eine objektorientierte, verteilte Datenbank basierend auf einer dateibasierten Datenhaltung. Die Kommunikation zwischen den Komponenten und der Datenbank ist auf einer gesicherten UDP-Schicht realisiert. Weiterhin bietet PRISMA eine Klasse zur Realisierung des Watchdog-Protokolls. Somit verlagern sich ein Großteil der Anforderungen auf die bestehende Architektur, was die Entwicklung der Komponenten erleichtert.

4.1.3. Flugplandaten

Um erfolgreich die Sprache in den Kontext Flugsicherung zu integrieren, wurden die Standardflugpläne analysiert und deren Zusammensetzung erfasst. Die einzelnen Merkmale eines Flugplans wurden mit entsprechenden Datentyp in die Syntax der Sprache aufgenommen, [Org96]. Die Kodierung der Flugpläne basiert auf dem ASCII[2]-Zeichensatz, in den folgenden Abschnitten soll ein Zeichen bzw. Zeichenketten jeweils auf dem ASCII-Zeichensatz basieren.

Aerodrome

Der Flughafen wird in Flugplänen nach [Org96] mit vier Buchstaben kodiert. Jedem internationalen Flughafen wird ein dedizierter Kode zugewiesen, der in den Flugplänen als Abflughafen und Zielflughafen zur Identifikation genutzt wird. Abkürzungen, die oft benutzt werden, lauten *ADEP* oder *DEP* für den *Aerodrome of Departure* und *ADES* oder *DEST* für *Aerodrome of Destination*. Zur Berücksichtigung der Flughäfen soll in der Sprache die Feldnamen *ADEP/ADES* für Startflughafen, respektive Zielflughafen, reserviert werden. Zulässige Werte für sind Zeichenketten zusammengesetzt aus vier Buchstaben.

[2] American Standard Code for Information Interchange – ein Standard zur Zeichenkodierung für u.a. das lateinische Alphabet und die arabischen Ziffern

Runway

Die Bezeichnungen für die Start- und Ladebahnen – Englisch *Runway* – können lokal vergeben werden und sind meist eine Kombination aus mehreren Buchstaben und darauf folgenden Zahlen, z.B. RWY01. Es sind bis zu fünf Zeichen für die Identifikation erlaubt, die Abkürzung *RWY* soll das Feld in der Sprache bezeichnen.

Aircraft Type

ICAO legt einen vierstelligen Identifikationskode für die Flugzeugtypen fest. Die Kodierung erlaubt sowohl Buchstaben als auch Zahlen. Beispielkodierungen sind A332 für den *Airbus A330-200* oder B744 für eine *Boeing 747-400*. Gebräuchliche Abkürzung ist *ATYP*. In der Sprache soll das Schlüsselwort *ATYP* für den Flugzeugtyp reserviert sein, zulässige Werte sollen alle Zeichenkette der Länge vier darstellen.

Flight Type

Der *Flight Type* beschreibt den Einsatztyp des Fluges, sei es ein Passagierflug, Transportflug oder ein militärischer Einsatz. Hierfür wird ein Zeichen zur Identifizierung genutzt. Die Abkürzung hierfür ist *FTYP*. Die Sprache soll als Wertemenge alle Zeichen akzeptieren.

True Airspeed

In der Luftfahrt gibt es verschiedene Möglichkeiten die Fluggeschwindigkeit eines Luftfahrzeugs zu ermitteln. Die *True Airspeed* bezeichnet die Geschwindigkeit des Luftfahrzeugs relativ zu den Luftmassen, korrigiert nach den Druck- und Temperaturverhältnissen. Die abkürzende Bezeichnung lautet *TAS*. Die Modellierungssprache soll den Feldnamen *TAS* für diese Flugplaneigenschaft reservieren, eine natürliche Zahl dient als zulässiger Wert, die implizite Einheit ist der Knoten.

4.1.4. Flussdichtenregelungen

Eine Äquivalenzklassenanalyse der möglichen Regeln für die Definition von Flussdichten eines Fluginformationsraums ergab eine Anzahl an Regeltypen, die von der Sprache unterstützt werden mussten (siehe Abschnitt 3.1.3). Zur erfolgreichen Definition jeder möglichen Regeln stellte sich die Boolesche Algebra als ausreichend heraus.

Die einzelnen Regelterme sind somit entweder Teilmengenrelationen oder Äquivalenzrelationen, mehrere Regelterme können mit logischen Und- und Oder-Operatoren verknüpft werden. Die Negation dient zum Ausschluss eines Zustands, die Klammersetzung soll die gewünschte Priorität der logischen Auswertung eines Terms vorgeben.

4.2. Abflugplanungskomponente

Die Komponente soll in die PRISMA-Architektur integriert werden, wobei die Komponente als flugplanverarbeitendes Aggregat dem FDPS zugeordnet wird. Die Schnittstelle zum Gesamtsystem wird durch die DMAP gewährleistet, welche die Kommunikationsschicht für ein verteiltes objektorientiertes Datenbanksystem bildet.

Anhand eines Ereignismodells soll die Komponente auf bestimmte Dateneinträge reagieren, diese bearbeiten und zurück in das System geben. Es gibt zwei mögliche Ereignisse, die eine Bearbeitung initiieren können:

- Benutzeranfrage vor Startfreigabe
- Überflüge

Die Kommunikation mit dem Display, welches die Benutzerinteraktion mit der Komponente gewährleistet, ist ebenfalls durch die DMAP-Kommunikationsschicht entkoppelt. Die strikte Trennung der Komponenten erleichtert u.a. die unabhängige Entwicklung der beiden Module und erhält die Modularität der PRISMA-Architektur. PRISMA erlaubt es dadurch ein System je nach Wunsch des Kunden mit beliebigen Komponenten auszustatten. Auch Fremdkomponenten können integriert werden, solange diese die entsprechenden Standards, wie z.B. das Flugplanformat, unterstützen. In diesem Fall wurde das Display für die Abflugplanungskomponente unabhängig entwickelt und ist nicht Bestandteil dieser Arbeit.

Die besonderen Anforderung an die Komponente sind:

- Robustheit
- Kurze Bearbeitungszeit
- Keine Systembeanspruchung bei Inaktivität

4.3. Musskriterien

Die folgende Anforderungen sind Kernbestandteil der Komponenten:

1. **Dynamische Flugplanfilterung**
   Die Abflugplanungskomponente soll anhand von Flugplandaten eine Zuordnung zu Flow Points setzen. Die Zuordnung soll anhand der Flugplaneigenschaften und der Tageszeit entschieden werden.

2. **Berechnung der optimalen Abflugzeit**
   Die Abflugplanungskomponente soll anhand von Flugplandaten die optimale Abflugzeit, in Abhängigkeit aller Restriktionen der betreffenden Flow Points, berechnen.

3. **Zuweisung der optimalen Flugfläche**
   Die Abflugplanungskomponente soll anhand von Flugplandaten die optimale Flugfläche, in Abhängigkeit aller Restriktionen der betreffenden Flow Points, bestimmen. Die Höhenstaffelung soll die bevorzugte Staffelungsart der Komponente sein.

4. **Manuelle Übersteuerung**
   Zu jeder Zeit hat der Bediener der Abflugplanungskomponente Möglichkeiten die automatisch zugewiesenen Werte zu ändern. Manuelle Änderungen sind dauerhaft und keinen weiteren automatischen Überprüfungen ausgesetzt. Manuell angepasste Einträge sollen bei nachfolgenden Berechnungen beachtet werden.

5. **Konfigurierbarkeit**
   Die Abflugplanungskomponente soll konfigurierbar sein. Die Konfiguration soll alle möglichen Staffelungskriterien der GCAA-FIR modellieren können. Die Konfiguration soll alle relevanten Flugplandaten berücksichtigen.

6. **Transaktionsprotokollierung**
   Alle Vorgänge der Abflugplanungskomponente sollen in persistenter Form protokolliert werden. Dies umfasst die Benutzereingaben und automatischen Wertsetzungen der Komponente. Die Protokolle sollen vollständig sein, um die Nachvollziehbarkeit aller Aktionen und Entscheidungen zu gewährleisten.

4.4. Sollkriterien

Die folgende Anforderung *sollen* berücksichtigt werden:

1. **Statistische Protokollierung**
   Für jedes von der Abflugplanungskomponente erfasstes Luftfahrzeug soll ein Eintrag in einer persistenten Logdatei erstellt werden. Die Einträge sollen die Anfragezeit, die früheste Abflugzeit, die optimale Abflugzeit, die tatsächliche Abflugzeit neben einer Reihe von Flugplandaten zur eindeutigen Identifizierung des Fluges erfassen. Das Format der Logdatei soll zur statistischen Auswertung geeignet sein, z.B. CSV[3].

2. **Fehleranalyse der Konfiguration**
   In der Initialisierungsphase der Abflugplanungskomponente soll eine Analyse der Konfiguration stattfinden. Es sollen alle Fehler in möglichst expliziter Form dargestellt werden.

3. **Hohe Effizienz**
   Die Abflugplanungskomponente soll unmittelbar auf Anfragen reagieren, die Kalkulationszeit soll minimal gehalten werden, sodass der Arbeitsablauf der bedienenden Personals nicht gestört wird. Die Effizienz der Komponente soll sowohl im Bearbeitungszeitraum als auch während der Inaktivität hoch sein, die Belastung auf das Gesamtsystem soll minimiert werden.

4. **Robustheit**
   Die Abflugplanungskomponente soll sich stets in einem validen Zustand befinden. Die Komponente soll Fehlbedienung, Auftritt von fehlerhaften Daten und Berechnungsfehler korrekt behandeln und eine Fortsetzung der Funktionalität garantieren. Die Komponente soll nach den Richtlinien von IEC 61508 zur Entwicklung von Software nach SIL 2 entwickelt werden, um eine Zertifizierung zu ermöglichen.

[3] Comma Separated Values – ein einfaches Dateiformat, bei dem einzelne Dateneinträge mit einem Komma oder einem anderen Trennungszeichen separiert werden

4.5. Abgrenzungskriterien

Die folgende Funktionalität soll *nicht* realisiert werden:

1. **Kein Abflugmanagement auf Flughafenebene**
   Die Abflugplanungskomponente soll Position bzw. Gate des Luftfahrzeugs, Rollzeiten oder Startbahnbelegung bei der Bestimmung von Abflugzeiten *nicht* berücksichtigen. Die übermittelte früheste Abflugzeit soll bereits diese Details berücksichtigen – dies ist Aufgabe des Planers.

2. **Keine globale Optimierung**
   Die Abflugplanungskomponente soll keine Neuberechnung der Abflugzeiten und Flugflächen aller erfassten Flüge durchführen, sobald Änderungen in dem Berechnungsvektor stattfinden. Solch eine Änderung kann z.B. durch manuelle Übersteuerung durch das Flugsicherungspersonal oder durch Abweichungen der Flugvektoren eines Verkehrsfahrzeugs entstehen. Tritt eine Abweichung von den vergebenen Slot-Zeiten ein, sodass Potential für eine Optimierung bereits vergebener Slots entsteht, sollen alle vergebenen Slots unverändert bestehen bleiben.

3. **Keine automatische Slot-Zeitenkorrektur**
   Bedingung von 2. gilt mit folgender Änderung: durch die eintretende Abweichung von den vergebenen Slot-Zeiten entsteht Potential zur Verletzung einer Luftraumbeschränkung. Die vergebenen Slots sollen unverändert bestehen bleiben – die Gewährleistung der Einhaltung der Luftraumbeschränkung fällt in die Zuständigkeit der Fluglotsen und Luftfahrzeugführer.

4.6. Dokumentation

Alle ermittelten Anforderungen werden in der sog. System Requirements Specification festgehalten. Das SRS ist die Basis aller folgenden Entwicklungsschritte. Es dient später zur Testfallerzeugung, da hierbei alle Anforderungen mindestens einmal abgedeckt werden müssen.

Das SRS ist das Referenzdokument für die Abnahme vor Ort und muss deshalb auch vom Kunden abgenommen werden. Jede Änderung im System muss gegen die Anforderungen geprüft werden, gegebenenfalls müssen neue Anforderungen hinzugefügt werden oder alte angepasst, beides hat zur Folge, dass eine erneute Abnahme vom

Kunden fällig wird.

Im SRS wird für jede Anforderung eine Identifikationsnummer allokiert, diese wird in den weiteren Phasen des Projekts referenziert. Nach [Com05] bildet die Dokumentation die Basis der Nachvollziehbarkeit der Entscheidungsfindung und dadurch auch die Grundlage einer erfolgreichen Zertifizierung.

5. Entwurf

Die Entwurfsphase definiert die zu realisierenden Module, die alle gestellten Anforderungen erfüllen. Dabei werden u.a. verschiedene Lösungsvarianten verglichen und die Einbindung von Hilfswerkzeugen geplant.

Ab diesem Abschnitt werden die beiden Komponenten mit ihren Systemnamen referenziert:

- **ATCCL**
  Die *Air Traffic Control Constraint Language* ist die domänenspezifische Modellierungssprache der Luftraumbeschränkungen. Sie gilt als die Konfigurationssprache der DFLOW-Komponente. Der dazugehörige Compiler und die virtuelle Maschine werden oft auch mit dieser Abkürzung, oder aber explizit als das *ATCCL-Framework*, referenziert.

- **DFLOW**
  DFLOW bezeichnet die Abflugplanungskomponente und steht für *Departure Flow*.

5.1. ATCCL

Die Anforderungen spannten einen Raum von möglichen Lösungsansätzen auf, während die Sicherheits- und Benutzerakzeptanz die realisierbaren Möglichkeiten reduzierten. So wurde die Gestaltungsmöglichkeit der Syntax der Modellierungssprache durch die Benutzerzielgruppe stark begrenzt, die Semantik wurde zum großen Teil durch die Anwendungsdomäne bestimmt.

Bei strikten Vorgaben, also der Reduktion der Freiheiten, gestaltet sich die Entwicklung des Systementwurfs höchst effizient. Da der Grobentwurf zu einem großen Teil eine direkte Ableitung der Anforderung ist, hat sich die Entwurfsphase auf die einzelnen Module und deren Interaktion beschränkt.

5.1.1. Syntax

Bereits in der Anforderungsanalyse wurden Syntaxprototypen entwickelt, welche sowohl dem Kunden als auch den Projektmitarbeitern präsentiert wurden. Dieses Vorgehen ist hilfreich, um aus einer Reihe von Kandidaten, die mit vergleichbarer Komplexität die selben Ergebnisse liefern, die geeigneteren herauszufinden.

Zur Spezifizierung der ATCCL-Syntax wurde anhand der Anforderungen und den Prototypen ein vollständige *Erweiterte Backus-Naur-Form* – kurz EBNF, siehe [Wir77] – für die Syntax erstellt. Die EBNF wird zur Darstellung von kontextfreien Grammatiken genutzt, ist standardisiert und eignet sich sowohl als Spezifikation der Syntax einer Sprache als auch als Basis für die Erstellung von Parser oder Compiler mit Hilfe von Werkzeugen. Folgend befindet sich die komplette Spezifikation der ATCCL.

Notation

Die Syntax ist in EBNF spezifiziert. Klein geschriebene Produktionsnamen werden zur Tokenidentifizierung verwendet. Nicht-Terminale werden groß geschrieben. Direkte lexikalische Symbole werden in doppelten Anführungszeichen umschlossen.

Die Form a ... b beschreibt eine Menge von Zeichen, angefangen von a bis b.

```
Produktion      = produktionsname "=" Ausdruck "."
Ausdruck        = Alternative { "|" Alternative }
Alternative     = Term { Term }
Term            = produktionsname | token [ "..." token ]
                  | Gruppe | Option | Wiederholung
Gruppe          = "(" Ausdruck ")"
Option          = "[" Ausdruck "]"
Wiederholung    = "{" Ausdruck "}"

|   Alternation
()  Gruppierung
[]  Option (0 oder 1 mal)
{}  Wiederholung (0 bis n mal)
```

Listing 5.1: EBNF Notation

Buchstaben und Ziffern

Standardflugpläne unterstützen nur ASCII-Zeichen, deshalb gibt es keine Unicode-Unterstützung bei Zeichen. Die Sprache und der Compiler sind in der Lage, neben den natürlichen Zahlen auch reelle Zahlen zu verarbeiten. Jedoch gibt es keine reellen Merkmale in einem Flugplan.

```
letter            = /* Ein ASCII-Zeichen: A..Z und a..z */
digit             = "0" .. "9"
visible_character = /* Ein sichtbares ASCII-Zeichen */
```

Listing 5.2: ATCCL EBNF Buchstaben und Zeichen

Zusätzlich gibt es eine Produktionsregel für alle sichtbaren Zeichen, diese sind zulässige Eingaben innerhalb eines Kommentars oder einer Zeichenkettenkonstante.

Kommentare

ATCCL unterstützt nur einen Typ von Kommentaren, den Zeilenkommentar.

Ein Kommentar wird mit dem Zeichen # eingeführt, zum Kommentar gehört der nachfolgende Text bis zum Ende der Zeile. Will man ein Kommentar über mehrere Zeilen setzen, so muss jede Kommentarzeile mit # eingeleitet werden.

Kommentare werden vom Parser ignoriert und können alle unterstützen Zeichen enthalten.

```
com               = "#" { visible_character } EOL
```

Listing 5.3: ATCCL EBNF Kommentare

Terminatoren

ATCCL ist frei von Terminatoren. Es muss kein besonderes Zeichen – bei vielen Sprachen ist es ein Semikolon, bei Prolog ein Punkt und bei Python das Zeilenende – zur Terminierung einer Anweisung gesetzt werden. Die Sprache verzichtet dadurch auf ein weiteres syntaktisches Element zur Vereinfachung und Bereinigung der Syntax.

Aufgrund der einfachen und abgeschlossenen Struktur der Regeldefinitionen wird eine differenzierte Fehlerdiagnose basierend auf der Klammersetzung erreicht.

Bezeichner

Ein Bezeichner setzt den Identifizierungsnamen für eine Regeldefinition. Er kann sich aus beliebigen Buchstaben und Ziffern zusammensetzen und das Unterstrichsymbol beinhalten, solange er mit einem Buchstaben beginnt.

```
id                  = letter { letter | digit | "_" }
```

Listing 5.4: ATCCL EBNF Bezeichner

Schlüsselwörter

Schlüsselwörter werden in ATCCL zur Festlegung von Regeldefinitionstypen und für die Operatoren verwendet. Zusätzlich gibt es eine Reihe von Schlüsselwörter, welche die Einträge eines Flugplans bezeichnen.

Folgende Schlüsselwörter sind reserviert und dürfen nicht als Regeldefinitionsbezeichner genutzt werden.

```
/* Flight plan properties */
copx     adep     ades     rwy
route    atyp     equip    tas
frul     ftyp     travel_type
fl

/* Rule types */
time_sep          constraint
pattern           flowpoint

/* Operators */
is       in       not      less
greater  than     and      or
at       from     until    on

/* SI-Units */
s        min
```

Listing 5.5: ATCCL EBNF Schluesselwoerter

Die SI-Typen zur Zeitangabe, *s* für Sekunde und *min* für Minute, sind Schlüsselwörter in der ATCCL-Syntax, diese sind jedoch nicht auf globaler Ebene reserviert und werden nur innerhalb einer Constraint-Regeldefinition genutzt.

Datentypen und Konstanten

Zur erfolgreichen Modellierung der Flussdichtenbeschränkungen eines Fluginformationsgebiets müssen sowohl alle Einträge eines Flugplans als auch Zeitangaben von der Sprache unterstützt werden.

Die Flugplaneinträge sind in sechs verschiedenen Datentypen erfasst. Als Grundtypen gelten einzelne Zeichen, zusammengesetzte Zeichenketten (Wörter) und die natürlichen Zahlen. Für jeden Grundtyp gibt es die Möglichkeit mehrere Werte in einer Liste zusammenzufassen.

Zur Festlegung von Separationsregeln ist es notwendig Zeitangaben sowohl in Minuten und in seltenen Fällen sekundengenau zu definieren. Um Regeln bestimmten Uhrzeiten zuzuweisen, ist es notwendig, minutengenau die Tageszeit festzulegen.

ATCCL kennt folgende Datentypen:

- Zeichenkette (String)
- ganze Zahl (Integer)
- Zeichenkettenliste
- Liste von ganzen Zahlen

```
string          = """ { visible_character } """
int             = [ "-" ] digit { digit }
string_array    = "[" { string { "," string } } "]"
int_array       = "[" { int { "," int } } "]"
```

Listing 5.6: ATCCL EBNF Datentypen

Die Sprache kommt mit nur zwei Basisdatentypen aus und sie erlaubt zudem noch die Zusammenfassung mehrerer Konstanten in einer Liste. Eine Liste wird durch eckige Klammern gekennzeichnet, wobei die einzelnen Element mit Kommata separiert sind.

Zu bemerken sind der fehlende Datentyp zur Uhrzeitbestimmung und eine ungewöhnliche

Variante von natürlichen Zahlen, wobei eine Zahl mit einer Null beginnen darf. Beide Besonderheiten hängen zusammen: indem vorgeschaltete Nullen bei dem Integer-Datentyp zugelassen werden, kann man auf einen dedizierten Datentyp für die Uhrzeit verzichten und akzeptiert somit die direkte Eingabe im HHMM-Format. Dieses Format ist in der Flugsicherung geläufig und wird auch in den Flugplänen und anderen Flugsicherungssystem so erfasst.

Alle im Code verwendeten Konstanten müssen von einem der definierten Datentypen sein.

Flugplaneigenschaften

Alle für das System relevanten Einträge eines Flugplans – auch Flugplaneigenschaften genannt – sind in ATCCL als Schlüsselwörter reserviert und mit einem festen Datentyp versehen.

Um auf Eigenschaften zu operieren, werden diese entsprechend ihrem Datentyp in eine Produktion zusammengefasst.

```
int_pt          = "tas"
char_pt         = "ftyp"
string_pt       = "adep" | "ades" | "rwy" | "atyp" | "copx"
                  | "copn" | "frul" | "rfl" | "travel_type"
string_array_pt = "route"
char_array_pt   = "equip"
```

Listing 5.7: ATCCL EBNF Flugplaneigenschaften

Eine Besonderheit stellt das Property EQUIP dar. In der EBNF-Spezifizierung ist es als Zeichenliste definiert. Bei der weiterführenden Verarbeitung wird trotzdem auf den Datentyp Zeichenkette (String) zurückgegriffen, dadurch werden spezielle Operationen auf dieser Eigenschaft möglich.

Das Navigationsausrüstungsfeld (Navigational Equipment) eines Flugplans ist eine Aneinanderreihung von Buchstaben, wobei jeder Buchstabe für eine fest definierte Navigationsausrüstung steht. Es ist üblich, die einzelnen Buchstaben ohne Trennungszeichen aneinanderzureihen, was dem Basisdatentyp Zeichenkette entspricht. Gleichzeitig gilt diese besondere Zeichenkette als eine Liste von Zeichen, damit auch Teilmengenopera-

toren darauf angewendet werden können.

Operatoren

Nach der Definition der Datentypen und der Zuweisung der Flugplaneigenschaften zu dem jeweiligen Datentyp folgt die Spezifizierung der Operationen auf den Konstanten und Flugplaneigenschaften.

Alle Operationen werden mit Hilfe von Operatoren definiert. Eine Operation ist in der Regel ein Tupel von Operator, einer Flugplaneigenschaft und mindestens einer Konstanten.

```
int_op                    = "is" [ "not" ]
                            | ("greater" | "less") "than"
string_op                 = "is" [ "not" ]
string_array_op           = "is" [ "not" ]
                            | [ "not" ] "in"
string_string_array_op = [ "not" ] "in"
```

Listing 5.8: ATCCL EBNF Operatoren

Regeldefinitionen

Ein ATCCL-Programm besteht aus Regeldefinitionen. Es gibt drei Typen von Regeldefinitionen:

- Flugplanmuster
- Separationsregel
- Flow-Point

Ein Flugplanmuster definiert eine Regel, die bestimmte Flugplaneigenschaften prüft. Diese Regel setzt sich aus booleschen Termen zusammen und bedient sich dabei den Operatoren, Properties und Konstanten.

```
pattern        = "pattern" id "(" pattern_term ")"
pattern_term   = int_pt int_op int
```

```
        | (char_pt | string_pt) string_op string
        | string_array_pt string_array_op string_array
        | string_array string_array_op string_array_pt
        | string string_string_array_op
          (string_array_pt | char_array_pt )
        | pattern_term ("or" | "and") pattern_term
        | "not" pattern_term
        | "(" pattern_term ")"
```

Listing 5.9: ATCCL EBNF Flugplanmuster

Eine Separationsregel (Constraint) dient zur Festlegung der Separation an einem, oder mehreren Wegpunkten.

Zum erfolgreichen Durchsetzen einer Flussdichtenregelung reicht eine zeitliche Staffelung, denkbar sind jedoch auch Abstandsseparationen und das Setzen von durchschnittlichen Durchflussdichten verteilt auf eine Zeitspanne. Bei Fertigstellung dieser Arbeit wurde nur die zeitliche Staffelung unterstützt.

Die Separation kann in Minuten und Sekunden festgelegt werden, kann einen oder eine Liste von Wegpunkten betreffen und kann zeitabhängig sein.

```
constraint        = "constraint" id "(" constraint_term ")"

constraint_term   = "time_sep" "at" (string | string_array)
                      "is" int si_unit
                      [ "from" int "until" int ]
                      [ "at" "fl" int_array ]
                    | constraint_term "and" constraint_term
si_unit           = "s" | "min"
```

Listing 5.10: ATCCL EBNF Separationsregeln

Der Flow-Point definiert eine Zuweisung von Flugplanmuster auf eine Separationsregel. Ein Flow-Point ist somit nicht an einen geographischen Wegpunkt gebunden. Diese Abbildung wird von der Abflugplanungskomponente zur Zuweisung von Flow-Points zu Flugplänen genutzt, worauf die Kalkulation des optimalen Abflugzeitpunkts basiert.

```
flow_point      = "flowpoint" id "(" id ":" id ")"
```

Listing 5.11: ATCCL EBNF Flow Point

Zur Verdeutlichung: die erste id dient zur Vergabe des Flow-Point-Namen, die zweite zur Identifizierung des Pattern und die dritte zur Zuweisung des Constraint. Dabei spielt die Lokalität der Pattern- und Constraint-Definition innerhalb der Konfigurationsdatei keine Rolle, diese müssen also nicht der Flow-Point-Definition voranstellen.

Programm

Ein ATCCL-Programm ist eine Konfiguration, welche sich aus einer Reihe von Regeldefinitionen zusammensetzt.

```
config          = { rule | comment }
rule            = pattern | constraint | flow_point
```

Listing 5.12: ATCCL EBNF Konfiguration

5.1.2. Beispiele

Anhand von repräsentativen Beispielen werden nun mögliche Einsätze der ATCCL vorgeführt.

Bsp. 1: Zielstellung

Es soll eine zeitliche Mindestseparationsbeschränkung von 5 Minuten an Wegpunkt *COPPI* gesetzt werden. Die Beschränkung soll für alle Flüge gelten, die vom *Dubai International Airport* starten und den Frankfurter Flughafen als Ziel haben. Der ICAO-Code für den Flughafen Frankfurt-Main ist *EDDF* und für den Flughafen Dubai *OMDB*.

Bsp. 1: Code

```
# flight plan pattern used for the filtering of flight plans
pattern DBCoppiDF(adep is "OMDB" and ades is "EDDF"
                  and "COPPI" in route)

# constraint definition for the way point
constraint Coppi5(time_sep at "COPPI" is 5 min)

# flow point definition by the mapping of a pattern to a constraint
flowpoint DBCoppiDF(DBCoppiDF: Coppi5)
```

Listing 5.13: ATCCL Einfaches Beispiel

Bsp. 1: Bemerkungen

Der obige Code definiert einen Flow Point mit dem Namen *DBCoppiDF*. Die Beschränkung auf diesem Flow Point, ist eine zeitliche Separation auf dem Wegpunkt *COPPI*, dies ist in dem Constraint *Coppi5* definiert. Die Bestimmung der Flugpläne, die dem Flow Point *DBCoppiDF* zugeordnet werden, wird durch das Pattern *DBCoppiDF* realisiert. Wie man sieht, ist die Namensauflösung nicht global, sondern basiert auf den drei Regeldefinitionstypen. So ist die mehrfache Vergabe eines Namens für mehrere Pattern-Definitionen nicht zulässig. Die Definition eines Patterns und eines Constraints mit gleichem Namen ist dagegen erlaubt.

Bsp. 2: Zielstellung

Ein weiteres Beispiel soll eine komplexe Regel realisieren. Es sollen folgende zeitliche Mindestseparationsbeschränkungen in Abhängigkeit der Uhrzeit gewährleistet werden:

1. 0:00 bis 6:00 Uhr: 5 Minuten, freigeschaltete Flugflächen sind 300, 320, 340
2. 6:01 bis 12:00 Uhr: 3 Minuten, freigeschaltete Flugflächen sind 300, 320
3. 12:01 bis 22:00 Uhr: 130 Sekunden, keine Höhenstaffelung

Wie im ersten Beispiel gilt der Flow Point für alle Flüge, die von Dubai nach Frankfurt unterwegs sind. Die Separation soll dieses mal jedoch an dem Wegpunkt *COPPI* oder *BALUS* durchgeführt werden, abhängig davon welcher der beiden Wegpunkte in der jeweiligen Flugroute liegt.

Bsp. 2: Code

```
pattern DBCoppiBalusDF(adep is "OMDB" and ades is "EDDF"
                       and ("COPPI" in route or "BALUS" in route))

constraint CoppiBalus(time_sep at ["COPPI", "BALUS"] is 5 min
                        from 0000 until 0600 at FL [300, 320, 340]
                      and time_sep at ["COPPI", "BALUS"] is 3 min
                        from 0601 until 1200 at FL [300, 320]
                      and time_sep at ["COPPI", "BALUS"] is 130s
                        from 1201 until 2200)

flowpoint Example2(DBCoppiBalusDF: CoppiBalus)
```

Listing 5.14: ATCCL Komplexes Beispiel

Bsp. 2: Bemerkungen

Der Flow Point *Example2* realisiert alle Anforderungen aus der Zielstellung. Es ist ersichtlich, dass für den Zeitraum von 22:01 Uhr bis 23:59 keine Separation definiert wird, in dieser Zeit gilt keine Beschränkung auf dem Flow Point. Liegen beide Wegpunkte in der Route, so wird lediglich auf einem Wegpunkt gestaffelt. Die Sprache macht keine Aussagen darüber, welcher Wegpunkt in so einem Fall gewählt wird.

Diese Beispiele sollen keine realistischen Gegebenheiten realisieren und dienen somit ausschließlich der Beschreibung der möglichen Anwendungen der Sprache. Im letzten Beispiel tritt durch die Wegpunkte eine Uneindeutigkeit für die Wegpunktwahl zur Staffelung auf, da die Wegpunkte *BALUS* und *COPPI* sich oft in einer Route befinden. Die Konfigurationen der realen Luftraumbeschränkungen sind im Gegensatz dazu eindeutig gewählt.

5.1.3. Compiler

Der Übersetzer soll eine ATCCL-Konfiguration in C++-Strukturen überführen. Die entstandene Objektdatenbank wird die Eingabe für die virtuelle Maschine darstellen, auf der alle Laufzeitoperationen stattfinden.

Vor der Überführung in den Bytecode wird der Syntaxbaum erstellt, dabei gilt es die Konfiguration auf Fehler zu überprüfen. Der Compiler besteht somit aus einem Analysemodul und einem Synthesemodul. Die Analysephase beinhaltet die lexikalische Analyse gefolgt von der Syntaxanalyse.

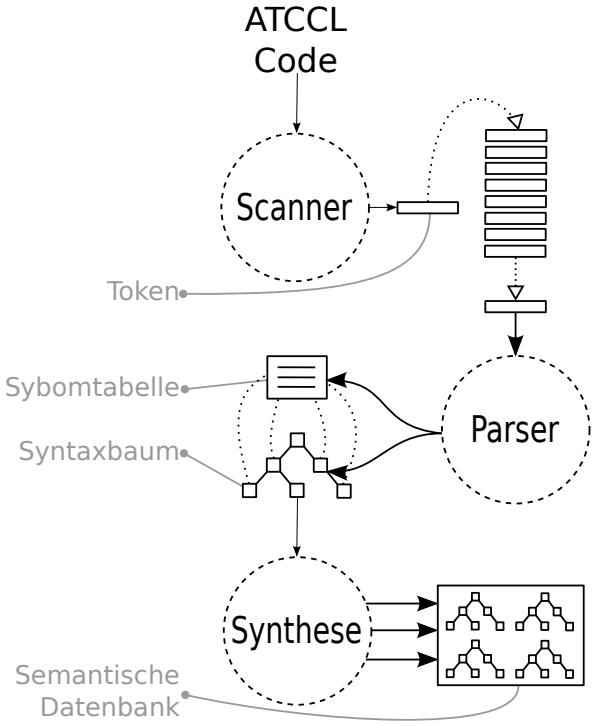


Abbildung 5.1.: Der Übersetzungsprozess

Scanner

Das Modul der lexikalischen Analyse – auch Scanner genannt – erstellt aus dem ATCCL-Code eine Token-Codierung. Der Quellcode wird hierbei auf definierte Token untersucht, die Token dienen als Eingabe für die Syntaxanalyse.

`GNU flex` bietet eine effektive Möglichkeit lexikalische Scanner zu generieren. Die Basis des Scanners ist eine `flex`-spezifische Konfiguration zur Definition der Token. Die Konfiguration verwendet eine Syntax ähnlich den regulären Ausdrücken und ist in der Lage Produktionen für Untermengen von Zeichenketten zu erstellen, die bestimmte Entitäten einer Sprache darstellen. Anhand der Konfiguration erstellt `flex` C-Code für einen Scanner, welcher durch das Bereitstellen eines Interface mit einen `yacc`- oder `bison`-generierten Parser gekoppelt werden kann.

Parser

Getrieben durch die Ausgabe des Scanners wird die Syntaxanalyse durchgeführt. Diese Aufgabe übernimmt im ATCCL-Framework der Parser.

Der Parser wird mit Hilfe von `bison - GNU Parser Generator` erstellt. Der Parser benötigt ähnlich wie der Scanner eine spezifische Konfiguration, diesmal analog zu der BNF/EBNF-Notation. `bison` generiert ebenfalls C-Code, basierend auf der `bison`-Konfigurationsdatei und der Header-Datei des `flex`-generierten Scanners.

Das Resultat ist ein Syntaxbaum und eine Symboltabelle der ATCCL-Konfiguration. Der Baum hält die Struktur des Codes fest, während die Symboltabelle Tupel mit Regelsymbol, Regeltyp und Regelreferenz hält. Die semantische Prüfung erlaubt die Analyse von Zusammenhängen zwischen den Elementen eines Programms. Die Prüfung meldet u.a. die Verwendung unerlaubter Kombinationen von Datentyp und Operator und falsche Typenwahl bei Konstanten.

Synthese

Nach Abschluss der Analysephase erfolgt die Synthese. Anhand der Symboltabelle werden alle Symbole in dem Syntaxbaum aufgelöst. Das Resultat sind vollständige Regeldefinitionen, die im nächsten Schritt in C++-Objekte transformiert werden.

Die Basis der Objektinstanziierung sollen *Factory*-Klassen nach [GHJV95] bilden. Die `TermFactory` wird die Instanzen der Regelterme realisieren. Durch ein Klassen-Template soll der Datentyp abstrahiert und dadurch eine generische Lösung für alle Fälle bereitgestellt werden. Gleiches gilt für die `OperatorFactory`. Diese soll je nach Datentyp einen passenden Operator instantiieren.

Durch die semantisch korrekte Verknüpfung der einzelnen Terme entstehen die Regeldefinitionen. Die Regelobjekte werden in einer semantischen Datenbank zusammengefasst und beinhalten zu diesem Zeitpunkt alle Konstanten und die Entscheidungsbäume zur Regelauswertung. Die semantische Datenbank – auch `SemanticTree` genannt – dient als Eingabe und Basis aller Transformationen für die virtuelle Maschine. Sie beinhaltet:

- `Pattern`-Regelinstanzen
- `Constraint`-Regelinstanzen
- `Flowpoint`-Mappings

Die `SemanticTree`-Datenstruktur bietet zusätzlich verschiedene Zugriffsmöglichkeit auf die Regeln und Mappings, u.a. nach der, in der Konfiguration, vergebenen id, also den Namen bzw. Symbol der Regeldefinition.

5.1.4. Virtuelle Maschine

Die virtuelle Maschine bietet eine API[1] mit folgenden Merkmalen:

- **Evaluierung von Flugplan nach Flugplanmuster**
  Eingangsparameter: *Pattern*, *FlightPlan* (siehe Abschnitt 5.1.4)
  Ausgangswert: *Boolean*, *true* falls das Muster für den Flugplan gültig ist, sonst *false*
- **Zuordnung von Flow-Points zu Flugplan**
  Eingangsparameter: *FlightPlan*
  Ausgangswert: *FlowpointCollection*, eine Liste von *Flowpoint*-Referenzen, denen der Flugplan zugeordnet wird
- **Durchsetzung einer Separationsregel zwischen zwei Flugplänen**
  Eingangsparameter: *FlightPlan 1*, *FlightPlan 2*, *Constraint*

[1] Application Programmable Interface – eine Programmierschnittstelle zur Bereitstellung von Softwarefunktionen und zur Integration mit anderen Softwaremodulen

Ausgangswert: *FlightPlan*, dessen Abflugzeit entsprechend des *Constraint* angepasst wurde, um keine Luftbeschränkungen zu verletzen

Alle Operationen basieren auf den übergebenen Flugplänen und der semantischen Datenbank, die vom Compiler geliefert wird (Abb. 5.2). Eine detaillierte Beschreibung zu der Evaluation von Flugplanmuster und der Bestimmung von optimalen Abflugzeiten befindet sich in den Abschnitten 5.1.6 und 5.1.7.

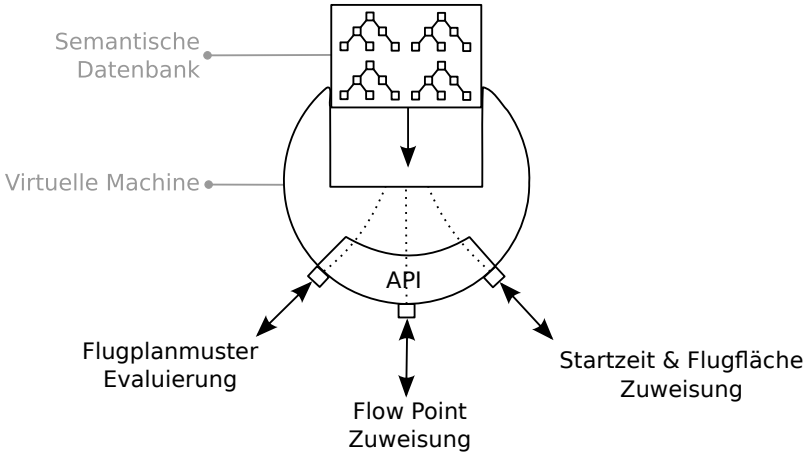


Abbildung 5.2.: ATCCL `VirtualMachine`

FlightPlan-Interface

Die ATCCL-API steht in Abhängigkeit zum `FlightPlan`-Interface. Die Entscheidung über die Schnittstelle zum Flugplan war essentiell und deshalb nicht einfach. Dabei galt es möglichst hohe Sicherheit bei Laufzeit zu garantieren, ohne die Nutzung und das Testen der API unnötig kompliziert zu machen.

Zur Auswahl standen ein `StorageBinder`, der dynamisch zur Laufzeit Objekte vom Typ `Storage` binden und den internen Prozeduren in der `VirtualMachine` bereitstellen kann. Der Vorteil dieses Systems wäre der niedrige Wartungsaufwand bei Änderungen in den Flugplaneigenschaften. Bestandteile eines Flugplans, die für ein Projekt nicht relevant sind, könnten ignoriert werden. Der Nachteil ist die hohe Fehleranfälligkeit seitens des Benutzers. Sobald ein Benutzer nicht alle notwendigen Strukturen in den `StorageBinder` übergibt, fehlen der virtuellen Maschine möglicherweise `Properties`, die es zur Verarbeitung benötigt. Die Fehlerbehandlung muss bei Laufzeit geschehen

und erhöht dadurch die Fehleranfälligkeit des Systems.

Zur Vorbeugung vor Bedienungsfehler wird ein monolithisches Interface verwendet, das *Zugriffsmethoden* zu allen Flugplaneigenschaften deklariert, siehe Anhang A.1.5. Der Vorteil dieses Designs bietet eine vollständige Kontrolle über die Implementierung bei *Übersetzungszeit*. Der Nachteil liegt in der Test- und Wartbarkeit der Klasse. Zum Testen muss hierfür auf eine *Dummy*[2]-Implementierung oder ein *Mock-Object*[3] zurückgegriffen werden. Sobald eine Änderung des Interface erfolgt, muss jeder Nutzer dieser Bibliothek Anpassungen vornehmen.

Um dem Letzteren entgegenzuwirken, wurde das Interface außerhalb des ATCCL-Frameworks, in der sog. `stdbase`-Bibliothek abgelegt. Zusätzlich wurde eine `DMAP`-spezifische Implementierung in der `DMAP`-Bibliothek erstellt. Dies bedeutet, dass ein Benutzer des ATCCL-Frameworks nicht selbständig das Interface zu realisieren hat, sondern auf die `DMAP`-spezifische Version zurückgreifen kann. Gleichzeitig bleibt das ATCCL-Framework unabhängig von externen Bibliotheken. Eine Ausnahme bildet die Abhängigkeit von `stdbase`, wobei diese bereits bestand und wünschenswert ist.

Flugsicherungssysteme und insbesondere deren Standards erfahren eine sehr langsame Evolution. So wurde die Spezifikation des aktuellen Flugplanstandards bereits vor mehrere Dekaden verabschieden. In den kommenden Jahren wird jedoch ein neuer Standard eingeführt, somit sind die Überlegungen zur effektiven Anpassung der Schnittstelle berechtigt.

5.1.5. Compilerprototyp

Als Abschluss der Entwurfsphase des ATCCL-Frameworks und Einleitung der Implementierung wurde ein Prototyp für den Übersetzer entwickelt. Anhand eines Beispiels sollte die Realisierbarkeit des Entwurfs bestätigt werden, besonderes Augenmerk galt dabei der erfolgreichen Integration der externen Werkzeuge wie `flex` und `bison` in den Compiler-Baustein.

Die drei Phasen des Compilers wurden inkrementell entwickelt und auf Funktionalität und Fehleranfälligkeit getestet. Bei diesem Vorgehen wurde eine alternative Compiler-

[2]Ein Objekt, das ein Interface mit statischen Rückgabewerten und Verhalten implementiert
[3]Ein Objekt, das ein Interface mit zur Laufzeit bestimmten Werten und Verhalten implementiert

Architektur verwendet, als in der finalen Version des Prototyps. Zur Steigerung der Flexibilität der Tests wurde ein Interpreter entwickelt, welcher eine dynamischere Eingabe erlaubte als es mit statischen Konfigurationsdateien möglich gewesen wäre.

Mit Hilfe des Interpreters konnte gut mit den Werkzeugen experimentiert werden. Es war ohne großen Aufwand möglich, die Syntax zu erweitern oder zu verändern. Durch das unmittelbare Feedback wurden so die letzten Zweifel beseitigt.

Der Verzicht auf unnötige Syntaxelemente, wie z.B. den Terminatoren, erhöhte die Komplexität der Fehlerfindung und Fehlerbehandlung. Sollen in einem Übersetzungsdurchgang möglichst viele – im Idealfall alle – Fehler einer ATCCL-Konfiguration diagnostiziert werden, so muss eine differenzierte Fehlerbehandlung implementiert werden. Der Compiler muss in der Lage sein, eine Fehlerquelle genau zu lokalisieren, den Fehlerraum von der Bearbeitung ausschließen und die Zustandsmaschine in den letzten korrekten Zustand versetzen.

Terminatoren sind bei vielen Sprachen gut dafür geeignet solche Fehlerräume abzugrenzen, dadurch wird durch einen Fehler maximal eine Anweisung invalidiert. Durch das Entfernen der fehlerhaften Anweisung kann das restliche Programm übersetzt werden, mögliche Folgefehler sind meist trivial zu lösen. Im Falle der ATCCL-Syntax boten sich die runden Klammern zur Abgrenzung der Fehlerräume an. Alle ATCCL-Regeln enden mit einer solchen Klammer, somit ist im trivialen Fall ein Fehler dadurch auf eine Regel isoliert.

Die Nachteile dieser Fehlerbehandlungsmethode sind:

- bei einfachen Regeln
  Es wird maximal ein Fehler pro einfacher Regel berichtet.

- bei komplexen Regeln
  Ein Fehler kann propagieren, es werden Folgefehler für die Regel berichtet.

Die Unterscheidung zwischen einfachen und komplexen Regeln besteht in dem Vorhandensein von Klammern innerhalb einer Regeldefinition, die dazu genutzt werden um Prioritäten der Auswertung von booleschen Termen festzulegen. Einfache Regeln haben keine Klammersetzung innerhalb der Regeldefinition.

5.1.6. Evaluation von Flugplanmustern

Die virtuelle Maschine des ATCCL-Frameworks soll in der Lage sein, die Gültigkeit eines Musters anhand von Flugplandaten zu bestimmen. Die virtuelle Maschine initiiert die Evaluation des Musters durch die Weiterleitung des Flugplans an die `evaluate`-Methode der entsprechenden `Pattern`-Referenz.

Beispiel

Es sei folgendes Flugplanmuster definiert:

```
pattern Example1(adep is "OMDB" and
                 (tas greater than 200 or "BALUS" in route))
```

Listing 5.15: ATCCL Pattern-Beispiel

Das Muster definiert die Menge aller Flüge, die in Dubai starten (*Aerodrome of Departure* ist *OMDB*) und entweder eine höhere Geschwindigkeit als 200 kn erreichen (*True Airspeed* ist größer als 200) oder der Wegpunkt *BALUS* ist in der Flugplanroute enthalten.

Die Synthese übersetzt diese Konfiguration in eine Baumstruktur mit zwei Booleschen Termen repräsentativ für die beiden Booleschen Verknüpfungen und sechs Blattknoten, jeweils paarweise eine Konstante und eine Flugplaneigenschaft. Die Operatoren sind für die Aktionen der Terms zuständig, in diesem Beispiel realisieren diese eine Und-Verknüpfung, eine Oder-Verknüpfung, einen Äquivalenztest, einen numerischen Vergleich und eine Teilmengenrelation. Der genaue Schrittabfolge der Auswertung eines solchen Flugplanmusters wird anhand eines Beispiels erläutert.

Die folgenden Schritte beschreiben die Stationen des Algorithmus, Abb. 5.3 visualisiert den Pattern-Baum und die Schrittabfolge:

0. Per `evaluate`-Methode wird ein Flugplan an den Wurzelknoten übergeben. Die Flugplaneigenschaften seien wie folgt: ADES: OMDB, TAS: 280, ROUTE: DESDI TARDI RASKI.
1. Der `evaluate`-Aufruf propagiert bis zum ersten Blattknoten.
2. Das *True Airspeed*-Attribut wird aus dem Flugplan extrahiert und an den Operator zurückgegeben. *Wert: 280.*

3. Der Wert der Konstante wird an den Operator zurückgegeben. *Wert: 200.*
4. Der Operator führt den Vergleich *280 >200* aus. Die `evaluate`-Methode terminiert mit dem *Rückgabewert: true.*
5. Der Wert der `evaluation`-Rückgabe wird an den Oder-Operator zurückgegeben.
6. Der Oder-Operator wertet den Ausdruck *true* $\vee$ *?* aus. Die Auswertung des rechten Terms ist dadurch unnötig. *Rückgabewert ist true.*
7. Der Rückgabewert aus 6 propagiert zurück zur Wurzel.
8. Zum Auswerten des Wurzelausdrucks *true* $\wedge$ *?* ist die Auswertung des rechten Terms nötig.
9. Das *Aerodrome of Departure*-Attribut wird aus dem Flugplan extrahiert und zurückgegeben. *Wert: OMDB.*
10. Der Wert der Konstante wird zurückgegeben. *Wert: OMDB*
11. Der Operator führt den Äquivalenzvergleich *OMDB* = *OMDB* aus mit dem *Ergebnis true.*
12. Das Ergebnis des Äquivalenzvergleichs wird zurück propagiert.
13. Der Operator evaluiert den Booleschen Ausdruck *true* $\wedge$ *true* mit dem *Ergebnis true.* Die Auswertung terminiert, der `evaluate`-Methodenaufruf gibt *true* zurück.

5.1.7. Optimierung der Abflugzeit

In den folgenden Gleichungen werden Vergleichsrelationen auf Uhrzeiten angewandt, hierbei gilt stets: datiert ein Zeitpunkt $t1$ einen *späteren* Zeitpunkt, unabhängig von der Tageszeit, als ein Zeitpunkt $t2$, so gilt:

$$t1 > t2$$

Anders formuliert liegt $t1$ zum Zeitpunkt $t2$ in der Zukunft.

Die Optimierung der Abflugzeit bedeutet eine konfliktfreie Slot-Zuordnung an allen betreffenden Flow Points.

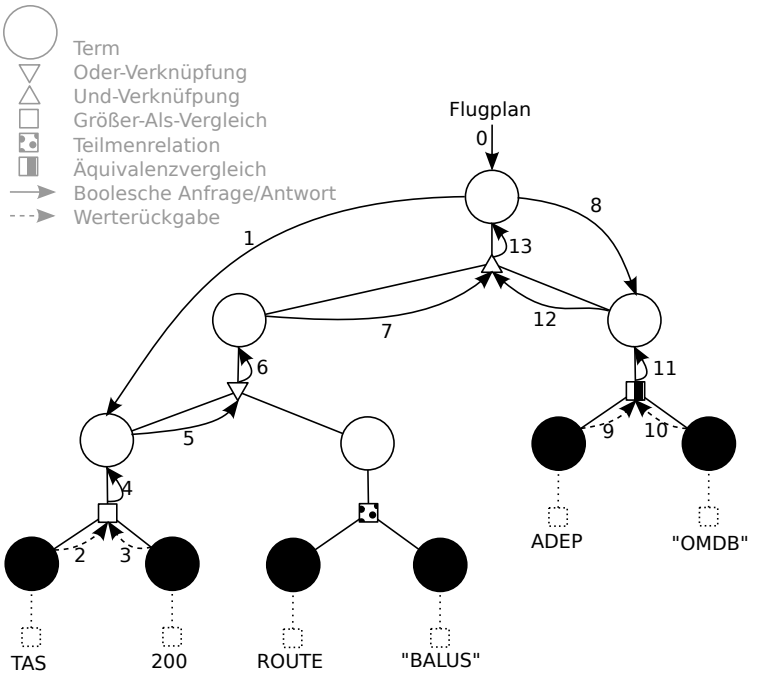


Abbildung 5.3.: ATCCL `Pattern`-Evaluation-Beispiel

Sei f ein Flow Point und M die Menge aller Slot-Zeiten von f. Eine Slot-Zeit t ist konfliktfrei für Separationszeit s, wenn gilt:

$$\forall m \in M : |t - m| \geq s$$

Wird die Optimierung angestoßen, findet die virtuelle Maschine die früheste Abflugzeit, sodass im nachfolgendem Routenverlauf kein Separationsregelverstoß auftritt.

Sei K die Menge aller Abflugzeitenkandidaten. Eine Abflugzeit d ist optimal für die früheste Abflugzeit e, wenn gilt:

$$d = \min\{k \in K | k \geq e\}$$

Dabei ist ein Abflugzeitenkandidat eine Abflugzeit, durch die nur konfliktfreie Slot-Zeiten auf den Wegpunkten der Route folgen.

Zur Realisierung dieser Optimierung wurden verschiedene Entwürfe in Betracht gezogen, wobei eine kundenspezifische Beschränkung von einem Flow Point pro Flugplan gesetzt wurde. Der Entwurf sollte eine möglichst einfache und dadurch robuste Implementierung ermöglichen. Der Algorithmus wurde hierfür in möglichst kleinen Teilschritten mit niedriger Komplexität modelliert.

Algorithmus

Zur Bestimmung der optimalen Abflugzeiten in Abhängigkeit der geltenden Luftraumbeschränkungen und der frühest möglichen Abflugzeit, wurde ein Algorithmus entwickelt.

Vor der Initiierung der Berechnung wird der DFLOW-Komponente die früheste Abflugzeit übergeben. Der Flugplan des betreffenden Fluges und alle weiteren Flugpläne, die den gleichen Flow Point teilen, werden bereitgestellt. Die Route des Flugplans muss vor der Kalkulation durch die SID-Wegpunkt vervollständigt worden sein.

Basierend auf der gesetzten frühesten Abflugzeit werden mit Hilfe von `CalcEstimates` die Überflugzeiten bestimmt, insbesondere auch die Überflugzeit des Wegpunkts an dem die Separation stattfinden soll, hier mit $flowtime$ referenziert. $flowtimes$ ist die Menge aller Wegpunktzeiten der anderen Flüge, die den Flow Point teilen. *separation* hält den definierten Mindestseparationsabstand. Alle Zeiten sind fortlaufende natürliche Zahlen, wie z.B. durch die *POSIX Time*[4]-Konvention gegeben.

Algorithmus 1 beschreibt die Bestimmung der konfliktfreien Flow Point-Zeit.

Algorithmus 1 $resolve(separation, flowtime, flowtimes) \rightarrow flowtime$

```
t_s ← separation
t_a ← flowtime
t_b ← invalid
while t_a ≠ t_b do
  t_b ← t_a
  for all t_f ∈ flowtimes do
    t_d ← (t_a − t_f)
    if |t_d| < t_s then
      t_a ← (t_a + t_s − t_d)
    end if
  end for
end while
return t_a
```

Nach der Terminierung wird die erhaltene Flow Point-Zeit zurück propagiert, um dar-

[4]Zeitangabe basierend auf der Anzahl an vergangenen Sekunden seit 1. Januar 1970 0:00 Uhr UTC – UTC steht für Universal Time Coordinated und stellt die heute gültige Weltzeit dar

aus die optimale Abflugzeit zu erhalten. Der Algorithmus ist auf die eindeutige Zuweisung von maximal einem Flow Point pro Flugplan beschränkt. Sollen mehrere Flow Points pro Flugplan erlaubt sein, so gilt es den gleichen Algorithmus iterativ auf die Menge aller gemeinsamen Flow Points anzuwenden.

Algorithmus 2 ist ein Vorschlag, wie der Algorithmus 1 erweitert werden kann, um mehrere Flow Points pro Flugplan zu erlauben. *atot* steht für die optimierte Abflugzeit, $flightplan$ ist ein Objekt mit den Eigenschaften früheste Abflugzeit (*etot*) und den zugeordneten Flow Points ($flowpoints$). Das Flow Point-Objekt hält Informationen zu der Flow Point-Überflugzeit ($flowtime$) und der definierten Separation (*separation*). Die Beschreibung soll nur die Idee des tatsächlichen Algorithmus erfassen, zur Vereinfachung wurde u.a. auf die Berücksichtigung der Tageszeit und die Höhenstaffelung verzichtet.

Algorithmus 2 $multresolve(flightplan, flightplans) \rightarrow atot$

$atot_a \leftarrow flightplan.etot$
$atot_b \leftarrow invalid$
while $atot_a \neq atot_b$ **do**
  $atot_b \leftarrow atot_a$
  for all $flowpoint \in flightplan.flowpoints$ **do**
    $atot_d \leftarrow (flowpoint.flowtime - atot_b)$
    $s \leftarrow flowpoint.separation$
    $f \leftarrow flowpoint.flowtime$
    $fs \leftarrow flightplans.flowtimes(flowpoint)$
    $flowpoint.flowtime \leftarrow resolve(s, f, fs)$
    $atot_b \leftarrow flowpoint.flowtime - atot_d$
  end for
end while
return $atot_a$

Komplexität

Die Entscheidung fiel auf ein iteratives Vorgehen, wobei pro Iteration alle beteiligten Flugpläne sequenziell zur Konfliktauflösung in Betracht gezogen werden. Eine Iteration besteht aus n Schritten, wobei n gleich der Anzahl der Flugpläne ist, welche die selben Flow Points belegen, die auch für den zu optimierenden Flugplan gelten. Schrittweise

werden auftretende Konflikte zwischen zwei Flugplänen aufgelöst, solange die ermittelte optimale Abflugzeit am Anfang der Iteration ungleich der optimalen Abflugzeit am Ende einer Iteration ist.

Ist dies der Fall – also sind die optimalen Abflugzeiten am Anfang und Ende einer Iteration identisch, so terminiert der Algorithmus und man erhält die optimale Abflugzeit. Die obere Schranke der Laufzeitkomplexität für diesen Ansatz liegt bei $\mathcal{O}(n^2)$, wobei n die Anzahl der Flugpläne ist mit mindestens einem gemeinsamen Flow Point. Bei Konfliktfreiheit oder anderen günstigen Konfliktbedingungen gleicht das asymptotische Verhalten $\Omega(n)$, da nach einer Iteration bereits die Abbruchbedingung erfüllt ist.

Alternative

Eine Alternative beinhaltet die Vorsortierung der Flugpläne nach Slot-Zeiten. Die Sortierung bereitet den Weg für eine Optimierung vor, bei der mit jedem Schritt der Suchraum halbiert werden kann. Man reduziert die betrachteten Flugpläne auf diejenigen, deren Slot-Zeiten in Konflikt oder in der Zukunft zur eigenen liegen. Das asymptotische Verhalten dieses Algorithmus liegt bei $\mathcal{O}(n^2)$ bzw. $\Omega(n \log n)$ für das Sortieren unter Einsatz von *Quicksort* und insgesamt bei $\mathcal{O}(n^2)$, da im ungünstigstem Fall n Schritte notwendig sind um nach der Sortierung die optimale Zeit zu bestimmen. Im Durchschnitt läge die Komplexität bei $\Theta(n \log n)$.

Der alternative Ansatz bedeutet eine höhere Komplexität für die Implementierung, da zusätzlich das Sortieren zu realisiert ist. Standardalgorithmen wie das `std::sort` der Standardbibliothek von C++ kommen aufgrund der dynamischen Speicherallokierung nicht in Frage. Außerdem wird für den operativen Betrieb mit einem hohem Aufkommen von günstigen Bedingungen für den iterativen Algorithmus gerechnet, wodurch die Vorsortierung der Flugpläne nur eine Verschlechterung der Leistung bedeutet.

Höhenstaffelung

Eine zusätzliche Komplexität bei der Bestimmung der optimalen Abflugzeit bestand in der Separation nach Flugflächen, siehe 3.1.2. ATCCL soll, wenn definiert, auf allen freigegebenen Flugflächen eines Flow Points separieren. Dieser Art der Staffelung soll eine höhere Priorität besitzen als die Längsstaffelung, siehe 3.1.2.

Zur Realisierung dieses Verhaltens wird jede freigegebene Flugfläche für einen Flow

Point getrennt behandelt. Besteht auf einer Flugfläche kein Konflikt, ausgehend von der frühesten Abflugzeit, so terminiert der Vorgang frühzeitig, weil das automatisch die optimale Abflugzeit darstellt. Findet kein frühzeitiger Abbruch statt, so wird nach der Ermittlung aller optimalen Zeiten die früheste Zeit mit der entsprechenden Flugfläche für den Flug reserviert.

Mit der Höhenstaffelung beträgt die Laufzeitkomplexität der Optimierung $\mathcal{O}(m*n^2) \rightarrow \mathcal{O}(n^3)$, wobei n für die Anzahl an Flugplänen und m für die Anzahl an freigegebenen Flugflächen steht. Jedoch ist die Anzahl an möglichen Flugflächen sehr begrenzt. Für einen Flow Point – der stets nur eine Flugrichtung realisiert – kann z.B. FL 200 bis FL 600 in Abständen von FL 20 freigegeben sein, was ein $m = 20$ ergibt und als Maximalwert gesehen werden kann.

5.2. DFLOW

Die DFLOW-Komponente soll als Teil des FDPS in die PRISMA-Architektur integriert werden. Die Komponente hat die Aufgabe, auf Anfragen zu reagieren, die Flugpläne mit Hilfe der virtuellen Maschine des ATCCL-Frameworks zu verarbeiten und die transformierten Flugpläne wieder in das System einzuspeisen. Die DFLOW-Komponente reagiert auf folgende Ereignisse:

- **Benutzeranfrage zur optimalen Abflugzeiten- und Flugflächenbestimmung**
  Über das DFLOW-Display wird eine Anfrage zur Bestimmung der Abflugzeit erzeugt. Die Komponente registriert die Anfrage, bearbeitet sie und gibt die optimale Abflugzeit und Flugfläche in das System.
- **Neuer Überflug registriert**
  Wird ein noch nicht registrierter Überflug erkannt, wird dieser bearbeitet, um ihn bei folgenden Anfragen bei eventuell auftretenden Konflikten zu berücksichtigen.

Zur Realisierung dieser Logik und der Integration in PRISMA wurden u.a. PRISMA-Standardklassen verwendet. PRISMA folgt einem datengetriebenen Modell, die interne Datenübertragung findet über die DMAP statt.

5.2.1. DMAP-Interaktion

PRISMA definiert Schnittstellen zur DMAP mit Hilfe von einer Reihe von Klassen. Es gibt u.a. eine Klasse, die den Lese- und Schreibzugriff auf die einzelnen Datensätze

der DMAP bereitstellt. Eine weitere Klasse bietet Möglichkeiten an, auf Transaktionen bestimmter Datensätze zu reagieren.

Map

Für jeden DMAP-Datensatz existiert als Schnittstelle zu den Transaktionen mit diesem Datensatz ein zugehöriges MAP-Interface. Mit Hilfe der MAP-Klasse lassen sich sog. Notify-Objekte als *Observer*, siehe [GHJV95], registrieren. Weiter bietet die Klasse Zugriffsmöglichkeiten auf die Dateneinträge an.

Notify

Ein Notify-Objekt realisiert die Reaktionen auf Datensatztransaktionen. Mögliche Transaktionen sind:

- Einfügen eines neuen Eintrags
- Ändern eines vorhandenen Eintrags
- Löschen eines vorhandenen Eintrags

Die Basis der Logik der datenverarbeitenden Module des PRISMA und insbesondere des FDPS basiert auf Notify-Objekten. Gänzlich entkoppelt von der Verarbeitungsoperationen der parallel laufenden Prozessen, wird mit dieser Architektur eine modulare und stabile Grundlagen für ein sicherheitskritischen System bereitgestellt.

Das DFLOW braucht zur Realisierung der Logik Zugriff auf folgende DMAP-Datensätze:

- **DFLOW**
  Dieser Datensatz beinhaltet die DFLOW-spezifischen Einträge. Dazu gehört der sog. *Slot State* – der aktuelle Verarbeitungszustand des Eintrags. Weiterhin hält der Eintrag Daten über die Startbahn und die geltenden Flow Points. Als Zeiten sind der Zeitpunkt der Anfrage und die früheste und zugewiesene Abflugzeit Bestandteil eines Eintrags. Die DFLOW-Erweiterung der DMAP-Datensätze wurden zum Zeitpunkt der Entwicklung bereitgestellt.
- **FPATH**
  Die *Flight Path*-Datensätze halten Informationen über die Wegpunkte und deren veranschlagte Überflugzeiten. Diese Datensätze werden stets vom FDPS korrigiert, um möglichst verlässliche Prognosen über den Flugverlauf der Luftfahrzeuge zu erlauben.

- **SFPL**
  Alle Datensätze halten für jeden Eintrag eine Referenznummer – die SFPI – zum dazugehörigen *System Flight Plan* – oder auch SFPL. Dadurch wird die Verbindung zwischen weiterführenden Informationen zum Flugplan gemacht. Der SFPL beinhaltet alle Details eines regulären Flugplans mit einigen systemspezifischen Zusatzinformationen.

Die oben genannten Datensätze bieten alle Informationen, die eine DFLOW-Verarbeitung eines Flugplans benötigt. Die Idee ist es möglichst effiziente Entscheidungen anhand von abgefangen Transaktionen der DMAP zu treffen, um dabei die Zugriffe auf die Datensätze zu reduzieren.

5.2.2. Verarbeitungslogik

Die Aufgabe des DFLOW ist es, optimale Abflugzeiten in Abhängigkeit der betroffenen Flow Points und dem, dafür zugewiesenen, Flugverkehr zu bestimmen. Zur Bedienung der Komponente wird ein DFLOW-Display entwickelt, das nicht Bestandteil dieser Arbeit war.

Das Display bietet dem Benutzer Möglichkeiten, Flugpläne zu selektieren, sie mit Zusatzinformationen wie der Startbahn und der frühesten Abflugzeit zu vervollständigen, und die Bestimmung der optimalen Abflugzeit anzufordern. Die Anfrage der optimalen Abflugzeit initiiert die Verarbeitungslogik, hierfür wird dem erzeugten DFLOW-Eintrag ein `REQUEST`-Flag gesetzt.

1. **Ergänzung: SID-Wegpunkte**
   Bevor die eigentliche Berechnung beginnen kann, müssen alle notwendigen Informationen bereitgestellt werden. Der erste Schritt ist die Ergänzung der Route um die SID-Wegpunkte, siehe 3.1.1. Für alle internationalen Flughäfen gibt es fest definierte Routen, die in Abhängigkeit der Startbahn eine Verbindung zwischen den Startvektor und der Reiseroute herstellt.

   Im Fall DFLOW ist die vollständige Route essentiell für die Genauigkeit der Zeitenprognosen und somit ausschlaggebend für die Qualität der Optimierung.

2. **Bestimmung: Flow Points**
   Sobald die Vervollständigung der Route abgeschlossen ist, findet die erste Interaktion mit der virtuellen Maschine des ATCCL-Frameworks statt. Die virtuelle

Maschine wird mit der Ermittlung aller Flow Points beauftragt, die für den betreffenden Flugplan gelten. Dazu gilt es, den Flugplan durch die Flugplanmuster der Flow Points zu evaluieren. Details dazu sind in Abschnitt 5.1.6 zu finden.

3. **Laden: Flugpläne**
   Sobald alle Flow Points bekannt sind, werden alle Einträge im DFLOW-Datensatz geladen, die den gleichen Flow Points zugewiesen sind. Bei Flow Point-Überschneidungen kann es zu Kollisionen in der Zeit-Slot-Vergabe führen, somit müssen alle betroffenen Flugpläne bei der Vergabe der Abflugzeit berücksichtigt werden.

4. **Berechnung: Wegpunktzeiten**
   Mit Hilfe der PRISMA-Klasse `CalcEstimates` lassen sich die Wegpunktzeiten der Route bestimmen. Ausgangspunkt der Rechnung ist die *Estimated Entry Time* – oder ETN – die wir mit der frühesten Abflugzeit belegen. Die Berechnung der Zeiten ist für die Konfliktauflösung der Flow Point Slot-Zeiten notwendig.

5. **Berechnung: optimale Abflugzeit**
   Die Vorbereitungsphase ist abgeschlossen und die Berechnung der optimalen Abflugzeit wird eingeleitet. Die Kalkulation wird von der virtuellen Maschine geleistet. Die Eingabeparameter bestehen aus dem Flugplan, der zur Bearbeitung steht und den Flugplänen mit den gleichen zugewiesenen Flow Points. Der Optimierungsalgorithmus ist in 5.1.7 beschrieben. In diesem Schritt ist ebenfalls die Höhenstaffelung realisiert.

Nach Abfolge der Verarbeitungsschritte sind die Flow Points und Flugflächen zugewiesen, die optimale Abflugzeit bestimmt und die resultierende Flow Point-Überflugzeit ermittelt. Die gewonnen Informationen werden durch eine Transaktion an den DMAP-Datensatz übermittelt. Mit der Übermittlung beendet die Verarbeitung des DFLOW, für die Visualisierung der DFLOW-spezifischen Daten ist das DFLOW-Display zuständig. Zur Verifizierung der Verarbeitungsterminierung wird dem Eintrag das `PROPOSAL`-Flag gesetzt.

Eine automatisierte Vergabe von Abflugzeiten ist stets nur ein Vorschlag, welcher vom Benutzer überstimmt werden kann. Dies stellt eine wichtige Anforderungen an das System dar. Eine voll automatisierte Verarbeitung bietet nicht die Flexibilität, die in einem Umfeld wie der Flugsicherung zu einem reibungsfreien und effektiven Betrieb notwendig ist. Treten Ausnahmesituationen ein, oder wird die Qualität der Automatisierung

durch andere Faktoren gestört, so muss es möglich sein, den Flugverkehr ordnungsgemäß in manueller Steuerung abzuhalten. Ist die manuelle Steuerung unmöglich, so kann das zu einer Beeinträchtigung der Flugsicherung und damit der Sicherheit oder Effektivität führen, was letztendlich mit der Abschaltung des Systems enden kann, [Klu07].

5.2.3. Protokollierung

DFLOW bedient sich der DMAP, welche alle Transaktionen protokolliert. Diese Zustand erleichtert die Nachvollziehbarkeit der Entscheidungen, die von der DFLOW-Komponente automatisch oder manuell gemacht werden. Zur Ergänzung der statistischen Auswertemöglichkeiten des FDPS, sollen zusätzlich DFLOW-spezifische persistente Logdateien erzeugt werden. Ein Logeintrag soll einen Starter mit relevanten Zusatzinformationen wie dessen Rufzeichen, zugewiesene Flow Points und den Zeiten protokollieren.

Um eine automatisierte Auswertung solcher Protokolle zu erleichtern, wird für die DFLOW-Protokolle ein CSV-Datenformat verwendet. Ein Protokolleintrag hat folgende Datenfelder:

- *Call Sign:* das Rufzeichen des Luftfahrzeugs
- *ADEP:* die ICAO-Kennung des Startflughafens
- *RWY:* die Kennung der Startbahn
- *ADES:* die ICAO-Kennung des Zielflughafens
- *ETOT:* die früheste Abflugzeit – vom Planer gesetzt
- *ATOT:* die optimierte Abflugzeit – von DFLOW berechnet
- *Flow Point:* der von DFLOW zugewiesene Flow Point
- *Flow Time:* die Prognose der Überflugzeit über den Wegpunkt des Flow Points
- *Flow Flight Level:* die von DFLOW zugewiesene Flugfläche
- *ATD:* die tatsächliche Abflugzeit – von PRISMA automatisch erfasst

6. Realisierung

6.1. Programmiersprache & Hilfsbibliotheken

Alle Module in der PRISMA-Architektur sind in C++ realisiert. Zur nahtlosen Integration in das bestehende System und entsprechend der fachlichen Qualifikation der Entwickler wurden auch die Komponenten dieser Arbeit in C++ entwickelt.

C++ eignet sich für systemnahe und objektorientierte Programmierung. Aufgrund einer fehlenden automatischen Speicherverwaltung, auch *Garbage Collection* genannt, sind einige Richtlinien nach [Com05] zu befolgen.

Zur Entwicklung von robuster Software, wie sie in sicherheitskritischen Systemen verlangt wird, soll u.a. auf dynamische Speicherallokierung verzichtet werden. Wobei der Standard die Laufzeit eines Systems in eine Initialisierungsphase und eine Betriebsphase teilt, in der Initialisierungsphase ist die Allokierung von benötigten Speicher erlaubt, [Com05]. Auch *Stack*-basierte Allokationen sind möglich, oft unumgänglich. [Com05] beschreibt die Initialisierungsphase als die Zeit unmittelbar nach dem Start eines Systems bis zum Zeitpunkt der Bereitstellung aller Funktionen. Somit ist ein Zeitraum gegeben, um auch dynamische Strukturen zu initialisieren, ohne eine Gefährdung der Stabilität eines Systems zu riskieren.

6.1.1. Compiler

Der ATCCL-Parser wurde mit Hilfe von `flex` und `bison` entwickelt und ist daher in der Programmiersprache C realisiert. Nach der Evaluation der C++-Varianten beider Werkzeuge hat sich die schwierige Integration der generierten Module bemerkbar gemacht, die aufgrund von Versionsinkompatibilitäten entstanden sein muss. Gleichzeit war der Aufwand solche Konflikte aufzulösen nicht gerechtfertigt, da der Vorteil eines objektorientierten Designs an dieser Stelle vernachlässigbar ist.

Die C-Versionen der beiden Werkzeugen sind ausgereift, stabil und bieten eine pro-

blemlose Integration. Die Effektivität der Compilerentwicklung wurde durch den Einsatz von `flex` und `bison` erheblich gesteigert. Die Entwicklung der Analysephase des Compilers fand in C statt, während die Synthese, in C++ realisiert, eine C-Schnittstelle bietet, um eine erfolgreiches Zusammenspiel zu gewährleisten.

6.1.2. Comsoft `stdbase`

Die *C++ Standard Library* schließt dynamische Speicherallokierung nicht aus, in den gängigen Implementierungen, so auch in dem `GCC`, wird davon auch intensiv gebraucht macht. Dieser Zustand macht den Einsatz der Standardbibliothek für die Entwicklung von PRISMA-Modulen nicht möglich.

Die `stdbase`-Bibliothek bietet eine Reihe von Klassen an, die entsprechend der sicherheitstechnischen Richtlinien arbeiten. Diese bieten ein vergleichbares Interface wie die *C++-STL* an, sodass erfahrene C++-Entwickler ohne Einlernungsaufwand diese bedienen können.

Als Beispiel bietet die `stdbase` eine längenbeschränkte `String`-Klasse an, die eine *Stack*-basierte Variante der `std::string` mit fast identischem Interface darstellt.

Bisher bot die `stdbase` keine Variante der `std::vector` und `std::map` an. Die Verwendung dieser Klassen würde an vielen Stellen in dem Entwurf Anwendung finden und die Entwicklung effektiver machen. `StackVector` (Abb. 6.1) wurde für lokale Objektsammlungen mit begrenzter Lebenszeit konzipiert und bei den Auswertungsprozeduren verwendet. Die Klasse bietet ein ähnliches Interface, wie es die `std::vector`-Klasse bietet, jedoch mit einigen Einschränkungen wie einer maximalen Kapazität. Außerdem ist das Verhalten beim Löschen beschränkt, nur das Löschen des jeweils letzten Elements ist erlaubt.

6.1.3. CppUnit

CppUnit ist ein C++-Testframework. Mit Hilfe von Präprozessormakros bietet das Framework Möglichkeiten, Klassen nach potentiellen Fehlern, d.h. Abweichungen von deren Spezifikation, zu testen. Die Auswertung der Unit-Tests wird im XML-Format bereitgestellt und kann somit bequem aufbereitet werden.

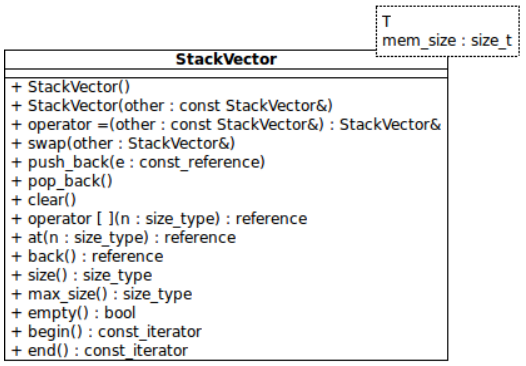


Abbildung 6.1.: `StackVector`

6.1.4. Code Coverage

Code Coverage bezeichnet die verschiedenen Metriken zur Festlegung der Codeabdeckung von Tests. Bei der Integration verschiedener Überdeckungsanalysen ergibt sich eine verlässliche Statistik über die Testqualität. Diese kann dazu genutzt werden, Schwachstellen in der Testerstellung auszumachen und zu beseitigen.

6.2. Entwicklungsumgebung

6.2.1. IDE

Der Großteil der Implementierungsarbeit fand in der *Eclipse IDE* statt. *Eclipse* bietet als Framework ein gelungene Schnittstelle an, um die Funktionalität der IDE anzupassen oder zu erweitern. Für die C++-Entwicklung gibt es ein Plugin, das eine vollwertige Entwicklungsumgebung bereitstellt. Die Vorteile der Nutzung einer integrierten Entwicklungsumgebung sind:

- Syntaxhervorhebung (*Syntax highlighting*)
- Autovervollständigung (*Autocomplete*)
- Integration mit anderen Werkzeugen wie Unit-Tests, Compiler, Versionierungssystem

Zudem bietet Eclipse genügend Anpassungsmöglichkeit des Editors, um diesen entsprechend den Comsoft-Richtlinien zu konfigurieren.

Weiterhin kamen die Texteditoren *Vim*[1] und *Emacs* und zum Einsatz, u.a. bei der Entwicklung der Python-Skripte.

6.2.2. Versionsverwaltung

Ein Unternehmen für die Entwicklung sicherheitskritischer Systeme ist stets bestrebt, sowohl die Produkte als auch den Entwicklungsprozess sicher zu gestalten. Dies hat mitunter ein konservatives Verhalten bei der Technologiewahl als Konsequenz. Erprobte und bewährte Systeme und Prozesse werden nur nach gründlicher Voruntersuchung der Alternativen ersetzt, so auch im Falle des Versionsverwaltungssystems. So wurde für dieses Projekt *CVS*[2] für die Versionsverwaltung des Quellcodes und der Systemkonfigurationen eingesetzt.

6.2.3. Betriebssystem

Das eingesetzte Betriebssystem während der Entwicklung war *Red Hat Enterprise Linux 5.2*. Auch die Zielsysteme sind mit einem Red Hat Betriebssystem konfiguriert.

6.3. Dokumentation & Entwurf

Die Comsoft-Dokumentation wird in *Microsoft Word* erstellt, basierend auf hauseigenen Vorlagen. Für die Ausarbeitung dieses Buches wurde LaTeXverwendet.

Für die Erstellung von Vektorgrafiken wurde *Inkscape (`www.inkscape.org`)* eingesetzt. Zur Entwicklung des UML-Entwurfs wurden *MagicDraw™UML* und der *Umbrello UML Modeller (`www.uml.sourceforge.net`)* verwendet. Die Codedokumentation wird automatisch mit *Doxygen (`www.stack.nl/~dimitri/doxygen`)* generiert.

Für die grafische Auswertung der aufbereiteten Protokolldaten des operativen Betriebs diente das Programm *Gnuplot (`www.gnuplot.info`)*.

6.4. ATCCL

6.4.1. `flex`-Konfiguration

`flex` unterstützt reguläre Ausdrücke zur Bestimmung der Token. Anhand einfacher Regeln und der Verwendung von Metazeichen (Abb. 6.1) kann nach Zeichenkettenmuster

[1] Vi Improved – eine Weiterentwicklung des konsolenbasierten Texteditors vi
[2] Concurrent Versions System – ein dateibasiertes Versionsverwaltungssystem für Quellcode

gefiltert werden. Dafür entwickelt man Regeln, die auf der linken Seite einen regulären Ausdruck erhalten, während auf der rechten Seite die Aktion festgelegt wird.

Die Definition von Zeichenklassen erhöht den Ausdruckswert von komplexeren Regeln und dadurch deren Verständlichkeit. Für ATCCL wurden Zeichenklassen erstellt, die Basistypen der Sprache definieren.

```
delim     [ \t]
ws        {delim}+
letter    [A–Za–z]
digit     [0–9]
id        {letter}({letter}|{digit}|_)*
real      {digit}+\.{digit}+
```

Listing 6.1: ATCCL `flex`-Konfiguration: Zeichenklassen

Metazeichen	**Bedeutung**
`.`	beliebiges zeichen außer Zeilenumbruch
`\n`	Zeilenumbruch
`\t`	Tabulator
`*`	null- oder mehrfache Kopie eines Ausdrucks
`+`	ein- oder mehrfache Kopie eines Ausdrucks
`?`	null- oder einfache Kopie eines Ausdrucks
`^`	Zeilenanfang
`$`	Zeilenende
`a|b`	`a` oder `b`
`(ab)+`	ein- oder mehrfache Kopie von `ab`
`'a+b'`	`'a+b'` Literal
`[]`	Zeichenklasse

Tabelle 6.1.: `flex`-Metazeichen *(Quelle: [LS], [Nie])*

Nun werden für die Schlüsselwörter der Flugplaneigenschaften entsprechende Token erstellt.

```
copx          { return COPX; }
adep          { return ADEP; }
ades          { return ADES; }
rwy           { return RWY; }
route         { return ROUTE; }
atyp          { return ATYP; }
equip         { return EQUIP; }
tas           { return TAS; }
frul          { return FRUL; }
ftyp          { return FTYP; }
travel_type   { return TRAVEL_TYPE; }
```

Listing 6.2: ATCCL `flex`-Konfiguration: Flugplaneigenschaften

Zur Bestimmung der Separationszeiten wird auf eine weitere Flugplaneigenschaft zugegriffen – die Flugfläche, dies ist nur innerhalb einer Separationsdefinition möglich. `time_sep` bezeichnet die Art der Separation, in diesem Fall die Längsstaffelung nach Zeit.

```
time_sep { return TIMESEP; }
fl       { return FL; }
```

Listing 6.3: ATCCL `flex`-Konfiguration: Separationstyp und Flugfläche

Die folgenden Regeln gelten der Identifizierung des Regeltyps.

```
constraint { return CONSTRNT; }
pattern    { return PATTERN; }
flowpoint  { return FLOWP; }
```

Listing 6.4: ATCCL `flex`-Konfiguration: Typ der Regeldefinition

Die Operatoren bestehen aus einem oder mehreren Token. Zusammengesetzte Operatoren sind z.B. `not in`, `greater than` und `is not`.

```
is       { return IS; }
in       { return IN; }
not      { return NOT; }
```

```
less      { return LESS; }
greater   { return GREATER; }
than      { return THAN; }
and       { return AND; }
or        { return OR; }
```

Listing 6.5: ATCCL `flex`-Konfiguration: Operatoren

Zusätzlich werden eine Reihe von Hilfstoken zur deskriptiven Definition der Separationsregeln erstellt.

```
at        { return AT; }
from      { return FROM; }
until     { return UNTIL; }
on        { return ON; }
```

Listing 6.6: ATCCL `flex`-Konfiguration: `Constraint`-Token

Das yylval wird in der `bison`-Konfiguration als `union` von `integer, double` und `char*` deklariert. Dadurch wird eine typensichere Wertübertragung von Konstanten in die Syntaxanalyse ermöglicht.

```
{digit}+ { yylval.integer = atoi(yytext); return INTEGER; }
{real}   { yylval.real = atof(yytext); return REAL; }
{string} { yylval.string = installString(); return STRING; }
```

Listing 6.7: ATCCL `flex`-Konfiguration: Basisdatentypen

Zur zusätzlichen Unterstützung der nachfolgenden Analyse wird eine Regel für die Regeldefinitionsidentifikation gestellt und die Kommentare lokalisiert. Der Kommentarinhalt wird nicht übergeben sondern ignoriert. Die letzte Regel leitet alle Zeichen weiter, die bisher in keiner Regel aufgefangen wurde, wie z.B. die Klammern.

```
{id}      { yylval.string = installString(); return ID; }
\"        { return QUOTE; }
#.*$      { return COM; }
{ws}      {  }
.         { return yytext[0]; }
```

Listing 6.8: ATCCL `flex`-Konfiguration: IDs und Kommentare

Eine Besonderheit in der `flex`-Konfiguration stellt die letzte Zeile dar, die bei Zeilenende den Inhalt der Zeile in einem Puffer zwischenspeichert und einen Zeilenzähler inkrementiert. Diese Regel dient der Fehlerbehandlung. Durch das Puffern der Zeilen ist im Fehlerfall eine weitere Analyse möglich, außerdem kann man durch die Zeilenangabe den Fehler lokalisieren.

```
\n.*      { strcpy(line_buffer, yytext+1);
            line_no++; yyless(1); }
```

Listing 6.9: ATCCL `flex`-Konfiguration: Fehlerbehandlung

6.4.2. `bison`-Konfiguration

Die Syntaxanalyse wird mit Hilfe von `bison` erstellt. Das Werkzeug erlaubt es in EBNF-ähnlicher Notation die Produktionen zu entwickeln und ist mit `flex` integrierbar.

Die Integration mit `flex` basiert auf der zustandsbehafteten Funktionsrückgabe der Funktion `yylex`, welche die Token zur Weiterverarbeitung an `bison` liefert. Die vollständige Konfiguration befindet sich in Anhang B.1.

6.4.3. Synthese

Der Entwurf sah die Nutzung des Factory Patterns zur generischen Objekterzeugung vor. Die Umsetzung wurde entsprechend durchgeführt. Es wurden zwei Factory-Klassen mit jeweils mehreren überladenen Funktionen erstellt. Anhang A.1.1 beschreibt die implementierten Klassen.

Die Analysephase erfasst die Operator- und Property-Typen und übergibt diese an die Factory-Klasse. Aus den verschiedenen Kombinationen ergeben sich die Term-Instanzen. Das *Adapter Pattern* nach [GHJV95] erlaubt eine generische Lösung für die

Klassendefinition der Flugplaneigenschaften (Properties). Die Klassendefinition einer Flugplaneigenschaft wird in den typenabhängigen Rumpf, das `Property`-Template und eine Adapterklasse verteilt. Der Rumpf erhält den Datentyp der betreffenden Eigenschaft und den Adapter als Template-Parameter. Für jede Flugplaneigenschaft wird ein Adapter definiert, welcher mit Hilfe des FlightPlan-Interface die entsprechenden Dateneinträge in den Cache des Property-Objekts lädt.

Muss das ATCCL-Framework zu einem späteren Zeitpunkt um die Unterstützung einer weiterer Flugplaneigenschaft erweitert werden, so ist es notwendig einen Adapter dafür zu definieren und die Factory soweit zu erweitern, dass dieser Typ instanziiert wird. Selbstverständlich muss auch die `flex`- und `bison`-Konfiguration die neue Flugplaneigenschaft unterstützen und dieser einen Datentyp zuordnen. Diese Methode erlaubt ein strukturiertes Erweitern der Sprache und reduziert dabei den Implementierungsaufwand durch die Auslagerung aller gemeinsamen Funktionalität.

6.5. DFLOW

Zur Realisierung der DFLOW-Komponente konnte man auf den großen Funktionsumfang von PRISMA zugreifen. So war es nicht notwendig eigene Routinen zur Erstellung der Prognosen über die Überflugzeiten zu entwickeln, diese Funktion wurde bereits von *CalcEstimates* bereitgestellt. FPATH übernimmt als FPDS-Komponente die Aktualisierung solcher Routentabellen samt prognostizierter Zeiten entsprechend dem realen Flugverlauf. Die Ergänzung der Flugroute unmittelbar nach dem Startvorgang wurde mit Hilfe von SID-Lookup-Tabellen realisiert, eine Erklärung zur Bedeutung der SIDs befindet sich in Abschnitt 3.1.1. Die SID-Wegpunkte sind für jede Kombination von Flughafen, Startbahn und Routenverlauf festgelegt und statisch.

6.5.1. FDPS

Die DFLOW-Komponente wurde in den FDPS integriert. Die Komponente ist eine flugplanverarbeitende Instanz und interagiert mit anderen Komponenten des FDPS oder auch den Displays durch das DMAP-Protokoll.

Sog. *Notifier*-Klassen implementieren das Observer Pattern für bestimmte Datensätze der DMAP-Datenbank. *Storage*-Klassen bieten eine transparente Zugriffsschicht auf DMAP-Datensätze. Durch die beiden Klassentypen ist es möglich, Event-getrieben und dadurch effektiv auf der DMAP zu arbeiten und gleichzeitig jederzeit volle Zu-

griffsmöglichkeit mit Hilfe von transparenten Transaktionen auf alle notwendigen Datensätze zu haben.

6.5.2. Node Manager

Der Node Manager ist für die Kontrolle der Vitalität der einzelnen PRISMA-Komponenten zuständig. Basierend auf dem Watchdog-Verfahren werden Prozesse aktiv angesprochen um die Reaktionszeiten zu messen.

Ist die Vitalität des Systems durch eine Komponente gefährdet, so kann der Node Manager den Neustart oder die Deaktivierung der identifizierten Komponente mit Fehlfunktion veranlassen. Der Node Manager bietet für dessen Protokoll eine Klasse an, die eine voll automatisierte Lösung des Watchdog-Verfahrens auf Client-Seite darstellt.

6.5.3. AWP

Die AWP bietet eine Reihe strategischer Displays als integrierte Lösung an. Ein Planer hat u.a. die Aufgaben Flugpläne den Startern zuzuordnen, Prognosen über Verkehrsaufkommen zu analysieren und basierend darauf Entscheidungen zu treffen.

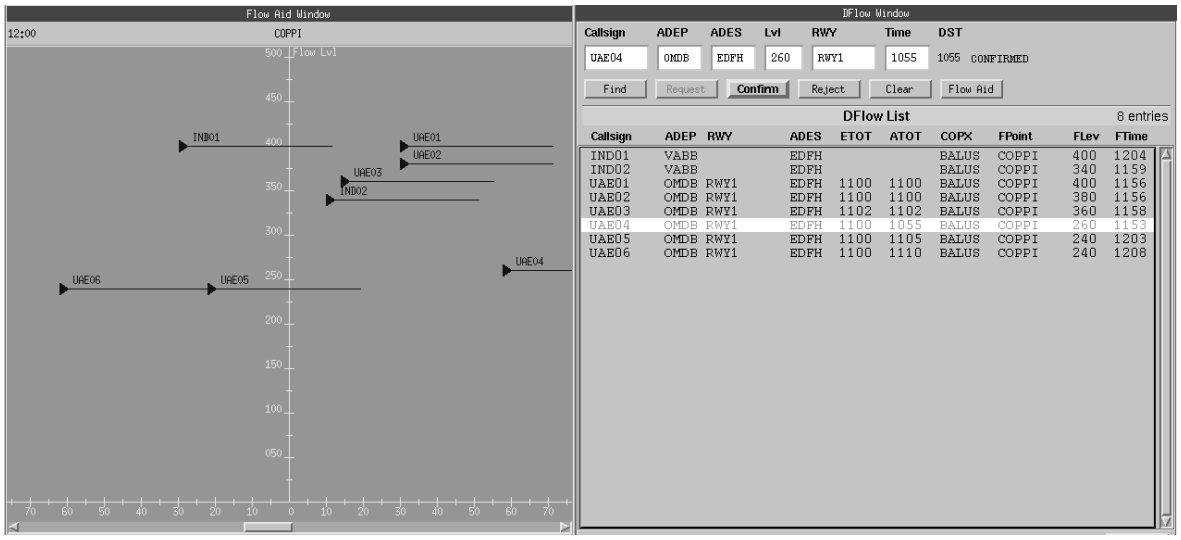


Abbildung 6.2.: Links: das Flow Aid Window zeigt eine graphische Übersicht der Flow Point Slot-Verteilung über Zeit und Flugfläche. Rechts: das DFLOW Window stellt die Interaktion mit der DFLOW-Komponente dar samt Übersicht über die erfasste Flugpläne.

Die DFLOW-Komponente soll mit Hilfe eines in die CWP integrierten Displays zu steuern sein. Die Entwicklung des DFLOW-Displays war ein Teil des Projekts, die Dokumentation dessen ist aber nicht Teil dieser Arbeit. Das Display bietet eine Übersicht über alle DFLOW-Dateneinträge, d.h. alle DFLOW-kontrollierten Flüge. Das DFLOW-Display erlaubt es dem Benutzer nach Flugplänen zu suchen und mit der Übergabe der frühesten Abflugzeit die Berechnung des optimalen Abflugslots zu initiieren.

6.5.4. CWP

Das CWP-Display ist das Kernstück des Flugsicherungssystems aus der Sicht des Benutzers. Die Controller erhalten mit diesem Display eine Übersicht über das Luftlagebild, potentielle Gefahrensituationen und erlaubt ein intelligentes Verteilen von Zuständigkeitsbereichen. Im Gegensatz zur AWP ist das Display für die unmittelbare Sicherung des Luftraums entwickelt, jedoch soll auch hier bei Bedarf die Information der CWP bereitgestellt werden.

Das DFLOW-Window wurde dazu auch in die CWP integriert. Da die Abflugplanung nicht unter die Zuständigkeit der Flutlotsen, die an der CWP aktiv sind, fällt, ist das DFLOW-Display hier als *read-only*-Variante mit Echtzeitverfolgung realisiert.

7. Verifikation

7.1. Werkzeugeinsatz

Nach Beendigung der Entwicklungsphase war man in der Lage ein Fazit über die Qualität der eingesetzten Werkzeuge zu stellen. Deren Einsatz wurde hinsichtlich der Effektivitätssteigerung bewertet. Tabelle 7.1 bietet eine quantitative Übersicht über die Komplexität der entwickelten Komponenten.

Komponente	**Lines of Code**	**autogenerierter Anteil**
`flex`-Konfiguration	84	-
`bison`-Konfiguration	348	-
autogeneriert	4188	-
ATCCL	13741	30%
DFLOW	5796	0%
Gesamt	19537	21%

Tabelle 7.1.: Anteil an automatisch generiertem Code

Das ATCCL-Framework ist in Codezeilen bemessen der Schwerpunkt der Entwicklung gewesen. Mit einem Konfigurationsaufwand von 432 Zeilen Code, wurde mit Hilfe von `flex` und `bison` rund 30% des ATCCL-Frameworks generiert. Die quantitative Auswertung beinhaltet nicht die hohe Komplexität einer Parsergenerierung und der notwendigen Testprozeduren, um die Fehlerfreiheit des Parsers zu gewährleisten. Zusammenfassend kann man behaupten, mit dem Einsatz der Parsergeneratoren eine enorme Steigerung der Produktivität erzielt zu haben.

Nach [Glo08] bietet der Einsatz von Codegeneratoren auch beim Zertifizierungsprozess einen Vorteil gegenüber manuell entwickeltem Code. Bei dem ATCCL-Framework

wird die Zertifizierung der Parsergeneratoren nicht notwendig sein, da der automatisch generierter Code ausschließlich im Compiler-Front-End zum Einsatz kommt. Das Front-End ist für die Syntax- und Semantikanalyse zuständig, welche nur bei der Initialisierung durchgeführt werden. Fehlverhalten in der Initialisierungsphase bedeutet keine Gefährdung der Sicherheit, solange diese erfolgreich erkannt werden und somit die operative Phase verhindert wird. Die Synthese erfolgt durch manuell entwickelten Code, die generierten Objekte daraus werden auch zur Laufzeit aktiv.

7.2. Unit-Tests

Während der Implementierungsphase wurden für die C++-Klassen Unit-Tests erstellt, die das Verhalten der Einheiten gegen die Anforderungen prüfen. Mit Hilfe von Überdeckungsdiagnosen werden nicht ausreichend getestete Module lokalisiert und die ungeprüften Bereiche mit weiteren Tests belegt.

Zu den kritischen Modulen gehören u.a. `VirtualMachine`, `StackVector` und `TermFactory`. Die `VirtualMachine` beherbergt die zeitbestimmenden Algorithmen und muss deshalb besonders intensiv getestet werden. Hier ist nicht nur die Korrektheit der Ergebnisse relevant, sondern auch die Laufzeiten der Berechnungen, mehr dazu in Abschnitt 7.5. Die StackVector-Klasse dient als Standardklasse für eine Reihe von Situationen, Fehler in dieser Klasse würden Konsequenzen für eine Reihe von Modulen haben. Die TermFactory ist als objekterzeugende Einheit wegen der dynamischen Speicherallokierung kritisch. Gleichzeitig bieten die Tests dieser Klasse eine unmittelbare Prüfung für neu integrierte Property-Klassen.

7.3. Testspezifikation

Die Testspezifikation – auch System Test Description – wird analog zu dem SRS[1] entwickelt. Jeder Test hat mindestens eine Anforderung abzudecken, alle Anforderungen müssen mindestens in einem Test verifiziert werden.

Die Testspezifikation beschreibt unabhängige Tests. Ein Test besteht aus einer Vorbereitungssequenz und den Testschritten. Jeder Test muss die getesteten Anforderungen und die zu testende Version der Software referenzieren.

[1] System Requirements Specification – Dokument zur Spezifikation der Systemanforderungen

Ein Testschritt beschreibt eine Aktion und den erwarteten Effekt. Tritt dieser Effekt nicht auf, oder ist die Durchführung der Aktion nicht möglich, gilt der Testschritt als fehlgeschlagen. Bei Fehlschlag findet eine Gewichtung des Fehlers statt, die in Abhängigkeit von der Priorität der getesteten Anforderung und der Art des Fehlereffekts ermittelt wird.

Am Ende des Tests wird ermittelt ob der Durchlauf erfolgreich war. Sind während des Durchlaufs Fehler aufgetreten, hängt die Gesamtbewertung von der Gewichtung und Anzahl der Fehler ab.

Die Granularität der Testabdeckung spielt eine entscheidende Rolle bei der Effektivität der Testläufe. Ist diese zu hoch, entstehen möglicherweise viele redundante Testschritte, die einen Testdurchlauf unnötigerweise verlangsamen. Deckt man hingegen zu viele Anforderungen mit einem Test ab, so erschwert es die Lokalisierung des Fehlverhaltens.

7.4. Testdurchführung

Die Tests werden entsprechend der STD[2] durchgeführt. Für die Durchführung der Tests sind die Testingenieure des Testteams verantwortlich.

Die Verifizierungstests werden bei jeder neuen Softwareversion durchgeführt. Bei verteilten Systemen kann eine Komponente Einfluss auf andere Komponenten haben, in diesem Fall müssen alle in Verbindung stehenden Komponenten auch getestet werden, dies sind die sog. *Regression Tests*[3].

Da das ATCCL-Framework eine eigenständige Library ist, ist diese größtenteils unabhängig von anderen Modulen und muss nur bei neuen Versionen getestet werden. Die DFLOW-Komponente hingegen ist in das FDPS integriert. Bei jeder Aktualisierung des FDPS oder einer Komponente, die in Verbindung mit dem FDPS steht, muss auch DFLOW neu verifiziert werden.

[2] System Test Description – Dokumentation der Systemtestprozeduren

[3] Testprozeduren, die mit jeder neuen Version wiederholt durchgeführt werden, um alle in Abhängigkeit stehenden Komponenten auf ihre Korrektheit zu überprüfen

7.5. Effizienz

Beansprucht das DFLOW zu viel Bearbeitungszeit für eine Anfrage, so kann es den Betrieb eines Planers stören. Eine höhere Belastung als bei der manuellen Eingabe ist die automatische Übernahme von Überflügen in das DFLOW-System. Da die Komplexität der Bearbeitung in der virtuellen Maschine des ATCCL-Frameworks liegt, muss diese auf ihre Leistungsfähigkeit geprüft werden.

Zum Testen der Performance der virtuellen Maschine wurde eine Reihe von Worst-Case-Szenarien mit Hilfe von Unit-Tests entwickelt und diese auf einer GCC-optimierten Version ausgeführt. Dafür wurden Flugplanmuster mit unwahrscheinlichen und sehr vielen Kombinationen gewählt. Die Testdurchläufe wurden auf Systemen durchgeführt, die auch beim Kunden in Einsatz kommen.

Die Evaluation ergab einen Mittelwert von 11 µs pro Auswertung eines Flugplanmusters, was bei der Evaluation von 3000 Flugplänen eine Gesamtzeit von 33 ms ergibt. Im Realbetrieb werden sich maximal 1000 bis 1500 Flugpläne zum gleichen Zeitpunkt im System befinden, bei einem Umsatz von ca. 150 pro Stunde.

7.6. Leistungsanalyse

Das Ziel der DFLOW-Komponente bestand in der Optimierung der Abflugplanung unter Berücksichtigung der Luftraumbeschränkungen. Nach der erfolgreichen Realisierung aller Anforderungen, galt es die Leistung der Komponente im operativen Betrieb zu bewerten.

7.6.1. Analysewerkzeuge

Die Bewertung basiert auf den von DFLOW erstellten Logdaten und den DMAP-Transaktionsprotokollen. Die DFLOW-Logs enthalten einen Eintrag für jeden erfassten Starter, während die *Daily Movement Logs* – kurz DML – alle Flüge innerhalb des kontrollierten Gebiets protokollieren.

Durch die detaillierte Protokollführung entstehen große Datenmengen. Die tägliche Datenmenge misst im Durchschnitt ca. 350 kB für die DFLOW-Logs, 6 MB für die DMLs und 60 MB für die DMAP-Transaktionen. Um eine Analyse dieser Daten zu ermöglichen, wurden *Python*[4]-Skripte entwickelt. Die Skripte dienen der Vorverarbei-

[4]Eine dynamische, höhere Programmiersprache, der gute Lesbarkeit attestiert wird – wird durch die

tung der Datensätze, im Detail:

- `dflowlog.py` ist das Basismodul zur Interaktion mit DFLOW-Logs
- `filter.py` dient zum Entfernen nicht relevanter Datensätze
- `analyse.py` bietet mehrere Auswertemöglichkeit, u.a. Differenzenbildung von Zeiten und Ermittlung der zeitlichen Separation der individuellen Flüge
- `pushback.py` versetzt Spalteneinträge nach hinten, erleichtert den Einsatz von cutbase.py
- `cutbase.py` entfernt nicht relevante Spalten, als Vorbereitung zur Weiterverarbeitung mit `gnuplot`

Zusätzlich wurde nach dem Prinzip des *Data-Warehouse* eine *MySQL*[5]-Datenbank angelegt. In die Datenbank wurden Daten in aufbereiteter, auf den aussagekräftigen Informationsteil reduzierten Form aus den verschiedenen Protokollquellen importiert. Die Datenbank bietet eine Möglichkeit mit SQL-Abfragen dynamische Auswertungen und Zusammenhänge zwischen verschiedenen Datensätzen zu bilden. Die Extraktion, Transformation und das Laden in die Datenbank wurde mit einem weiteren Python-Skripten realisiert:

- `dml.py` ist das Basismodul zur Interaktion mit DML-Dateien
- `dml2db.py` dient zur Extraktion relevanter DML-Einträge und dem Laden der Daten in die Datenbank

7.6.2. Datensatz

Die Analysen basieren auf den Aufzeichnungen von 1. Januar bis zum 8. April 2010. Der Datensatz basiert auf den Logdaten eines von mehreren FDPS-Servern. Da die Server in einem redundanten System agieren und die aktuellen Daten zum Zeitpunkt der Analyse noch nicht zusammengeführt waren, waren keine absoluten Bewertungen der Daten möglich, wie z.B. hinsichtlich der Verkehrsflussdichten.

hohe Abstraktion und weite Verbreitung oft als Skriptsprache eingesetzt

[5]Ein Open-Source-Datenbankmanagementsystem

7.6.3. Auswertung

Abbildung 7.1 zeigt die Verteilung der Differenz zwischen der frühesten Abflugzeit *ETOT* und der von DFLOW vergebenen Abflugzeit *ATOT*. Die Gegenüberstellung mit der Differenz der vergebenen Abflugzeit und der tatsächlichen Abflugzeit soll die Konformität der Abflugplanung mit der DFLOW-Vergabe visualisieren.

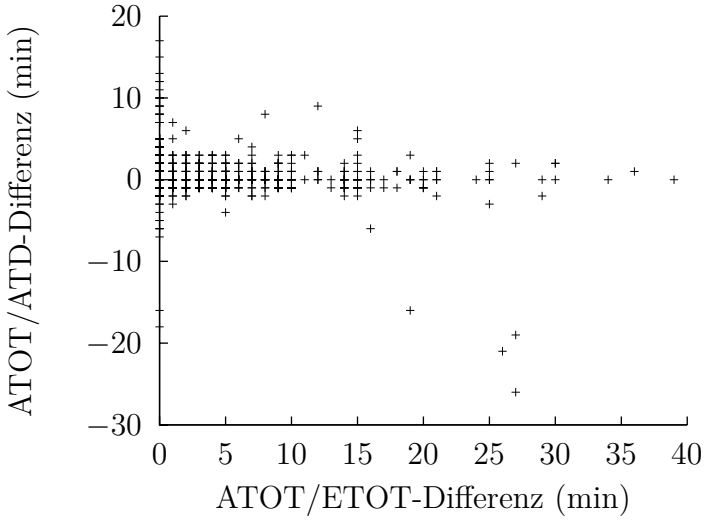


Abbildung 7.1.: Abflugzeitkonformität

Die Analyse der rund 1800 Flüge ergab, dass die vergebenen Zeiten zum größten Teil sehr strikt eingehalten werden. Man sieht eine starke Konzentration um den Nullpunkt auf der X-Achse. Die Streuung beschränkt sich, bis auf wenige Ausnahmen, auf eine Abweichung von ±3 Minuten.

Interessant ist eine kleine Gruppe von abweichenden Zeiten im Bereich (26, -20). Die Flüge in diesem Bereich wurden zur Einhaltung der Separationsregeln um rund 25 Minuten zeitlich versetzt. Die tatsächliche Abflugzeit liegt jedoch 20 Minuten vor der vergebenen Zeit, also unmittelbar nach der frühesten Abflugzeit. Die Entscheidung der Planer die vergebene Zeit nicht zu übersteuern und trotzdem den frühzeitigen Abflug zu gestatten, mag an der Funktion von DFLOW liegen, solche Verletzungen der Abflugslots visuell zu signalisieren. Sowohl der Planer an der AWP als auch der Fluglotse an der CWP werden durch eine definierte Zeichenkette über die Abweichung informiert und diesen Flügen besondere Aufmerksamkeit bei der Eingliederung in den Luftverkehr

widmen.

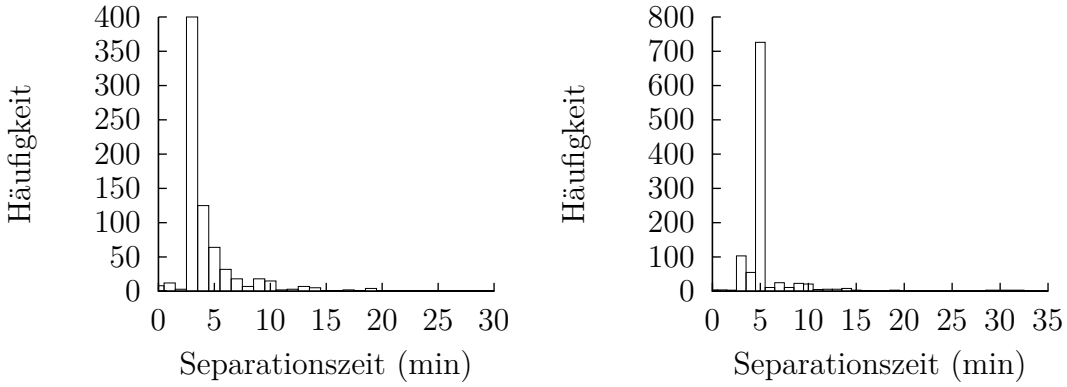


Abbildung 7.2.: Histogramm von den Separationszeiten an zwei verschiedenen Flow Points. Links: *LALDO*, rechts: *LABTAR*

Eine wichtige Anmerkung zu der folgenden Auswertung ist, dass die Separationsbeschränkungen auf Zeiträume mit hohem Verkehrsaufkommen gelegt werden. Da in diesen Zeiträumen der Großteil des täglichen Flugverkehrs abgehandelt wird, hat dies Auswirkung auf die Statistik.

Flow Point *LALDO* umfasst zwei Wegpunkte und definiert in der Summe ca. 6 Stunden am Tag eine zeitliche Separation von 3 Minuten. Die Statistik belegt diesen Zustand und die erfolgreiche Funktion des DFLOW: für über 54% der erfassten Flüge wurde von DFLOW eine Separation von 3 Minuten geplant. Die größeren Separationen zwischen 3 und 10 Minuten sind ein Zeichen dafür, dass der Flow Point noch freie Kapazitäten besitzt. Zum Vergleich weist *LABTAR* im Durchschnitt eine höhere Verkehrsflussdichte von 29% auf.

Auch für den Flow Point *LABTAR* sind zwei Wegpunkte für die Separation definiert. Die Separationskonfiguration für diesen Flow Point ist differenzierter und umfasst mehrere Zeiträume mit unterschiedlichen Zeiten. Rund 7,5 Stunden pro Tag gilt eine Separation von 5 Minuten, während ca. 1,5 Stunden eine zeitliche Separation von 3 Minuten gewährleistet werden soll. Das Histogramm zeigt dieses Verhalten mit den beiden größten Ausschlägen, 70% der Separationszeiten liegen bei 5 Minuten und 10% bei 3 Minuten.

8. Zusammenfassung

8.1. Fazit

Am Ende dieser Arbeit kann festgestellt werden, dass die Zielsetzung vollständig erreicht wurde. Der Projektverlauf war wie geplant, alle Phasen wurden erfolgreich abgeschlossen. Die Verifizierung ergab ein positives Bild über die Funktionalität der entwickelten Abflugplanungskomponente und des *ATCCL-Frameworks*.

Die Wahl eine domänenspezifische Sprache für die Modellierung der Luftraumbeschränkungen zu entwickeln, hat sich als ein großer Vorteil herausgestellt. Die Sprache gewährleistet große Freiheiten bei der Konfiguration der Abflugplanungskomponente, gleichzeitig bietet sie eine hervorragende Kontrolle über die Validität der Konfiguration. Bereits während der Bearbeitungszeit der Arbeit wurde *ATCCL* in einem weiteren Projekt erfolgreich eingesetzt, was die Entwicklungszeit verkürzte und die Dynamik des Projekts erheblich steigerte.

Seit dem 27. Januar 2010 wird *DFLOW* erfolgreich im operativen Betrieb für die Bestimmung der Abflugzeiten von der *GCAA* eingesetzt. Der störungsfreie Betrieb und ein zufriedener Kunde ist ein Indiz für die Qualität der realisierten Komponenten. Die Akzeptanz der computergestützten Abflugplanung ist ein weiterer Meilenstein in der Optimierung der Verkehrsflussregelung in dieser Region.

8.2. Ausblick

Ein weiterer Ausbau des *ATCCL-Frameworks* kann das Anwendungsgebiet der Sprache erweitern. Alle flugplanverarbeitenden Systeme können tendenziell vom Einsatz der Sprache profitieren. Durch das generische Konzept des Frameworks ist eine Erweiterung der Syntax und Semantik mit wenig Aufwand möglich, was eine schnelle Integration in zukünftige Projekte ermöglichen soll.

Die systematische Konfigurierbarkeit der *DFLOW-Komponente* macht eine Erweiterung der Verarbeitungslogik möglich. Als Vervollständigung des Automatisierungskonzepts der Abflugregelung ist eine Integration mit einem Departure Manager, zur Abhandlung des Mikromanagements auf Flughafenebene, geeignet.

Das Ziel dieser Systeme soll nicht die Vollautomatisierung des Luftverkehrs sein. Heutige Softwarelösungen sind geeignet für Problemstellungen mit großen, für Menschen unüberschaubaren Suchräumen – jedoch nicht für Probleme mit vielen äußeren und dynamischen Faktoren. Der Mensch hat die notwendigen Eigenschaften, sich flexibel neuen und unerwarteten Situationen anzupassen. Erst die Kombination beider Vorteile erlaubt eine effektive Bewältigung der Herausforderungen der modernen Flugsicherung.

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Literaturverzeichnis

[ALSU07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers - Principles, Techniques and Tools*. Addison Wesley, 2007.

[Com05] International Electrotechnical Commission. *IEC 61508 Functional safety of electrical/electronic/programmable electronic safety-related systems*. IEC, 1998/2005.

[DME99] E. Dubouchet, G. Mavoian, and E.Page. *PHARE - Advanced Tools Departure Manager*. EUROCONTROL / CENA, 1999.

[DS09] Charles Donnelly and Richard Stallman. *Bison – The Yacc-compatible Parser Generator*. Free Software Foundation, 2009.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman, 1995.

[Glo08] Tilman Gloetzner. *IEC 61508 Certification of a Code Generator*. 3rd IET International conference on System Safety, 2008.

[Joh] Stephen C. Johnson. *Yacc: Yet Another Compiler-Compiler*. The University of Texas At Austin *(Online)*.

[Klu07] Marek Kluzniak. *Departure Manager DMAN Frankfurt*. Der Flugleiter – Gewerkschaft der Flugsicherung GdF, April, 2007.

[LS] M. E. Lesk and E. Schmidt. *Lex – A Lexical Analyser Generator*. The University of Texas At Austin *(Online)*.

[May02] Rodney May. *Personal Competencies and the Requirements of IEC 61508*. Programmable Electronics and Safety Systems: Issues, Standards and Practical Aspects, 2002.

[Men04] Heinrich Mensen. *Moderne Flugsicherung - Organisation, Verfahren, Technik*. Springer, 2004.

[Nie] Tom Niemann. *A Compact Guide to Lex and Yacc*. epaperpress.com.

[Org96] International Civil Aviation Organization. *Procedures for Air Navigation Services: Rules of the Air and Air Traffic Services*. ICAO, 1996.

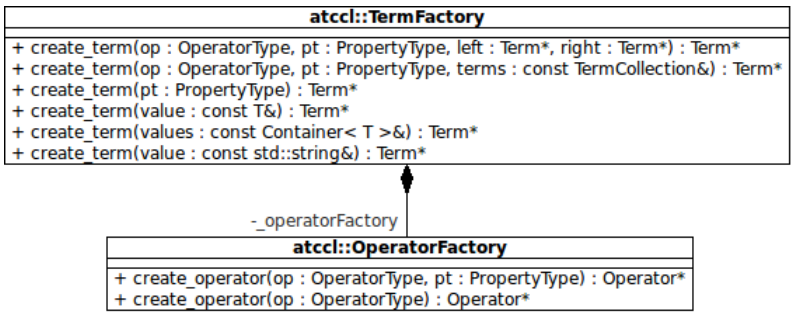
[PEM07] Vern Paxson, Will Estes, and John Millaway. *The flex Manual*. The Flex Project, 2007.

[Wir77] Niklaus Wirth. *Communications of the ACM*. ACM, 1977.

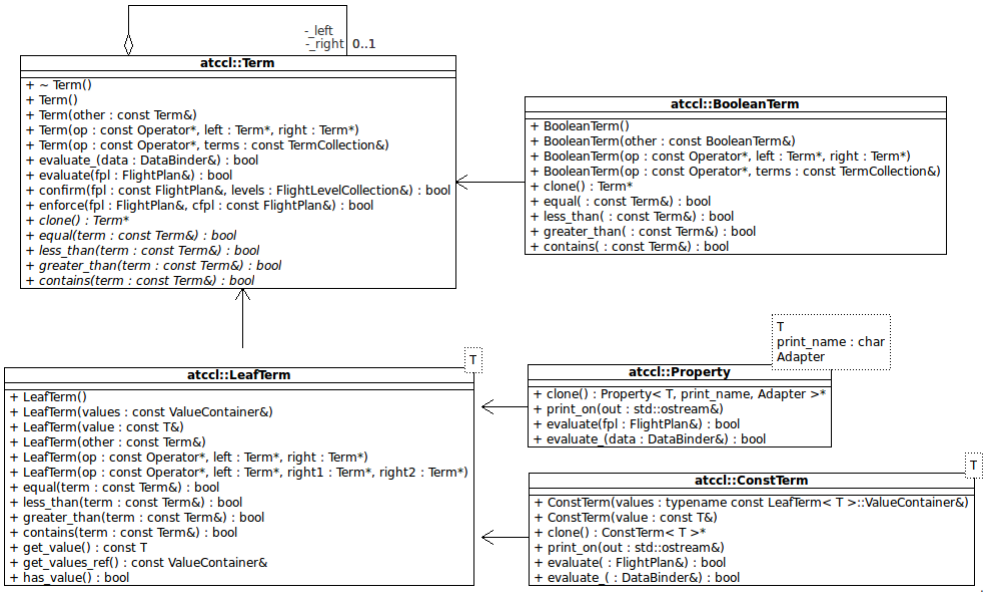
A. Feinentwurf

A.1. ATCCL

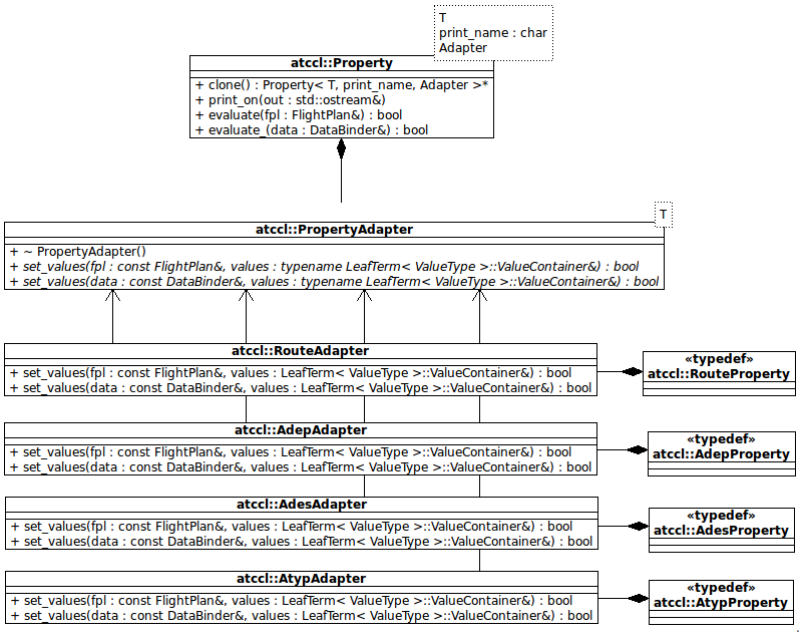
A.1.1. Factory



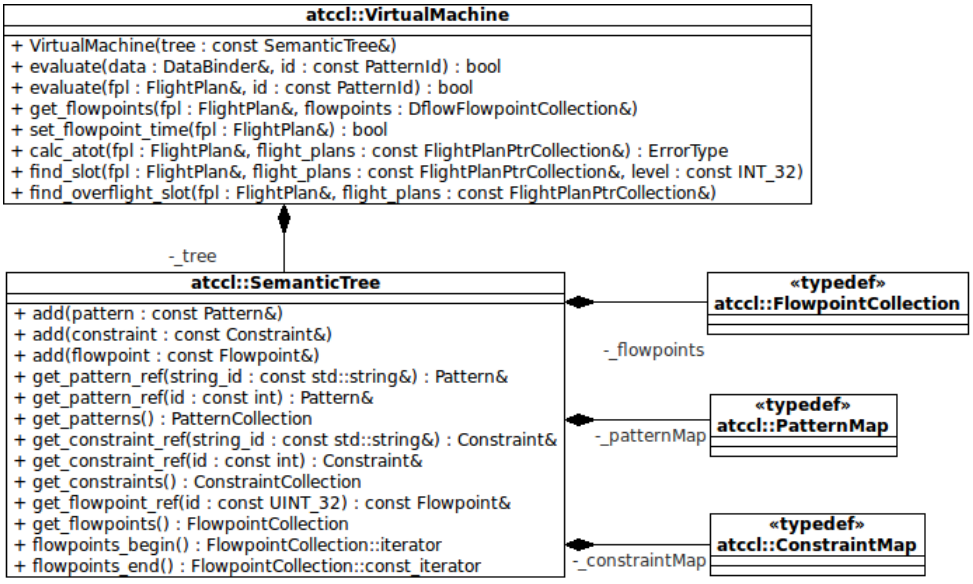
A.1.2. Term-Hierarchy



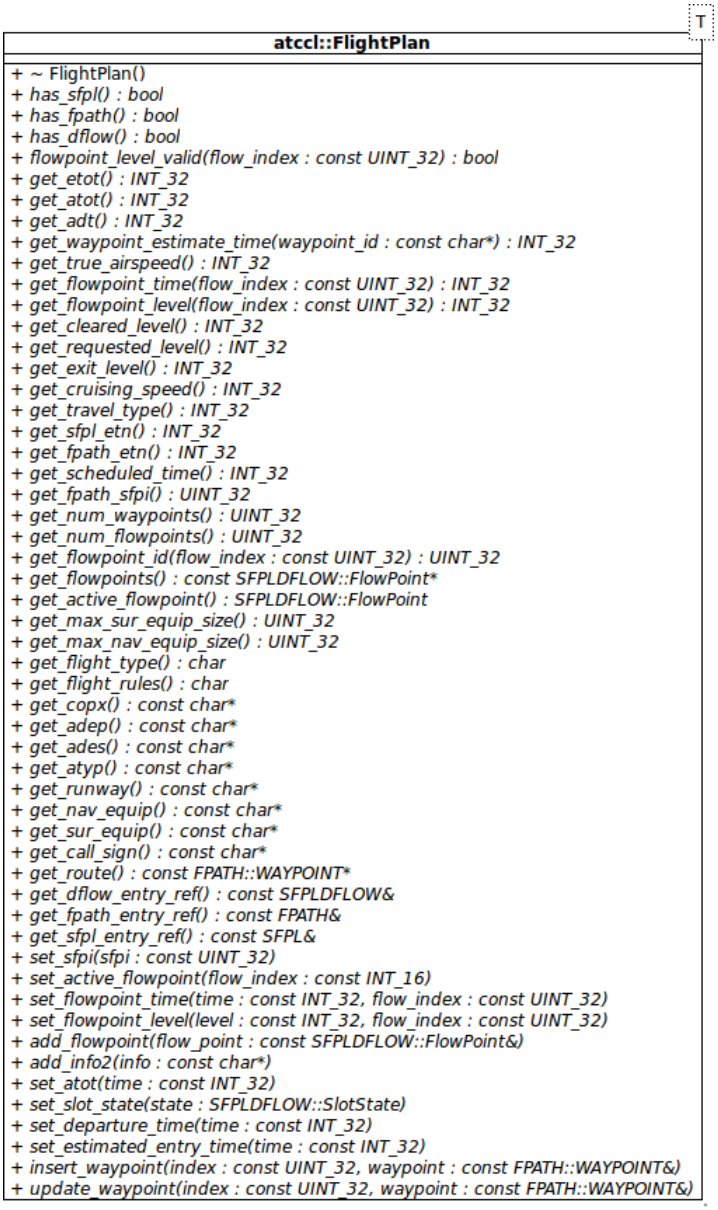
A.1.3. Property (Auszug)



A.1.4. Virtual Machine



A.1.5. Flight Plan Interface



B. bison-Konfiguration

```
%{
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "semantic_helper.hh"

#define MAX_ID_TABLE_NO 1024
char* id_table[MAX_ID_TABLE_NO];
int id_table_no;
ErrorType* errors;
int maxErrorNo;
int errorNo;
int yylex();
int yywrap();
int yyerror(const char *str);
void yyrestart(FILE* file);
%}
%token
CONSTRNT PATTERN FLOWP COM
QUOTE THAN FROM UNTIL ON

%union
{
        int integer;
        double real;
        const char* string;
}
```

```
%token <integer> INTEGER
%token <integer> TIME
%token <real>    REAL
%token <string> STRING
%token <integer> IS
%token <integer> IN
%token <integer> GREATER
%token <integer> LESS
%token <integer> NOT
%token <integer> AND
%token <integer> OR
%token <integer> AT
%token <integer> COPX
%token <integer> GATE
%token <integer> INTERNAL
%token <integer> GLOBAL
%token <integer> ADEP
%token <integer> ADES
%token <integer> RWY
%token <integer> ROUTE
%token <integer> ATYP
%token <integer> FLVL
%token <integer> TAS
%token <integer> FRUL
%token <integer> FTYP
%token <integer> EQUIP
%token <integer> RFL
%token <integer> TRAVEL_TYPE
%token <integer> TIMESEP
%token <integer> DEPTIME
%token <integer> ARRTIME
%token <integer> GATETIME
%token <integer> FL
%type <string> string
%type <string> pttrn
```

```
%type <string> flowp
%type <string> constrnt
%type <integer> int
%type <integer> string_array
%type <integer> int_array
%type <integer> int_pt
%type <integer> time_pt
%type <integer> char_pt
%type <integer> string_pt
%type <integer> string_array_pt
%type <integer> char_array_pt
%type <integer> int_op
%type <integer> string_op
%type <integer> string_array_op
%type <integer> string_string_array_op
%type <integer> si_unit

%left OR
%left AND
%right NOT
%%
prog: | prog rule {  }
      | error ')' { yyerrok; yyclearin;}
;
rule: pttrn      { if (!pattern($1))
                     printf("Error: %s\n", last_error()); }
      | constrnt { if (!constraint($1))
                     printf("Error: %s\n", last_error()); }
      | flowp    { if (!flowpoint($1))
                     printf("Error: %s\n", last_error()); }
      | COM      { comment(); }
;
flowp: FLOWP STRING '(' flowp_entry ')' { $$ = $2; }
;
flowp_entry: pttrn_lbl ':' constrnt_lbl
```

```
;
pttrn_lbl: STRING { if (!pattern_id($1))
                    printf("Error: %s\n", last_error()); }
;
constrnt_lbl: STRING { if (!constraint_id($1))
                       printf("Error: %s\n", last_error()); }
;
constrnt: CONSTRNT STRING '(' cterm ')' { $$ = $2; }
;
cterm:  TIMESEP AT string IS INTEGER si_unit
        { intpt_string_int_cterm(PT_TIMESEP, $3, $5 * $6,
                                 0, 0, 0); }
        | TIMESEP AT string_array IS INTEGER si_unit
        { intpt_stringarray_int_cterm(PT_TIMESEP, $3, $5 * $6,
                                      0, 0, 0); }
        | TIMESEP AT string IS INTEGER si_unit
          FROM INTEGER UNTIL INTEGER
        { intpt_string_int_cterm(PT_TIMESEP, $3, $5 * $6, $8,
                                 $10, 0); }
        | TIMESEP AT string_array IS INTEGER si_unit
          FROM INTEGER UNTIL INTEGER
        { intpt_stringarray_int_cterm(PT_TIMESEP, $3, $5 * $6,
                                      $8, $10, 0); }
        | TIMESEP AT string IS INTEGER si_unit AT FL int_array
        { intpt_string_int_cterm(PT_TIMESEP, $3, $5 * $6,
                                 0, 0, $9); }
        | TIMESEP AT string_array IS INTEGER si_unit
          AT FL int_array
        { intpt_stringarray_int_cterm(PT_TIMESEP, $3, $5 * $6,
                                      0, 0, $9); }
        | TIMESEP AT string IS INTEGER si_unit
            FROM INTEGER UNTIL INTEGER AT FL int_array
        { intpt_string_int_cterm(PT_TIMESEP, $3, $5 * $6, $8,
                                 $10, $13); }
        | TIMESEP AT string_array IS INTEGER si_unit
```

```
            FROM INTEGER UNTIL INTEGER AT FL int_array
        { intpt_stringarray_int_cterm(PT_TIMESEP, $3, $5 * $6,
                                      $8, $10, $13); }
        | cterm AND cterm { and_term(); }
;
si_unit: STRING { if (!strcmp($1, "s"))        { $$ = 1; }
                  else if (!strcmp($1, "min")) { $$ = 60; } }
;
pttrn: PATTERN STRING '(' term ')' { $$ = $2; }
       | PATTERN '(' term ')'      { $$ = get_pattern_id(); }
;
term:   int_pt int_op INTEGER
        { intpt_int_term($2, $1, $3); }
        | time_pt int_op INTEGER
        { time_pt_int_term($2, $1, $3); }
        | char_pt string_op string
        { charpt_char_term($2, $1, $3); }
        | string_pt string_op string
        { stringpt_string_term($2, $1, $3); }
        | string_array_pt string_array_op string_array
        { stringarraypt_stringarray_term($2, $1, $3); }
        | string_array string_array_op string_array_pt
        { if ($2 == OT_IN_C)          { $2 = OT_IN_P; }
          else if ($2 == OT_NOTIN_C) { $2 = OT_NOTIN_P; }
          stringarraypt_stringarray_term($2, $3, $1); }
        | string string_string_array_op string_array_pt
        { if ($2 == OT_IN_C)          { $2 = OT_IN_P; }
          else if ($2 == OT_NOTIN_C) { $2 = OT_NOTIN_P; }
          stringarraypt_string_term($2, $3, $1); }
        | string string_string_array_op char_array_pt
        { if ($2 == OT_IN_C)          { $2 = OT_IN_P; }
          else if ($2 == OT_NOTIN_C) { $2 = OT_NOTIN_P; }
          chararraypt_char_term($2, $3, $1); }
        | term OR term  { or_term(); }
        | term AND term { and_term(); }
```

```
        | NOT term
        | '(' term ')'
;
int_op: IS              { $$ = OT_IS; }
        | IS NOT            { $$ = OT_ISNOT; }
        | GREATER THAN { $$ = OT_GREATER; }
        | LESS THAN         { $$ = OT_LESS; }
;
string_op: IS        { $$ = OT_IS; }
          | IS NOT { $$ = OT_ISNOT; }
;
string_array_op: IS        { $$ = OT_IS; }
                | IS NOT { $$ = OT_ISNOT; }
                | IN      { $$ = OT_IN_P; }
                | NOT IN { $$ = OT_NOTIN_P; }
;
string_string_array_op: IN        { $$ = OT_IN_P; }
                      | NOT IN { $$ = OT_NOTIN_P; }
;
int_array: '[' ints ']' { $$ = integer_array(); }
;
ints: | ints ',' int
      | ints int
;
int: INTEGER { integer($1); $$ = $1; }
;
string_array: '[' strings ']' { $$ = string_array(); }
;
strings: | strings ',' string
         | strings string
;
int_pt: TAS         { $$ = PT_TAS; }
        | INTERNAL { $$ = PT_INTERNAL; }
        | GLOBAL    { $$ = PT_GLOBAL; }
;
```

```
time_pt: DEPTIME     { $$ = PT_DEPTIME;   }
       | ARRTIME     { $$ = PT_ARRTIME;   }
       | GATETIME    { $$ = PT_GATETIME;  }
;
char_pt: FTYP { $$ = PT_FTYP; }
;
string_pt:  ADEP           { $$ = PT_ADEP; }
          | ADES           { $$ = PT_ADES; }
          | RWY            { $$ = PT_RWY; }
          | ATYP           { $$ = PT_ATYP; }
          | COPX           { $$ = PT_COPX; }
          | GATE           { $$ = PT_GATE; }
          | FRUL           { $$ = PT_FRUL; }
          | RFL            { $$ = PT_RFL; }
          | TRAVEL_TYPE    { $$ = PT_TRAVEL_TYPE; }
;
string_array_pt: ROUTE { $$ = PT_ROUTE; }
;
char_array_pt: EQUIP { $$ = PT_EQUIP; }
;
string: QUOTE STRING QUOTE { string($2); $$ = $2; }
;
%%
#include "atccl_parser.c"
void clear_id_table()
{
    int i;
    for (i = 0; i < id_table_no; ++i)
    {
        free(id_table[i]);
        id_table[i] = 0;
    }
    id_table_no = 0;
}
int yywrap()
```

```
{
    clear_id_table();
    return 1;
}
int semantic_error(const char* str)
{
    if (errorNo < maxErrorNo)
    {
        errors[errorNo].line = -1;
        strcpy(errors[errorNo].text, "");
        strcpy(errors[errorNo].error, str);
        errorNo++;
    }
    return 1;
}
int yyerror(const char *str)
{
    if (errorNo < maxErrorNo)
    {
        errors[errorNo].line = line_no;
        strcpy(errors[errorNo].text, yytext);
        strcpy(errors[errorNo].error, str);
        errorNo++;
    }
    return 1;
}
int parse_syntax(FILE* file, ErrorType* errors_,
                 int maxErrorNo_, int passes)
{
    line_no = 1;
    errors = errors_;
    maxErrorNo = maxErrorNo_;
    errorNo = 0;
    clear_all();
    clear_id_table();
```

```
    yyrestart(file);
    yyparse();

    if (passes > 1)
    {
        if (!complete_semantics())
        {
            semantic_error(last_error());
        }
    }
    return errorNo;
}
```

Listing B.1: ATCCL bison-Konfiguration