Karlsruher Institut für Technologie

Fakultät für Informatik

Musterbasierte Modelltransformation zur Erzeugung von parallelen Softwaremodellen

Diplomarbeit
von

Sergej Poimzew

Verantwortlicher Betreuer: Prof. Dr. Walter F. Tichy
Betreuender Mitarbeiter: Dipl.-Inform. Korbinian Molitorisz

Bearbeitungszeit: 01. November 2012 – 29. April 2013

Karlsruher Institut für Technologie · Institut für Programmstrukturen und Datenorganisation ·
Geb. 50.34 · Am Fasanengarten 5 · 76131 Karlsruhe

Ehrenwörtliche Erklärung

Ich erkläre hiermit, die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben.

Karlsruhe, den 29. April 2013

Sergej Poimzew

Inhaltsverzeichnis

1 EINLEITUNG

1.1 Einführung in das Themengebiet

Moderne Softwareentwicklung ist eine Aufgabe, die sich in einen komplexen mehrstufigen Prozess aufteilt. In den einzelnen Phasen sind verschiedene Rollen eng miteinander verflochten. Über die letzten Jahre hat die Komplexität der Softwareentwicklung sogar noch enorm zugenommen und die Sprachen und Technologien entwickeln sich immer schneller und vielfältiger. Speziell die letzten fünf Jahre sind sehr stark geprägt durch das Aufkommen von Mehrkernprozessoren. Nun wird die Leistungssteigerung von alltäglichen Anwendungen durch Parallelisierung erreicht. Hierfür kamen neue Entwicklungswerkzeuge und Sprachkonstrukte hinzu, die das Leben der Softwareentwickler leichter machen sollten. Wie [Pan08] zeigt, ist dies allerdings nicht so einfach: Die Entwickler besitzen oft nicht die nötige Erfahrung und das Fachwissen, um diese neuen Möglichkeiten ausnutzen zu können. Aus Sicht der Forschung ist die automatisierte Überführung von Bestandssoftware, die nicht für Mehrkernprozessoren ausgelegt ist, auf Mehrkernprozessoren ein aktives Forschungsgebiet, das nach wie vor nicht erschöpfend angegangen wurde (siehe [Rus07], [Tou09], [MS+12]).

Das Parallelisieren von bestehenden Anwendungen kostet Zeit und Geld und sollte damit wohl überlegt sein. Viele wissenschaftliche Arbeiten beschäftigen sich mit der Parallelisierung auf der Code-Ebene - [Tou09], [Tou10], [Hot11], [MS+12]. Dabei muss der sequenzielle Code natürlich bereits vorhanden sein. Problematisch wird es auch für die neu entwickelnde Software, die von Anfang an parallel sein soll. Hier kann der modellbasierte Ansatz verwendet werden (engl. *model driven architecture*, *MDA*). Eine Bewertung des Parallelisierungspotenzials auf Modellebene soll die spätere Umsetzung erleichtern und dabei helfen, mögliche Fehler wie zum Beispiel wegen Datenabhängigkeiten zu vermeiden.

Ein weiterer Ansatz ist, die Parallelität explizit vom Programmierer ausweisen zu lassen. Es existiert eine Reihe von Programmierschnittstellen wie OpenMP und MPI, mit denen bestehende Programmiersprachen sich entsprechend erweitern lassen. Auch trotz der großen Anzahl an unterstützenden Werkzeugen und Entwurfsmustern ist das parallele Programmieren im Vergleich zum sequenziellen Programmieren deutlich komplexer. Auf ganzer Linie durchsetzen konnte sich noch kein Konzept und somit ist die Suche nach einer „besten“ Lösung zur parallelen Softwareentwicklung ein offenes Forschungsthema.

Viele Entwickler besitzen außerdem noch nicht genug Erfahrung und Fachwissen, um diese neuen Möglichkeiten voll ausnutzen zu können. Deswegen werden in der Wissenschaft automatische Parallelisierungstechniken entwickelt: [Rus07], [Tou10], [MS+12]. Automatische Parallelisierung bedeutet, dass die für die Nutzung paralleler Rechner erforderlichen Programmtransformationen entweder im Compiler durchgeführt werden oder explizit im automatisch generierten Code definiert sind. Dabei müssen mögliche Datenabhängigkeiten und mögliche Wettlaufsituationen erkannt und beseitigt werden. Die Ermittlung präziser Abhängigkeitsinformationen ist somit das zentrale Problem der automatischen Parallelisierung.

Die Sicht von Softwareunternehmen hingegen besitzt ganz andere Schwerpunkte in der Softwareentwicklung: Software soll möglichst schnell und kostengünstig entwickelt werden und trotzdem zuverlässig und effizient bleiben. Dabei soll die parallele Software zu vergleichbaren Kosten und Qualität wie die sequenzielle Software erstellt werden. Die Anforderungen der Kunden ändern sich oft und deswegen soll der Entwicklungsprozess flexibel gestaltet sein, um möglichst unmittelbar auf Änderungen reagieren zu können. Dies ist ein Grund, warum heutzutage agile Softwaremethoden und modellgetriebene Softwareentwicklung immer mehr an Bedeutung gewinnen [SV+07]. Der Einsatz von agilen Methoden führt unter anderem dazu, dass verschiedene Programmteile separat voneinander entstehen und oft sogar durch verschiedene Teams geschrieben werden können. Eine gute Dokumentation und eine saubere

Implementierung sind dabei essenziell wichtig, um den Code zugleich wartbar und erweiterbar zu halten. Insbesondere inhärente Datenabhängigkeiten und komplexe Klassenhierarchien machen die Parallelisierung von betrieblichen Softwaresystemen besonders schwierig.

Ziel dieser Diplomarbeit ist es, ein Werkzeug zu entwickeln, das dabei helfen soll, das Parallelisierungspotenzial eines Programms bereits auf Modellebene schnell und einfach zu erkennen und auszunutzen. Dafür werden unterschiedliche UML-Modelle (Klassen-, Aktivitäts- und Sequenzdiagrammen) gemeinschaftlich betrachtet und neben dem Kontrollfluss auch die Datenflüsse und Abhängigkeiten im Programm analysiert. Daraus sollen Parallelisierungskandidaten erkannt und durch entsprechende parallele Muster ersetzt werden.

1.2 In der Arbeit behandelte Fragestellungen

Diese Diplomarbeit beschäftigt sich mit der Analyse von UML-Softwaremodellen mit dem Ziel das Parallelisierungspotenzial zu erkennen und für die Erzeugung von neuen parallelen Modellen zu nutzen. Ein solches Modell, in dem die Anwendung beschrieben wird, wird häufig in enger Zusammenarbeit zwischen einem Auftraggeber und dem für die Realisierung verantwortlichen Softwareentwicklungsteam angefertigt. In der Regel entscheidet allein der Entwickler, wie dieses Modell im Code umgesetzt wird. Vielleicht kann er erkennen, dass zwei Fäden parallel ablaufen können, vielleicht aber nicht. Ein Werkzeug, das solche Fälle automatisch erkennt, ist natürlich sehr hilfreich, besonderes in großen Projekten.

Ein zweites Problem bei der Parallelisierung von Software ist die Erkennung von Datenabhängigkeiten. Die Abhängigkeiten zu finden ist nicht trivial, aber wichtig, um erfolgreiche Parallelisierung durchführen zu können. Hier sind auch verschiedene Ansätze möglich, die auf unterschiedlichen Ebenen passieren. Für eine neue Software ist es besonders wichtig, die kritischen Abschnitte zu erkennen, bevor man den Code schreibt. Wenn diese Datenstrukturen in Form von Klassen und deren Attributen in UML vorliegen und man, den Kontroll- und Datenfluss formal beschreiben kann, ist die Analyse von Datenabhängigkeiten einfacher und kann automatisch durchgeführt werden.

Unternehmerische Software wird oft in Drei-Schichten-Architektur konzipiert: Daten, Businesslogik und Darstellungsschicht (MVC, PAC Pattern) [Fow02]. Dabei werden die Daten (Datenbank, Webservices oder Dateien) oft vorgegeben und die Darstellung (Website, GUI) wird durch Designer oder frontend-Entwickler festgelegt. Diese Arbeit wird sich auf die Businesslogikebene konzentrieren, die das Modell und die View miteinander verbindet. Für die Businesslogik existieren bereits viele verschiedene parallele Architekturmuster.

Die in dieser Arbeit entwickelte Analysemethode soll ermöglichen, die Muster mit einem Parallelisierungspotenzial zu finden und die vorliegenden Modelle durch parallele Entwurfsmuster zu parallelisieren. Dabei werden drei Arten von UML-Modellen verwendet und transformiert: Klassendiagramme, Aktivitätsdiagramme und Sequenzdiagramme. Jedes Muster wird als ein Regelsatz in einem Graphersetzungssystem GrGen.NET beschrieben. Im Rahmen dieser Arbeit wird ein Musterkatalog mit einigen Grundmustern entwickelt und evaluiert. Weitere spezielle Muster sowie dazu passende parallele Varianten können die Entwickler später selbst hinzufügen.

Das Hauptziel dieser Arbeit ist es also, einen Ansatz für die musterbasierte Parallelisierung von UML-Modellen durch eine kombinierte Analyse des Kontroll- und Datenflusses zu entwickeln.

Um die beschriebenen Probleme zu lösen, wird in dieser Arbeit ein Konzept entwickelt, prototypisch umgesetzt und evaluiert. Die vorgeschlagene Analysetechnik ist musterbasiert und im Rahmen dieser Arbeit wird dafür ein Musterkatalog aufgestellt werden. Falls ein Muster erkannt wird, kann es durch ein paralleles Muster ersetzt werden. Das ist aber nur dann möglich,

wenn keine Datenabhängigkeiten vorliegen und die Parallelisierung überhaupt sinnvoll ist. Um Datenflüsse besser analysieren zu können und mögliche Datenabhängigkeiten zu finden, werden verschiedene UML-Diagramme zusammen betrachtet. So beschreiben die Aktivitätsdiagramme die Daten- und Kontrollflüsse. Sie liefern keine Informationen über Struktur der Daten und Ausführungsreihenfolgen innerhalb einer Aktivität. Dafür werden die Aktivitätsdiagramme mit Klassen- und Sequenzdiagrammen in einem abstrakten Objektmodell (AOM) kombiniert.

Diese Arbeit wird im Rahmen der .NET-Multicore-Gruppe unter der Leitung von Dipl.-Inform. Korbinian Molitorisz am Institut für Programmstrukturen und Datenorganisation, Lehrstuhl Prof. Dr. Walter F. Tichy und im Zusammenarbeit mit der Firma EXXETA AG geschrieben.

1.3 Gliederung der Arbeit

Im zweiten Kapitel werden die Grundlagen der modellgetriebenen Softwareentwicklung mit UML beschrieben. Es werden einzelne UML-Modelle und deren Elemente ausführlich beschrieben und entsprechende Analysen vorgestellt. Die Modelle dienen als Grundlage für automatische Parallelisierung und Codegenerierung, dazu werden Ansätze zur Parallelisierung auf Modellebene und für Codeerzeugung vorgestellt.

Im darauffolgenden Kapitel 3 wird ein Überblick über verwandte Arbeiten gegeben. Es werden vier Arbeiten vorgestellt, die sich mit der Modellierung von Parallelität und Datenflussanalyse in UML beschäftigen. Die zwei weiteren Arbeiten beschäftigen sich mit der automatischen Parallelisierung auf Code-Ebene. Abschließend werden die wichtigsten Merkmale von allen Arbeiten in eine Tabelle miteinander verglichen und bewertet. In diesem Kapitel werden außerdem die Gemeinsamkeiten und die Unterschiede zum eigenen Ansatz deutlich gemacht.

In Kapitel 4 werden zuerst die Anforderungen und Voraussetzungen dieser Arbeit definiert. Es wird das Konzept der Suchmuster vorgestellt. Diese Suchmuster bilden einen Katalog von Such- und Ersetzungsregeln, mit deren Hilfe diese Suchmuster in parallele Muster transformiert werden können. Kapitel 5 stellt die Implementierung des Ansatzes mit GrGen.NET dar. Dabei wird gezeigt, wie die in Kapitel 4 definierten Suchmuster umgesetzt und angewendet werden können. Abschließend werden im letzten Kapitel die Ergebnisse evaluiert und zusammengefasst.

Im Anhang befinden sich die in dieser Arbeit verwendeten Abkürzungen, das Abbildungsverzeichnis und sowie das Literaturverzeichnis.

2 GRUNDLAGEN

In diesem Kapitel werden die Grundlagen für modellgetriebene Softwareentwicklung mit *Unified Modeling Language* (UML) gegeben 2.2. Es werden einzelne Modelle und Diagramme beschrieben, sowie entsprechende Analyse- und Bewertungsmöglichkeiten auf Modellebene. Danach werden die Grundlagen der entwurfsmusterbasierenden Parallelisierung, sowie die Methoden für die automatischen Parallelisierungsverfahren vorgestellt 2.3.

2.1 Einführendes Beispiel: Online-Plattform zur Verwaltung von Aufträgen

In Abbildung 1 ist folgendes Szenario geschildert: Für die Verwaltung von Online-Aufträgen wurde eine Software, die dieses Bestellprozess abdeckt. Dazu wird über die Webseite ein Auftrag als Formular erstellt und die Kundendaten aus der Datenbank geladen. Danach erfolgt ein mehrstufiger Bearbeitungsprozess. Zum Schluss wird die Bestellung beendet und die Bestellung im Archiv gespeichert.

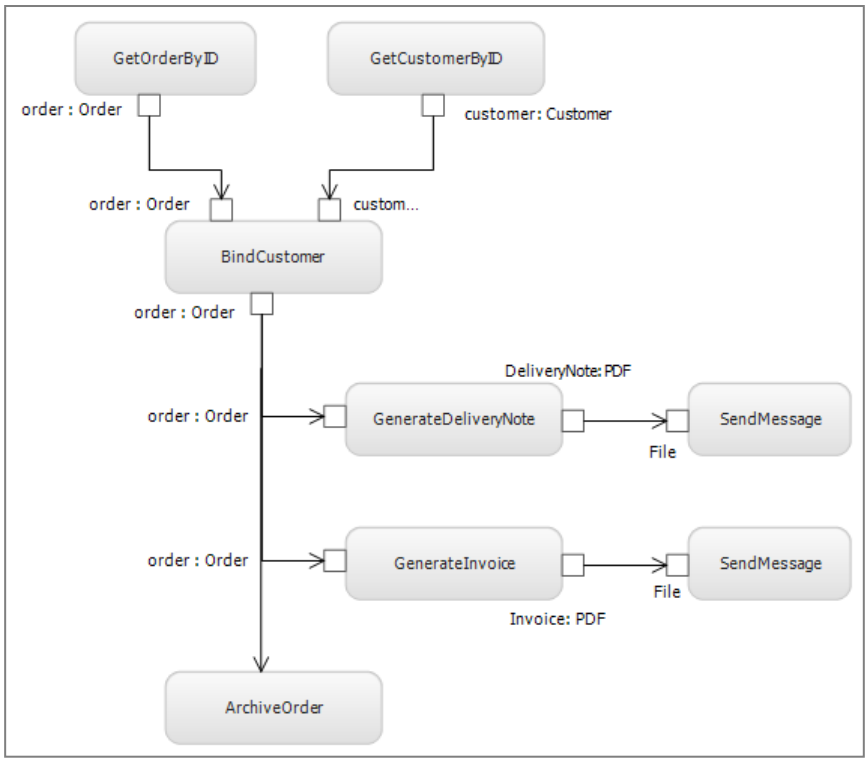


Abbildung 1: Beispiel zur Bewertung des Parallelisierungspotenzials

Dieses Diagramm stammt von einem Softwarearchitekten und wurde in Zusammenarbeit mit dem Auftraggeber entworfen. Es stellt den Bestellvorgang bildlich dar. Ein Softwareentwickler kann daraus nicht immer die parallele Datenflüsse und Datenabhängigkeiten erkennen, weil das gewählte Abstraktionsniveau zu hoch ist. Eine Schlussfolgerung über mögliche Parallelisierung ist auf dieser Ebene nicht möglich. Wenn es nun um die Parallelisierung in großen Projekten geht, hat man in aller Regel nun nicht mehr nur ein Diagramm vor sich, sondern eine große Zahl solcher Modelle zusammen mit bestehendem Code. Diese Menge an Modellen könnte nun in einer modellbasierten Analyse zusammen mit den bestehenden Quellcodeartefakten verwendet werden, um das Parallelisierungspotenzial des bestehenden Softwaregerüsts zu bewerten.

In [Stö05] ist beschrieben, dass Klassen- und Sequenzdiagramme zusammen betrachtet werden können um Rückschlüsse auf Datenflüsse zu machen. Auf diese Weise würde mehr Information über die real existierenden Datenflüsse und Klassenhierarchien im Programm gewonnen und es könnte so die Stellen bestimmt werden, an denen parallelisiert werden könnte.

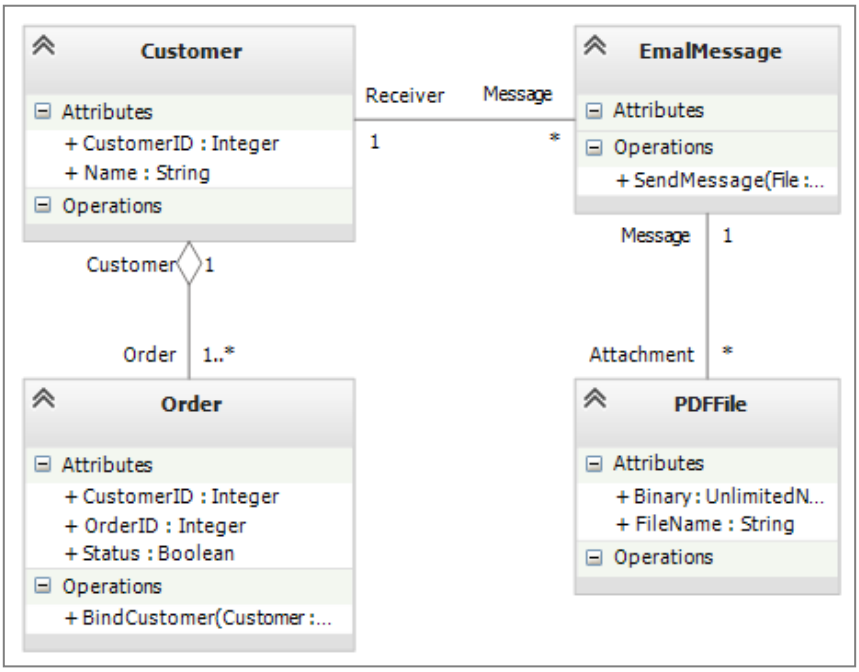


Abbildung 2: Klassendiagram für einführendes Beispiel

Abbildung 2 zeigt einen Ausschnitt aus dem Klassendiagramm für das obige Beispiel. Das Klassendiagramm stellt zusätzliche Informationen über die Klassenhierarchie und Assoziationen zwischen einzelnen Klassen dar. Durch diese Information kann das Aktivitätsdiagramm vervollständigt und der Datenfluss kann durch konkrete Methodenrümpfe beschrieben werden.

2.2 Modellgetriebene Softwareentwicklung mit UML

Die Modellierung hat eine lange Geschichte in allen Ingenieurdisziplinen und gewinnt immer mehr an Bedeutung in der Softwareindustrie. In der Softwareentwicklung können die gewählten Modelle einen großen Einfluss auf die Betrachtungsweise der Dinge haben.

Allgemein wird der Softwareentwicklungsprozess mit Sammlung und Analyse der Anforderungen begonnen, die in einer konkreten und informellen Weise beschrieben werden. Dabei werden natürliche oder andere informelle Sprachen verwendet. Die Anforderungen werden in einer Textform erfasst. Diese Sprachen sind schlecht strukturiert und formalisiert. Dadurch ist eine weitere maschinelle Verarbeitung nicht möglich. So würden die gesammelten Anforderungen durch eine manuelle Bearbeitung in eine formale Form umgewandelt. Dies kann natürlich leicht dazu führen, dass viele Missverständnisse, Missdeutungen und andere Fehler verursacht werden. Als Ergebnis steigen der Zeitaufwand und die damit verbundenen Kosten.

Im Gegensatz zu den informellen Sprachen stellt ein Modell dagegen eine vereinfachte Abbildung der Realität formal dar. In der modellgetriebene Softwareentwicklung spricht man von einer Domain (eine spezifische Umgebung, wie z.B. Bankwesen oder Medizin). Abbildung 3 zeigt einen hierarchischen Aufbau von Modellen in einer Domain. Ein Metamodell stellt dabei die Elemente einer Sprache und ihre Beziehungen zueinander dar und definiert somit die abstrakte Syntax dieser Sprache. Metamodelle werden wiederum durch ein Meta-Metamodell beschrieben.

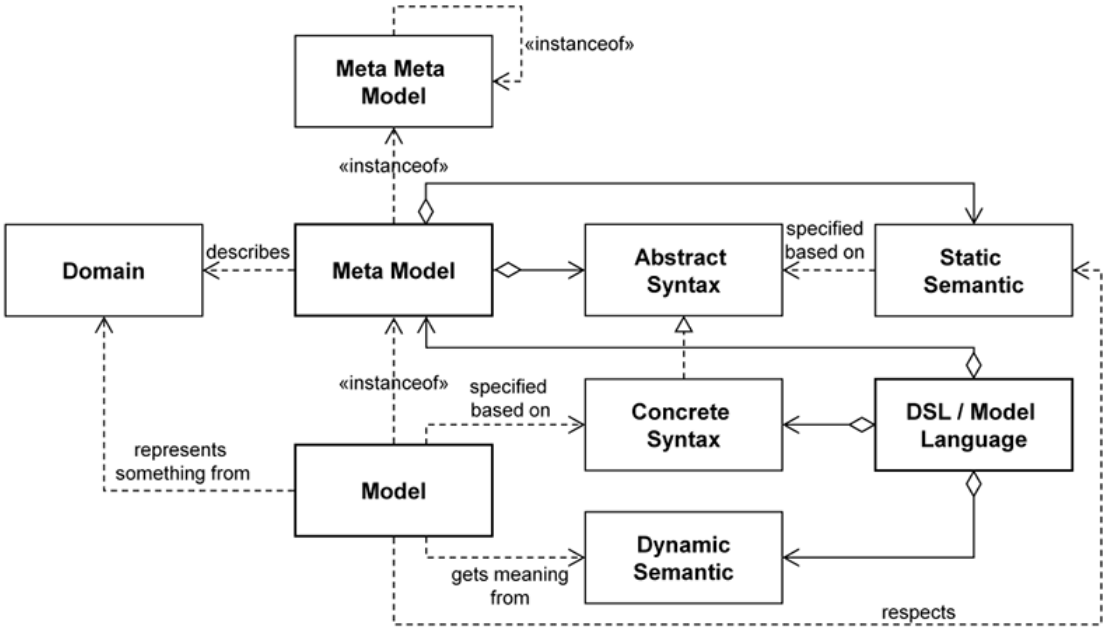


Abbildung 3: Drei Ebenen im modellgetriebenen Softwareentwicklung aus [Abe10]

Die modellgetriebene Softwareentwicklung kann in denselben Phasen wie traditionelle Entwicklung unterteilt werden [Abe10]. Es ist jedoch direkt von Anfang an, ein höherer Formalisierungsgrad erforderlich. Gewöhnlich ist das komplizierter und schwieriger als bei der Verwendung von natürlichen Sprachen. Eine formale Sprache oder ein Modell kann aber maschineninterpretiert und damit die Transformation zum Softwaredesign mindestens teilweise automatisiert werden kann. Dies führt zu einer besseren Softwarequalität, als durch eine manuelle Transformation. Deshalb helfen die Formalisierung und die Automation von Transformationen bessere und kostengünstigere Software zu entwickeln.

Ein Metamodell kann eine grafische oder eine textuelle Form haben und ist sozusagen die Benutzerschnittstelle der Sprache. In UML beschreibt das Metamodell verschiedene Diagrammtypen und ihre eigenen Bausteine.

2.2.1 UML-Grundlagen und Bausteine

In dieser Arbeit wird die aktuelle Version von UML 2.x verwendet, da es im Vergleich mit UML 1.x viele nützliche Änderungen und Erweiterungsmöglichkeiten hat [Stö05]. Die aktuelle Version definiert sechs Struktur- und sieben Verhaltensdiagramme. Die Spezifikation beschreibt diese Diagrammtypen nicht direkt. Sie ist in sogenannte Sprachelemente (engl. *language units*) unterteilt, die sich in drei große Gruppen aufteilen lassen.

Dinge (Elemente)

Dinge sind die wichtigsten Anteile eines Modells und stellen die Abstraktionen dar. Die Dinge bilden die objektorientierten Grundbausteine von UML. Dazu gehören Klassen und Schnittstellen (engl. *interfaces*) in Klassendiagrammen, sowie Zustände und Aktionen in Aktivitätsdiagrammen. Eine Gruppierung von mehreren Elementen gehört auch dazu.

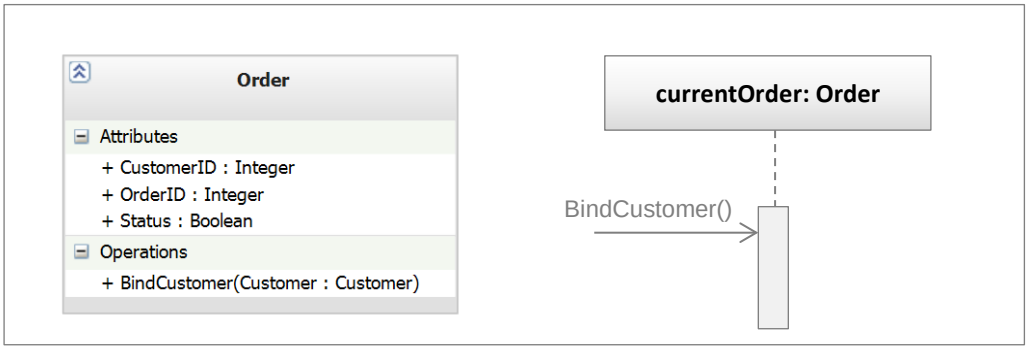


Abbildung 4: Beispiel für Dinge in UML - eine Klasse und eine *Lifeline*

Eine Klasse in UML kann wie in Abbildung 4 links dargestellt werden. Eine Klasse wird durch ihren Namen (hier *Order*) und die Menge von Attributen und dazugehörigen Methoden beschrieben. Die Methoden können auch Parameter und Rückgabetype haben, die durch andere Klassen im Modell oder primitiven Typen repräsentiert werden. Eine *Lifeline* ist ein benanntes Element, das einen einzelnen Teilnehmer (ein passives oder aktives Objekt) in der Interaktion darstellt. In diesem Beispiel sieht man eine Lebenslinie eines Objekts vom Type *Order*. So definieren die Klassendiagramme eine Hierarchie von Klassen und Typen, die in anderen Diagrammen verwendet werden.

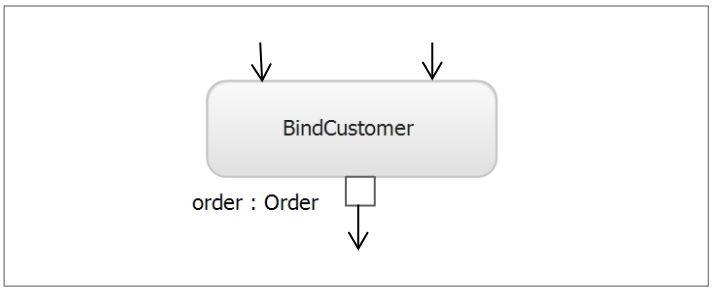


Abbildung 5: Beispiel für eine Aktion in UML

Die Methoden werden auch in anderen Diagrammtypen verwendet, wie zum Beispiel in Aktivitätsdiagrammen. Dort repräsentieren die Elemente einzelne Methoden und die Beziehungen zwischen ihnen. Abbildung 5 zeigt die Methode *BindCustomer* und dazugehörige Ein- und Ausgabeparameter mit der Typangabe. Durch eine Kombination von beiden Diagrammen kann erkannt werden, dass *order*-Objekt kein Eingabeparameter ist, sondern ein aktives Objekt (mehr dazu in Abschnitt 2.2.2), das die Methode *BindCustomer* ausführt und dadurch seinen Zustand ändert.

Beziehungen und Zugriffe

Die Beziehungen beschreiben die Abhängigkeiten zwischen einzelnen Elementen im Modell. Mit ihnen können wohlgeformte Modelle erzeugt werden. Es werden generell vier Arten von Beziehungen in Klassendiagrammen definiert- Abhängigkeit, Assoziation, Generalisierung und Realisierung. Wie in den folgenden Kapiteln zu sehen ist, spielen die Beziehungen eine wichtige Rolle bei der Analyse von UML-Diagrammen.

Die Analyse von Beziehungen zwischen Elementen ist ein wichtiger Bestandteil dieser Arbeit. Unterschiedliche Beziehungsarten beschreiben verschiedene Strukturen in Klassenhierarchien und Datenstrukturen und helfen zu verstehen, wie die Objekte im Programm bearbeitet werden. Zum Beispiel zeigt Abbildung 6 zwei Klassen in einer Beziehung 1 zu *. Dies bedeutet, dass ein *Customer* (Kunde) mehrere *Order* (Bestellungen) haben kann. Werden die Bestellungen in

einer Schleife bearbeitet, so kann diese Schleife parallelisiert werden, indem man die Bestellungen nach Kunden gruppiert und diesen Gruppen parallel bearbeitet.

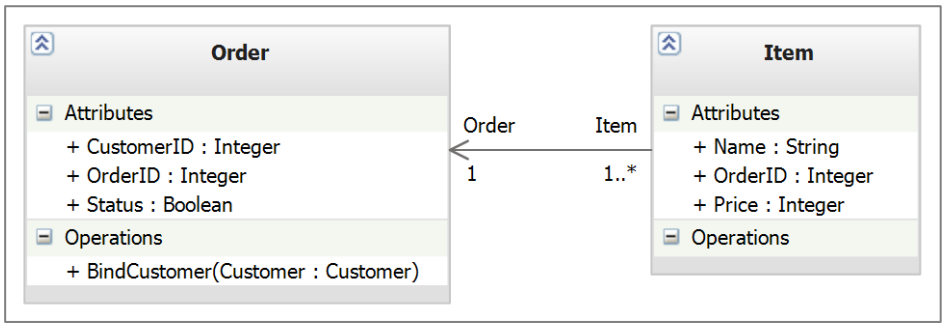


Abbildung 6: Beziehung zwischen zwei Klassen mit Kordialitäten

In den Verhaltensdiagrammen, wie Aktivitäts- oder Sequenzdiagrammen definieren die Kanten den Kontroll- und Datenfluss zwischen einzelnen Aktionen und Objekten. Mehr dazu in folgenden Abschnitten.

Diagramme

Ein Diagramm ist eine Menge von Elementen. In UML werden die Diagramme meistens grafisch als ein verbundener Graph von Knoten (Dingen) und Kanten (Beziehungen) dargestellt. Die unterschiedlichen Diagramme bieten eine Sicht von verschiedenen Perspektiven auf ein System und vervollständigen sich gegenseitig. Die in dieser Arbeit betrachteten Klassendiagramme definieren die statische Semantik und die Verhaltensdiagramme (Sequenz-, und Aktivitätsdiagramme) die dynamische Semantik eines Softwaremodells.

Ein Hauptkritikpunkt gegenüber der UML ist die fehlende formale Semantik. Das Spezifikationsdokument der [OMG] ist rein sprachlich und damit weder eindeutig noch widerspruchsfrei. Theoretisch kann ein Diagramm eine beliebige Kombination von Elementen enthalten. In der Praxis werden meistens nur die häufigsten Kombinationen verwendet.

2.2.2 Klassendiagramme und aktive Klassen

Die Klassen und Schnittstellen eines Programms sowie deren Beziehungen zueinander werden typischerweise in einem Klassendiagramm dargestellt. Das ist einer der am häufigsten verwendeten Diagrammtypen.

Steuerungsfluss

In einem sequenziellen Softwaresystem gibt es nur einen Steuerungsfluss. In jedem Zeitpunkt kann nur ein Ereignis passieren, indem ein Objekt seinen Zustand ändert. Wenn ein sequenzielles Programm ausgeführt wird, dann beginnt die Steuerung an dessen Anfang und die einzelnen Operationen werden nacheinander ausgeführt. In einem parallelen System existieren gleichzeitig mehrere Steuerungsflüsse und es können mehrere Ereignisse gleichzeitig passieren. Jeder dieser einzelnen parallelen Steuerungsflüsse wird in einem unabhängigem Prozess oder *Thread* ausgeführt.

In UML werden parallele Prozesse durch sogenannte **aktive Klassen** dargestellt. Dadurch wird ein Ausgangspunkt für einen unabhängigen und gleichzeitig neben anderen Steuerflüssen ablaufenden Fluss dargestellt. Aktive Klassen sind auch Klassen und haben alle dazugehörigen Eigenschaften wie Attribute und Methoden. Sie können Instanzen haben und von anderen Klassen erben und sie haben auch eine besondere Eigenschaft: Eine aktive Klasse steht für einen unabhängigen Steuerungsfluss und führt die Operationen durch. Die Klassen, die keine Steuerungsaktion auslösen, werden implizit passiv genannt. Technisch gesehen, repräsentiert eine Instanz einer aktiven Klasse einen Prozess oder Thread.

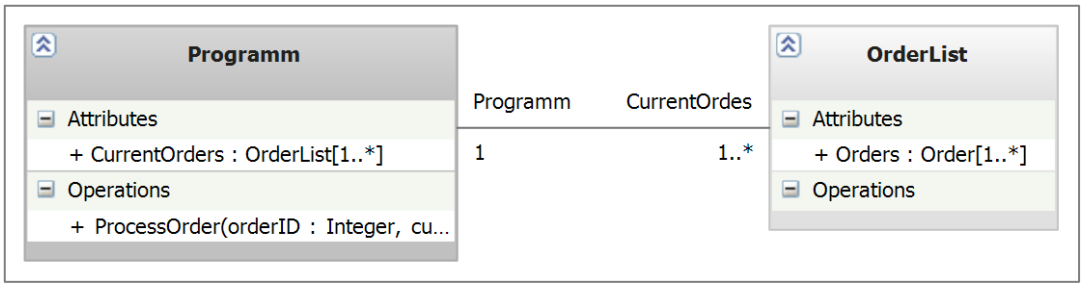


Abbildung 7: Aktive und Passive Klassen in UML

Abbildung 7 zeigt eine aktive Klasse *Programm* (links) und eine passive Klasse *OrderList* (rechts). Die Klasse *OrderList*, nimmt am Steuerungsfluss keine Rolle. Diese Klasse wird nur als passiver Datenspeicher im Programm verwendet.

Nebenläufigkeit und Synchronisierung

Ein System mit mehreren Steuerflüssen zu erstellen, ist ziemlich schwierig. Die Arbeit muss zwischen parallelen und aktiven Objekten aufgeteilt werden und es soll ein Kommunikationsmechanismus geben, um sicherzustellen, dass diese Objekte sich gegenseitig korrekt verhalten. Daher ist es für die Entwickler sehr hilfreich, sich bildlich vorzustellen, wie die einzelnen Flüsse miteinander in Wechselwirkung stehen. Das kann man realisieren, indem Klassendiagramme (statische Semantik) und Interaktionsdiagramme (dynamische Semantik) zusammenbetrachtet werden. Die Klassendiagramme definieren aktive und passive Klassen im Programm und beschreiben die Beziehungen zwischen einzelnen Klassen. Die Interaktionsdiagramme definieren die Reihenfolge der Aktionen im Programm.

2.2.3 Aktivitätsdiagramme und deren Analyse

Dieser Abschnitt führt in die Aktivitätsdiagramme ein. Die Aktivitätsdiagramme (AD) dienen zur Modellierung der dynamischen Aspekte eines Softwaresystems. Sie ähneln den Flussdiagrammen mit dem Unterschied, dass dort parallele Abläufe möglich sind. Im Grunde genommen stellt ein AD den Steuerungsfluss von Aktion zu Aktion dar.

Token-Begriff

In UML bilden die Token eine Grundlage für die Interpretation eines Aktivitätsdiagramms. Die Token kann man sich gedanklich als kleine Datenpakete vorstellen, die an den Aktivitätskanten entlang wandern und bei den Aktivitätsknoten etwas bewirken. In dieser Arbeit wird grundsätzlich zwischen Objekt-Token (Datenfluss) und Kontroll-Token (Kontrollfluss) unterschieden. Token besitzen kein Symbol in der UML.

Nach der Definition aus [UMLWiki] ist „ein Token (engl. für Zeichen, Marke) ein Hilfsmittel zur Synchronisation paralleler Prozesse – wer das Token hat, darf auf die Ressource (zum Beispiel einen Speicherbereich oder eine Schnittstelle) zugreifen. Wenn er das Token freigegeben hat, darf ein Konkurrent die Ressource benutzen."

Wächter

Ein Wächter beschreibt die Bedingung, unter der die Meldungen im Interaktionsoperanden (Kontrollknoten) ausgeführt werden. In dem Objektmodell sind die Wächter ein spezieller Aktivitätstyp, der am Anfang eines kombinierten Fragments (Alt, Opp, Loop) steht und die Bedienung angibt bzw. überprüft.

Datenfluss

Eine Aktivität ist eine fortlaufende, strukturierte Ausführung eines Teils des Prozesses oder Programms. Sie beschreibt einen zusammenhängenden Ablauf und besteht aus mehreren Aktionen [UMLWiki]. Mit einem AD lässt sich auch der Datenfluss zwischen einzelnen Schritten modellieren. Dabei sind auch Verzweigungen und Nebenläufigkeiten möglich. Die Aktionen innerhalb einer Aktivität bilden einen Programmzustand ab und können weitere Aktionen (Operationen) aufrufen, ein Signal senden, ein Objekt erstellen oder zerstören. Grafisch wird eine Aktion mithilfe eines abgerundeten Rechtecks dargestellt.

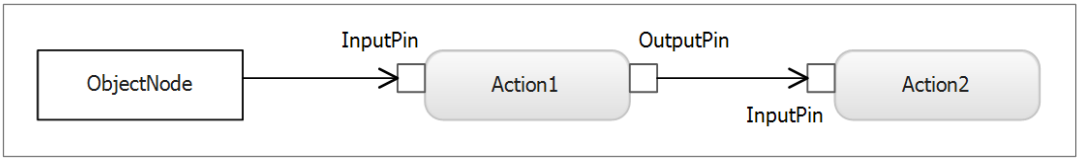


Abbildung 8: Aktivitätsdiagramm mit Datenfluss

Die Aktionen können, genau wie Methoden im Programm eigene Ein- und Ausgabeparameter haben. Diese werden in einem Aktivitätsdiagramm mithilfe von sogenannten Pins dargestellt. In Abbildung 8 ist links ein Objektknoten zu sehen. Der Datenfluss von diesem Objektknoten geht über die Pins (kleinen weißen Rechtecks) zur Aktionen weiter. Die Pins müssen nicht immer gezeigt werden. Sie eignen sich am besten, wenn gezeigt werden muss, welche Daten gebrauchen und welche von den verschiedenen Aktionen geliefert werden. Dies macht die Pins sehr nützlich für die Modellierung und Analyse verschiedenen Datenflüssen im Programm und somit auch hilfreich für die Bewertung des Parallelisierungspotenzials.

Ausdehnungsbereiche

In Aktivitätsdiagrammen treten oft Muster auf, in denen eine Aktion mehrere Aufrufe einer anderen Aktion auslöst, wie zum Beispiel in eine Schleife. Diese Situationen werden mit einem Ausdehnungsbereich (engl. *expansion region*) modelliert, in dem eine Aktion einmal pro Element einer Menge eintritt. Das obige Beispiel zeigt die Bearbeitung eines Auftrags. In Wirklichkeit werden mehrere Aufträge in eine Schleife bearbeitet. Dies kann auch parallel passieren, aber manche Operationen wie I/O können nur sequenziell ausgeführt werden. So werden die Daten zuerst in einem Puffer gespeichert und dann nacheinander sequenziell bearbeitet. Abbildung 9 illustriert diesen Prozess unter Verwendung eines Ausdehnungsbereichs.

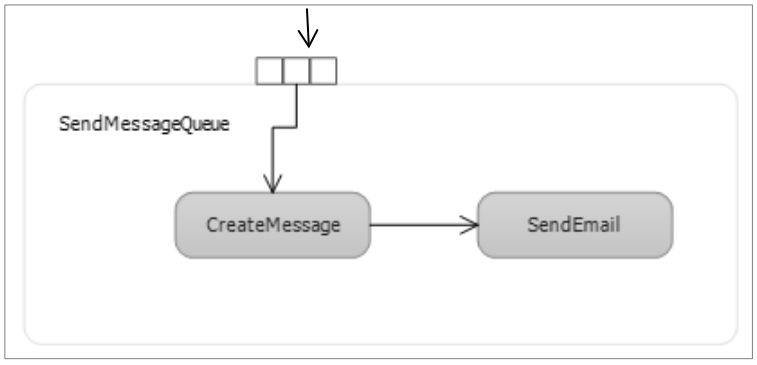


Abbildung 9: Ausdehnungsbereich im Aktivitätsdiagramm

Fork und *Join*

Für die Modellierung von nebenläufigen Flüssen in Aktivitätsdiagrammen bietet das UML einige spezielle Konstrukte. Die Änderungen in Aktivitätsdiagrammen werden hauptsächlich durch Einfügen von neuen Parallelisierungs- *(Fork)* und Zusammenführungsknoten *(Join)*.

Abbildung 10 zeigt ein *Fork-/Join*-Muster in einem Aktivitätsdiagramm. So kann der obere Teil des einführenden Beispiels (Abbildung 1) parallelisiert werden:

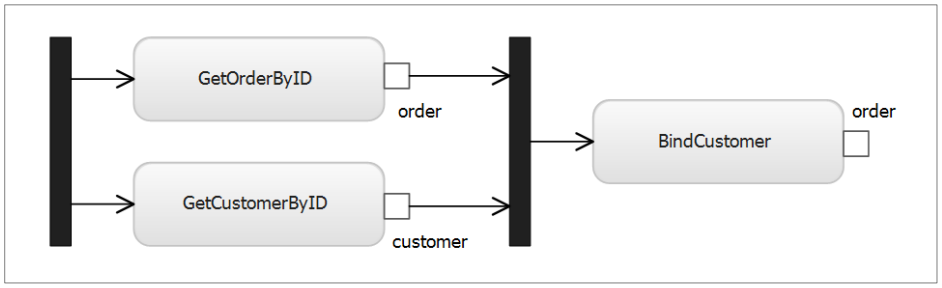


Abbildung 10: Fork-Join-Muster in Aktivitätsdiagramm

2.2.4 Weitere Modelle und Spracheinheiten

Ein Interaktionsdiagramm besteht aus einer Menge von Objekten und ihren Beziehungen. Ein Sequenzdiagramm ist ein Interaktionsdiagramm, das den zeitlichen Ablauf hervorhebt. Sie wurden ursprünglich zur Modellierung von Telekommunikationssystemen entwickelt (engl. *Message Sequence Charts*).

So können gleiche Sachverhalte in unterschiedlichen Diagrammen und bestimmte charakteristische Merkmale haben, die in anderen Diagrammen nicht erkennt werden. Folgende Abbildung zeigt die Erzeugung von zwei nebenläufigen Fäden in einem Programm. Gleiche Elemente in AD und SD sind farblich markiert. In diesem Beispiel wird eine Bestellung (*order*) bearbeitet und gleichzeitig eine Nachricht geschickt (*SendMessage*).

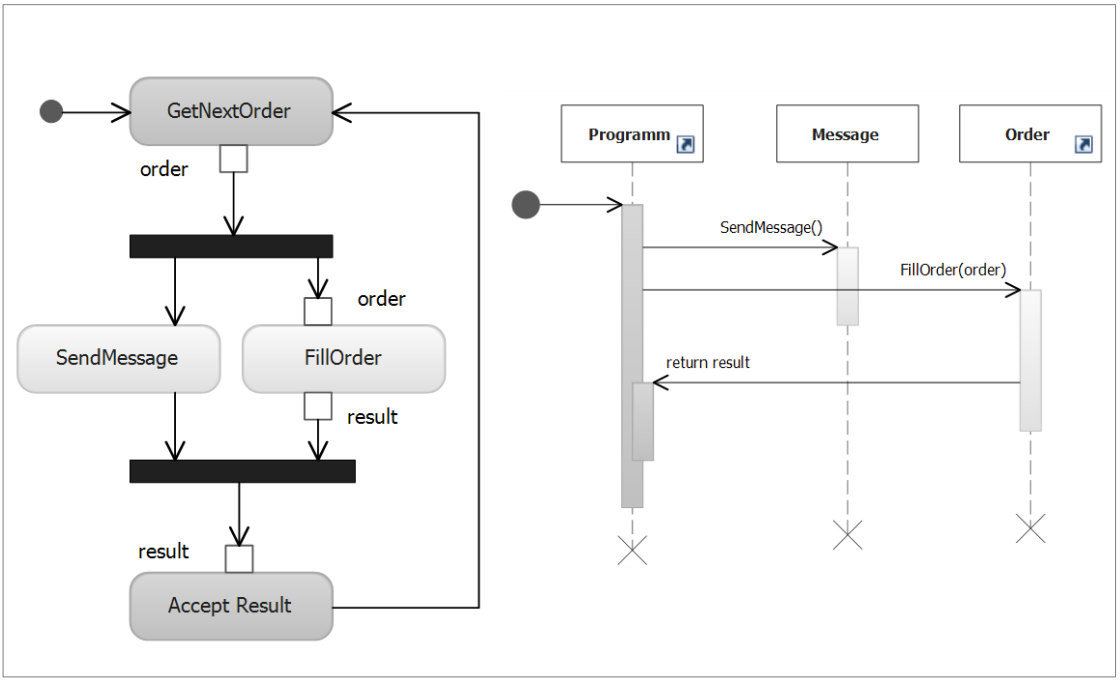


Abbildung 11: Nebenläufigkeit in Interaktionsdiagrammen

Sequenzdiagramme und UML-Kommunikationsdiagramme

Der UML 2-Standard bietet neben den Sequenzdiagrammen eine zusätzliche, den Sequenzdiagrammen inhaltlich eng verwandte Notation an. Kommunikationsdiagramme stellen eine Teilmenge der Informationen eines Sequenzdiagramms dar, fokussieren aber weniger auf

die zeitliche Reihenfolge, sondern mehr auf die Zusammenarbeit zwischen den Objekten. Die Reihenfolge der Interaktionen wird nicht durch Zeitlinien, sondern durch eine Nummerierung festgelegt. Dabei werden verschachtelte Aufrufe mit aufsteigenden Nummernlisten gekennzeichnet. Return-Werte werden beim Aufruf eingetragen. Ein zu den Aktivitätsbalken analoges Konzept existiert in Kommunikationsdiagrammen nicht. In dieser Arbeit wird dieses Konzept nicht verwendet. Er ist aber von Interesse für die zukünftigen Arbeiten. Mehr zu Kommunikationsdiagrammen kann in [RQ+07] nachgelesen werden.

2.3 Entwurfsmusterbasierende Parallelisierung

Die Softwareentwicklung für parallele Plattformen unterscheidet sich vom klassischen Softwareentwicklungsprozess. In diesem Kapitel werden die Grundlagen des parallelen Softwareentwicklung betrachtet, sowie der musterbasierte Ansatz zur Parallelisierung.

Die Definitionen und Begriffserklärungen in diesem Abschnitt basieren auf den Arbeiten von Dr. Viktor Pankratius - [Pan08], [Pan11].

2.3.1 Einführung in der Software-Parallelisierung

Parallele Programmierung bringt eine gewisse Komplexität mit sich. Wenn die Programme auf einem niedrigeren Niveau parallelisiert werden, erhöhen sich der Anzahl der Befehle drastisch und der Zusatzaufwand ist oft größer als durch die Parallelisierung gewonnene Leistung. Aus diesem Grund empfiehlt [Pan08], die Parallelisierung auf eine höhere Ebene durchzuführen und parallele Entwurfsmuster zu verwenden.

Generell werden folgende Parallelitätsarten unterschieden:

- **Parallelität auf Befehlsebene** (engl. *instruction level parallelism, ILP*).
  Ein Prozessor führt *N* Instruktionen gleichzeitig.

- **Datenparallelität** (engl. *data parallelism, DP*). Mehrere Datensätze werden auf *N* Prozessorkernen bearbeitet.

- **Parallelität auf Thread/Task-Ebene** (engl. *thread level parallelism, TLP*). *N* Fäden laufen parallel auf einem Rechnersystem.

Bei der Parallelisierung von bestehender Software sollte beachtet werden, dass ein Programm nie vollständig parallelisiert werden kann und in der Regel nur einzelne Abschnitte parallel ausgeführt werden können. Dies können parallele Schleifen oder Methodenaufrufe sein. Dabei hat man immer einen sequenziellen Teil und einen oder mehrere parallele Blöcke.

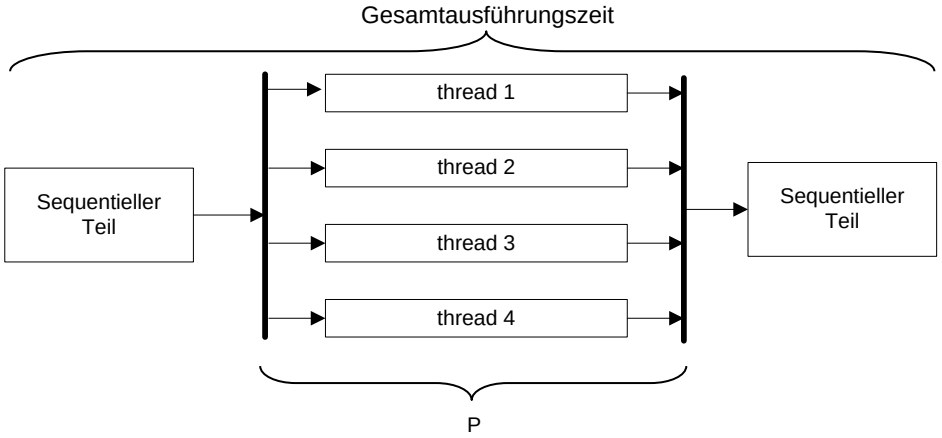


Abbildung 12: Sequenzielle und parallele Programmabschnitte

Um Kandidaten für eine parallele Ausführung zu identifizieren, wird ein Problem zunächst in einzelne Aufgaben zerlegt. Diesen Aufteilungsprozess nennt man auch Partitionierung. Anschließend wird geprüft, wie diese Aufgaben miteinander kommunizieren. Außerdem werden die Zugriffsmuster und Kommunikationsprotokolle festgelegt. Einzelne Aufgaben lassen sich oft noch gruppieren oder miteinander kombinieren.

Wie [Pan08] zeigt, wie der Parallelisierungsprozess in die vier folgenden Phasen unterteilt werden kann:

- **Partitionierung:** Eine logische Teilung einer Aufgabe in parallel ausführbare Methoden oder Fäden.
- **Kommunikation:** Festlegung der Kommunikation zwischen den Fäden. Es wird auch entschieden, wie einzelne Fäden auf gemeinsame Ressourcen zugreifen sollen und welche Schutzmechanismen dabei verwendet werden.
- **Agglomeration:** Zusammenfassung von Fäden zur Effizienzsteigerung.
- **Prozessorzuordnung:** Zuordnung der Fäden zu den Prozessoren. Dieser Schritt wird meistens durch die Ausführungsumgebung oder das Betriebssystem realisiert.

Diese ersten drei Phasen passen sehr gut für die musterbasierte Parallelisierung. In jede Phase existieren spezifische Muster, die gefunden und durch parallele Muster ersetzt werden können. Wie man später in Kapitel 4 sehen wird, werden auch in dieser Arbeit gleiche Phasen verwendet. Im *ObjectMerge* 4.5.1 und dem *ProducerConsumer* 4.5.4 Muster wird das Partitionierungsprinzip verwendet. Das *ActionsBunch* Muster nutzt die Agglomeration von mehreren Aktivitäten um unabhängige parallele Ausführungsfäden zu bilden.

2.3.2 Datenabhängigkeiten

Als Datenabhängigkeit $a_1 \overset{DataDep}{\longleftarrow} a_2$ (*engl. data dependency*) wird eine Situation bezeichnet, dass die Daten, die von einer Anweisung benutzt werden, auch von einer vorhergehenden Anweisung benutzt werden. Übertragen auf die UML-Modelle, kann man als Datenabhängigkeit ein Muster bezeichnen, indem zwei Aktivitäten a_1 und a_2 mit der Ausführungsreihenfolge a_1, a_2 auf den gleichen Objektknoten *obj* zugreifen und mindestens eine Aktion das Objekt verändert.

Generell wird zwischen drei Arten von Datenabhängigkeiten unterschieden:

Echte Datenabhängigkeit - RAW (engl. *read-after-write*):
$a_1 \overset{RAW}{\longleftarrow} a_2$: a_2 liest *obj*, nachdem a_1 *obj* verändert hat.

Gegenabhängigkeit - WAR (*engl. write-after-read*):
$a_1 \overset{WAR}{\longleftarrow} a_2$: a_2 verändert *obj*, nachdem a_1 *obj* gelesen hat.

Ausgabeabhängigkeit - WAW (engl. write-after-write)
$a_1 \overset{WAW}{\longleftarrow} a_2$: i_2 schreibt *obj*, nachdem a_1 *obj* geschrieben hat.

Im Gegensatz zur RAW-Abhängigkeit können WAR- und WAW-Abhängigkeiten durch die Verwendung neuer Ressourcen (Objekten) aufgelöst werden. Dies kann zum Beispiel durch Verwendung einer Pipeline möglich sein, in dem die a_1 und a_2 einzelnen Stufen repräsentieren. Der Fall a_2 liest *obj* nachdem a_1 *obj* gelesen hat, spielt für den weiteren Verlauf keine Rolle.

Kontrollabhängigkeiten

Eine Kontrollabhängigkeit $a_1 \overset{ContDep}{\longleftarrow} a_2$ (*engl. control dependency*) zwischen zwei Aktionen a_1 und a_2 mit der Ausführungsreihenfolge a_1, a_2 besteht, falls das Resultat von obj_1 darüber entscheidet, ob a_2 ausgeführt wird oder nicht. Diese Abhängigkeit kann man bei der Analyse von Kontrollknoten und damit verbundenen Datenflüssen finden.

2.3.3 Beschleunigung und Effizienz

Die Steigerung der Effizienz ist eine der wichtigsten Gründe für das Parallelisieren von Software und Algorithmen. Allerdings bedeutet die Erhöhung der Anzahl der Prozessorkerne auf eine bestimmte Zahl *N* nicht, dass ein paralleler Algorithmus *N* mal schneller läuft als eine serielle Variante. Dies liegt daran, dass sich die parallel ablaufenden Fäden nicht ausschließlich mit der Berechnung der Ergebnisse beschäftigen können, sondern auch miteinander interagieren oder aufeinander warten müssen.

Die **Beschleunigung** (eng. *speedup*) durch Parallelisierung wird als ***S(N)*** bezeichnet. Außerdem werden die Ausführungszeit eines sequenziellen Programms *T(1)* und die Ausführungszeit eines parallelen Programms auf *N* Prozessoren als *T(N)* bezeichnet.

$$\boldsymbol{S}(\boldsymbol{N}) = \frac{T(1)}{T(N)}$$

Formel 1: Beschleunigung durch Parallelisierung

Es ist zu beachten, dass *N* größer als die Anzahl der Fäden sein kann. So wächst *T(N)* nicht linear mit *N* zusammen, sondern ist nach [Poi11] treppenförmig. Der **Parallelitätsgrad** *P* eines Programms gibt an, wie viel Operationen parallel ausführbar sind. Ist $P \neq N$, so muss das S(N) anders berechnet werden.

Die **Effizienz E** (engl. *efficiency*) ist ein Wert, der typischerweise zwischen Null und Eins liegt und bestimmt, wie gut die Prozessoren bei der Lösung eines Problems ausgelastet sind. Der Effizienzwert zeigt auch, wie groß der Mehraufwand ist, der durch die Kommunikation und Synchronisation zwischen Prozessoren entsteht.

$$\boldsymbol{E}(\boldsymbol{N}) = \frac{S(N)}{N}$$

Formel 2: Effizienzwert bei der Parallelisierung

Die Effizienz ist auch von dem Verhältnis zwischen *P* und *N* abhängig. Optimale Beschleunigung und somit auch Effizienz wird erreicht, wenn *P* durch *N* restlos teilbar ist. Beim Effizienzbegriff wird zwischen drei möglichen Beziehungen der Beschleunigung und der Anzahl an Prozessoren unterschieden:

- Ist $S(N) < N$, spricht man von einer sublinearen Beschleunigung
- Ist $S(N) = N$, ist die Beschleunigung linear
- Ist $S(N) > N$, dann ist die Beschleunigung superlinear und *E* >100%

Arten der Parallelität

Diese Arbeit ist nicht auf einen Stil der parallelen Programmierung beschränkt, sondern es wird auf einem höheren Modellniveau gearbeitet. Die konkrete Implementierung und die Sprachen können somit durch den Entwickler gewählt werden. Die Architekturmuster helfen, die große Kontroll- und Datenflüsse zu bestimmen und optimal zu parallelisieren. Dabei spielen die drei wichtigen Parallelisierungskriterien eine Rolle:

Explizite und implizite Parallelität

Explizite Parallelität (EP) wird durch spezielle Sprachkonstrukte angegeben. Auf Modellebene sind es Verzweigungen und nebenläufige Aktivitäten sowie Ausdehnungsbereiche, wie in 2.2.3 beschrieben ist. Der Schwerpunkt dieser Arbeit ist aber die implizite Parallelität.

Die implizite Parallelität (IP) bedeutet, dass die Operationen nicht oder nur partiell geordnet sind. Implizite Parallelität wird auf der Codeebene oft genutzt, indem Compiler diese Situationen erkennt und der Code entsprechend automatisch parallelisiert (engl. *instruction level parallelism*, *ILP*). Eine weitere Möglichkeit, IP zu benutzen, ist die Instrumentierung von Code oder Modellen, sodass durch explizite Anweisungen gesagt wird, was parallel ausgeführt werden soll.

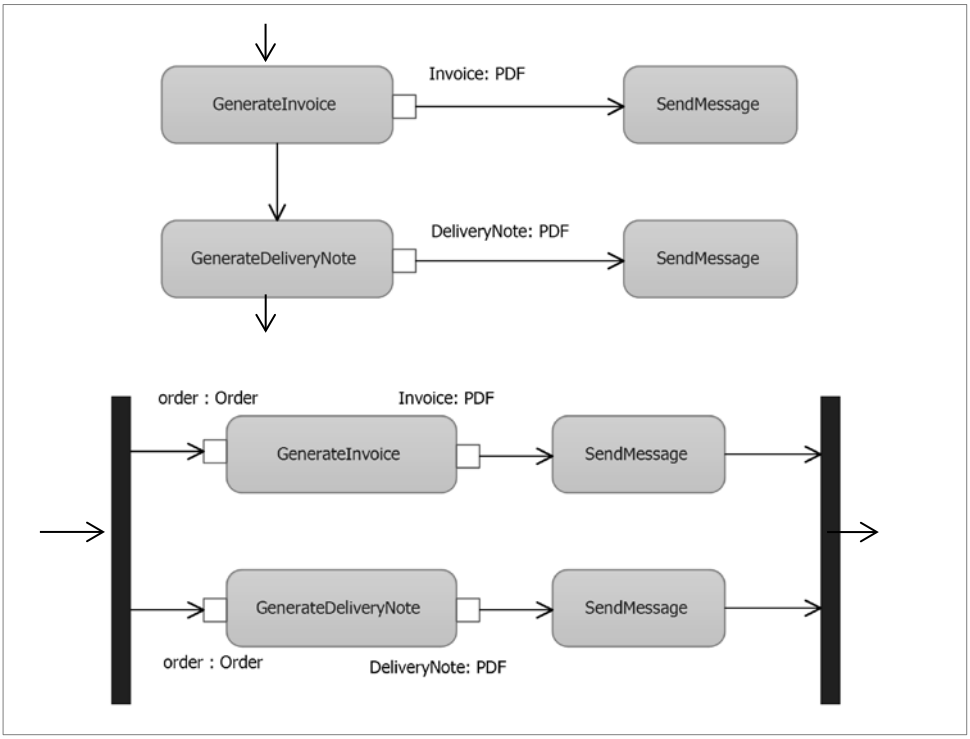


Abbildung 13: Explizite und implizite Parallelität in UML

Der in dieser Arbeit entwickelte Ansatz entscheidet selbstständig aus dem Daten- und Kotrollfluss, welche Operationen parallel ausgeführt werden können. So können auch sehr große Modelle auf implizite Parallelität untersucht und durch das Werkzeug automatisch parallelisiert werden.

Allgemein können in den Softwaremodellen folgende implizite Hinweise auf Parallelität gefunden werden:

- Kordialität der Klassen in einem Klassendiagramm gibt den Parallelitätsgrad an
- Ausdehnungsbereiche und Zyklen in Aktivitätsdiagrammen zeigen rekursive Gebiete an
- Partiell geordnete Aktivitäten mit direktem Datenfluss lassen sich in Äquivalenzklassen von parallel ausführbaren Anweisungen zerlegen (wie in Abb. 13)

Aufgaben- und Datenparallelität

Unter **Aufgabenparallelität** wird nach [Pankr11] die Verteilung und parallele Ausführung von Prozessen über verschiedene Prozessoren bzw. Prozessorkerne verstanden. Für die Modellierung von solchen Prozessen in UML werden hauptsächlich Aktivitätsdiagramme 2.2.3 und Sequenzdiagramme 2.2.4 verwendet.

Bei der Parallelisierung werden die Aufgaben definiert, die parallel ausgeführt werden müssen, und es wird versucht, diese möglichst voneinander unabhängig auszuführen. Auf dieselbe Weise funktioniert zum Beispiel ein Webserver, der für verschiedene Nutzer einzelne Fäden startet. Die voneinander unabhängigen Aufgaben können problemlos parallel ausgeführt werden und

sind somit sehr gute Kandidaten für eine Parallelisierung. Auch einzelne Schleifeniterationen, die nur mit lokalen Daten arbeiten, können leicht parallelisiert werden.

Deutlich schwieriger wird es, wenn parallele Aufgaben miteinander kommunizieren und eventuell aufeinander warten müssen. In diesem Fall werden spezielle Konstrukte für Kommunikation und Synchronisation zwischen einzelnen Fäden verwendet. Die Kommunikation (Datenfluss und Signale) kann durch die Analyse von Aktivitätsdiagrammen [2.2.3] extrahiert werden. Eine zusätzliche Analyse von Klassendiagrammen hilft dabei, die aktiven und passiven Klassen zu bestimmen [2.2.2], sowie Datenabhängigkeiten in eine Klassenhierarchie zu finden. So kann ein Signal nur von einer aktiven Klasse gesendet werden.

Bei der **Datenparallelität** geht es um das Aufteilen von Daten, sodass sie parallel bearbeitet werden könnten zum Beispiel bei einer Suche im Array. Einzelne Array-Teile können parallel durchsucht werden (Abbildung 14). Anschließend werden die Zwischenergebnisse verglichen und eine endgültige Lösung wird gefunden. Es ist aber schwieriger, den möglichen Gewinn abzuschätzen, da dafür nicht nur die Ausführungszeiten betrachtet werden müssen, sondern auch die Datengröße und der Durchsatz. In Abbildung 14 ist dieser Sachverhalt dargestellt. Links ist die sequenzielle Ausführung zu sehen. Im rechten Teil ist die parallele Variante mit zwei Fäden dargestellt.

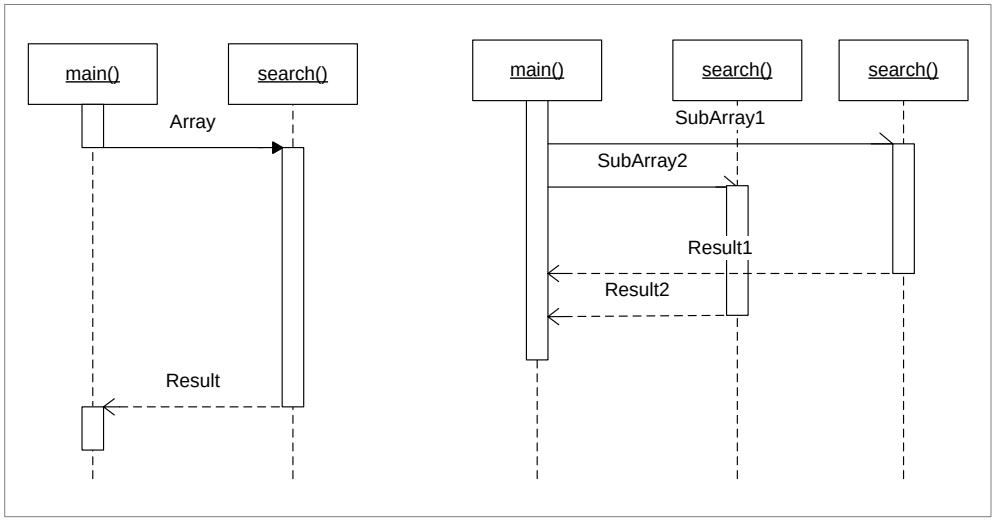


Abbildung 14: Sequenzdiagramme für die Veranschaulichung des Datenparallelismus

Ein Hinweis auf Datenparallelität sind die in 2.2.3 beschriebenen Ausdehnungsbereiche. Eine Menge von Objekten wird in so einem Bereich parallel bearbeitet. Es ist auch möglich, dass am Ende eines Ausdehnungsbereichs eine Menge aus neuen Objekten als Ausgabe vorkommt. Die Bearbeitungsreihenfolge innerhalb eines parallelen Bereichs oder einer Schleife kann durch Sequenzdiagramme beschrieben werden.

Statische und dynamische Parallelisierung

Eine weitere Unterteilung bezieht sich auf den Zeitpunkt der Parallelisierung. Wenn die Entscheidung neue parallele Fäden zu erzeugen, zur Laufzeit passiert und von der Laufvariablen abhängig ist, dann ist dies eine dynamische Parallelisierung. So ist eine parallele Schleife dynamisch, wenn die Menge der Arbeit vorher nicht bekannt ist. Das Gegenteil davon ist eine statische Parallelisierung, wenn zum Beispiel eine Methode, zwei nebenläufige Methoden aufruft.

Zusammenfassung in Bezug auf diese Arbeit

In dieser Arbeit werden Modelle auf implizite Parallelität analysiert und durch musterbasierte Transformationen parallelisiert. Dabei liegt der Schwerpunkt auf der Aufgabenparallelität. Auf der Modellebene entstehen statische parallele Muster, die bei der Implementierung auch dynamische Elemente, wie parallele Schleifen beinhalten können. Liegt noch zusätzlich die Information über Ausführungsumgebung vor, so kann der Parallelitätsgrad entsprechend angepasst und die Effektivität optimiert werden.

2.3.4 Parallele Architekturmuster

Einige klassische Architekturmuster lassen sich für die parallelen Anwendungen gut verwenden Es sind aber auch spezielle parallele Muster entstanden, wie *Pipeline* oder *Producer-Consumer* [Pankr11]**Ошибка! Источник ссылки не найден.**. Hier werden einige von ihnen näher betrachtet. In dieser Arbeit werden auch eigene kleine Muster vorgestellt, die als Hinweise auf mögliches Parallelisierungspotenzial verwendet werden. Diese kleineren Muster können dann zu einen großen Entwurfsmuster (zum Beispiel Master-Worker oder Pipeline) zusammengesetzt werden.

Parallele Threads und Tasks

Ein Thread (deutsch Faden) ist Teil eines Prozesses. Parallele Arbeitsthreads sind bereits in vielen Programmiersprachen (wie Java oder .NET) vorhanden und können für die Ausführung der unterschiedlichen Aufgaben erzeugt werden. Im Gegensatz zum Multitasking, bei dem mehrere unabhängige Programme voneinander abgeschottet quasi-gleichzeitig ausgeführt werden, sind die Threads eines Anwendungsprogramms nicht voneinander abgeschottet und sie können somit sogenannte Wettlaufsituationen (engl. *race conditions*) verursachen, die durch Synchronisation vermieden werden müssen.

Es existieren auch spezielle Entwurfsmuster, wie Fäden-Fabrik (engl. *ThreadFactory*) oder *Thread Pool*, die die Erzeugung und Verwendung von Fäden einfacher machen. Abbildung 15 zeigt ein schematischer Aufbau von einem *Thread Pool* unter .NET. Dabei werden die Arbeiterfäden (engl. *worker thread*) dynamisch erzeugt und aus einer globalen Schlange auf physikalische CPU-Kerne abgebildet.

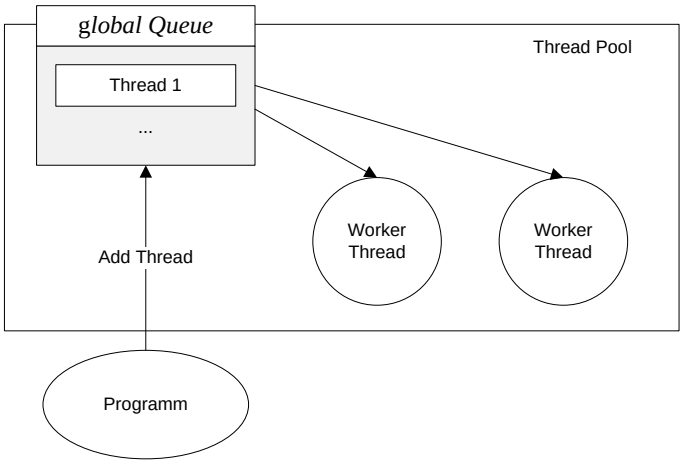


Abbildung 15: Thread Pool in .NET 4.0

Master-Worker

Das Master-Worker-Entwurfsmuster arbeitet nach einem ähnlichen Prinzip wie das Client-Server-Konzept. Beim Client-Server existieren mehrere unterschiedliche Prozesse, die einen Dienst unterstützen. Hier steht ein Master-Prozess im Mittelpunkt, der an eine beliebige Anzahl von anderen Prozessen, auch Worker genannt, Daten zur Bearbeitung sendet und auf die

Ergebnisse wartet. Die zentrale Aufgabe des Masters dabei ist die Koordination des Datenaustausches, der Lastenverteilung und die korrekte Terminierung der Arbeiter-Fäden.

Abbildung 16 zeigt eine schematische Darstellung dieses Musters aus [Pan11]. Das Muster kann durch eine dynamische Analyse relativ leicht erkannt werden. Die Auftraggeber-Methode verbraucht relativ viel Zeit (hauptsächlich durch Warten) und berechnet selber aber kaum etwas. Die Arbeiter-Methoden sind oft relativ klein, werden aber mehrmals von dem Auftraggeber aufgerufen.

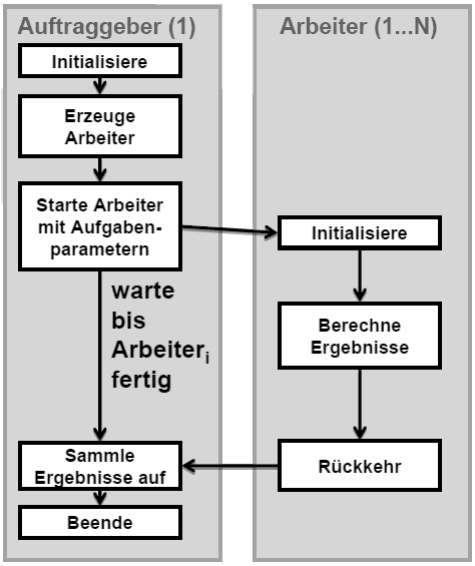


Abbildung 16: Auftraggeber-Auftragsnehmer-Muster aus [Pan11]

Pipeline

Die Pipeline ist ein Entwurfsmuster, das typischerweise zur Stromverarbeitung verwendet wird. Fließband ähnelt sich dem Auftraggeber-Arbeiter-Muster: jeder Arbeiter-Faden besteht aber aus mehreren Stufen, die wiederum parallelisiert werden können. Die Parallelität entsteht hier durch die zeitlich überlappte Ausführung nacheinander folgender Stufen. In einem linearen Fließband begrenzt die Fließbandstufe mit dem höchsten Rechenaufwand die Beschleunigung im gesamten Fließband [Pan11].

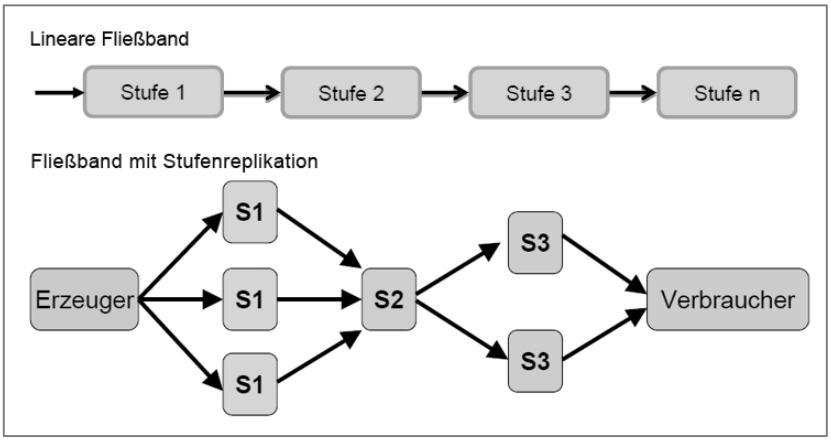


Abbildung 17: Verschiedene Fließbandarten aus [Pan11]

Parallele Schleifen

Das Konzept von parallelen Schleifen ist ganz einfach. Jede Iteration definiert eine Aufgabe, die gleichzeitig mit allen anderen Iterationen ausgeführt werden kann. Eine solche parallele Schleife endet erst, wenn alle Iterationsaufgaben beendet sind. Parallele Schleifen über Datenarrays gab es schon in der 1980er Jahren, wobei die Textkörper der Schleifen in der Regel eine recht einfache Codestruktur haben.

2.3.5 Automatische Parallelisierung

Heutzutage gibt es sehr viele sequenziell geschriebene Anwendungen. Oft sind es mehrere tausend Codezeilen, die analysiert werden müssen, um das Programm zu parallelisieren. Eine automatische Parallelisierung kann die Parallelität, die im verwendeten Algorithmus implizit vorhanden ist, entdecken und nutzen. Eine automatische Mustererkennung soll auch die folgende Parallelisierung leichter machen, da ein sequenzielles Muster durch ein paralleles Muster einfach ersetzt werden könnte.

In der Arbeit „*Parallel Pattern Detection for Architectural Improvements*“ [Hot11] wird versucht, die Entwurfsmuster automatisch zu erkennen. Dafür werden statische Codeanalyse sowie die dynamischen (engl. *run-time*) Leistungsindikatoren verwendet. In [Hot11] werden alle Entwurfsmuster in drei Gruppen aufgeteilt. Abbildung 18 zeigt drei Gruppen mit den entsprechenden Entwurfsmustern. Die Entwurfsmuster, die sich mit der Datenparallelität beschäftigen, werden in dieser Arbeit in zwei weitere Gruppen unterteilt: Bearbeitung von Daten (*data decomposition*) und die Datenverteilung (*flow of data*).

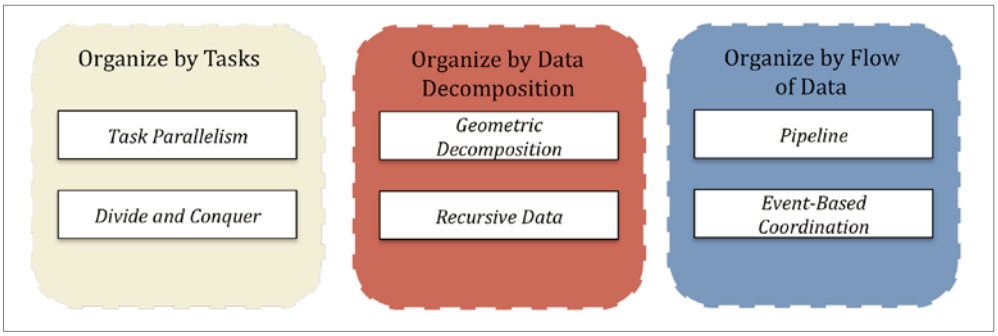


Abbildung 18: Drei Gruppen der Entwurfsmuster nach [Hot11]

Für die Mustererkennung werden in dieser Arbeit die Kommunikationsmuster (engl. *sharing patterns*) analysiert und auf ihre Komptabilität mit dem Entwurfsmuster verglichen. Es wird versucht, anhand von Kommunikationsmustern ein passendes Entwurfsmuster zu wählen:

	Task Parallelism	Divide and Conquer	Geometric Decomposition	Recursive Data	Pipeline	Event-based Coordination
Read-Only	★★★		★	★★★		
Migratory	★★	★★★	★★		★★★★	★★★★
Producer/ Consumer		★★	★★★★			
Private	★★★★	★	★	★★★★	★	★

Abbildung 19: Kommunikations- und Entwurfsmuster

Wie aus der Abbildung erkennbar ist, werden für private und *read-only*-Daten die Muster ohne großen Datenaustausch gewählt (zum Beispiel die *tasks*). Diese Art der Parallelisierung wird

auch „peinlich parallel“ (engl. *embarrassingly parallel*) genannt, da einzelne Fäden gar nicht miteinander kommunizieren und die Parallelisierung problemlos möglich ist. Für „wandernde Daten“ (engl. *migratory data*) ist es üblich, ein pipeline- und nachrichtenbasierte Muster zu verwenden. In diesen Mustern merken sich die Fäden die verschiedenen Phasen des Programms.

2.4 Graphen und Graphersetzungssysteme

Unter einem gerichteten Graphen wird in dieser Arbeit analog zu [Jak08] ein Tupel (*N,E,s,t*) verstanden, wobei:

- *N* - Menge der Knoten (engl. *nodes*),
- *E* - Menge der Kanten (engl. *edges*)
- $s: E \rightarrow N$ – Menge der Quellknoten zu einer Kante *E*
- $t: E \rightarrow N$ – Menge der Zielknoten zu einer Kante *E*

Eine Kante führt von einem Quell- zu einem Zielknoten, womit die Richtung der Kante festgelegt wird. Für einfache Graphen, bei denen zwischen je zwei Knoten in jeder Richtung nur eine Kante auftreten kann, wäre eine Knotenmenge N mit einer Kantenrelation *E* = (*N;N*) ausreichend, für Multigraphen jedoch benötigen wir diese kompliziertere Modellierung [Jak08].

Graphersetzungssysteme werden für die Beschreibung und regelbasierte Veränderung eines Graphen verwendet. Das System besteht aus einem Mustergraphen *L* und einem Ersetzungsgraphen *R*. Bei der Anwendung einer Regel wird im Arbeitsgraphen H eine Instanz des Mustergraphen gesucht und durch eine Instanz des Ersetzungsgraphen ersetzt, mit einem veränderten Arbeitsgraphen, dem Ergebnisgraphen H‘ als Resultat. Dieses Vorgehen ist in Abbildung 20 abstrakt skizziert.

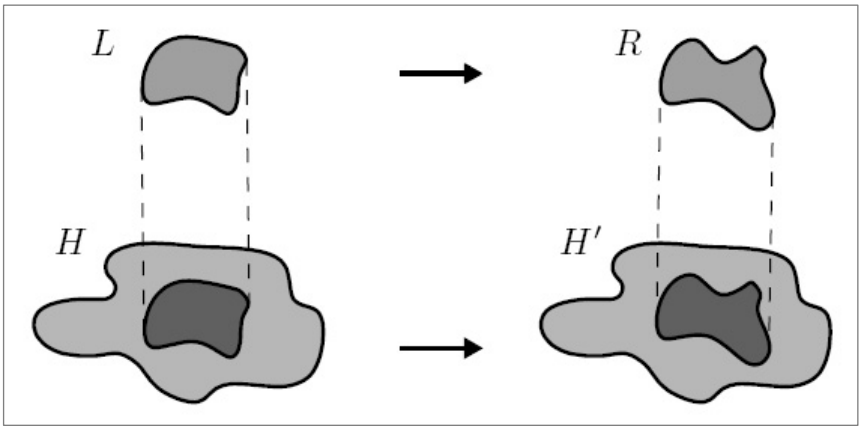


Abbildung 20: Graphersetzungssystem [Jak08]

Vor der Anwendung einer Regel $p: L \rightarrow R$ muss im Arbeitsgraphen erst nach einer Instanz des Mustergraphen *L* gesucht werden. Eine gefundene Instanz heißt Passung (engl. *match*) *m* von *L* in *H*. Formal wird sie durch einen Graphhomomorphismus vom Mustergraphen in den Arbeitsgraphen beschrieben.

Im Rahmen dieser Arbeit wird der graphenbasierte Ansatz verwendet, um das kombinierte Modell, das aus mehreren Graphen besteht, zu transformieren. Dafür wird das Graphersetzungssystem GrGen.NET [GrGen] verwendet. Der Mustergraphen *L* wird dabei als Suchmuster bezeichnet und die Abbildung $p: L \rightarrow R$ als Musterersetzung. Mehr dazu findet man in Kapitel 5 dieser Arbeit.

3 VERWANDTE ARBEITEN

Die modernen Softwaresysteme werden immer größer und komplexer. Gleichzeitig müssen sie flexibel genug bleiben, um schnell auf die neuen Anforderungen der Kunden reagieren zu können. Die Verwendung von Modellen und Architekturmustern in der Softwareentwicklung bietet diese Flexibilität und macht die Entwicklung von großen Systemen deutlich leichter und übersichtlicher. In diesem Kapitel werden einige wissenschaftliche Arbeiten betrachtet, die sich mit der musterbasierten Analyse von UML-Modellen beschäftigen.

Ein weiterer Trend in der betrieblichen Softwareentwicklung ist die Entwicklung von skalierbaren parallelen Softwaresystemen. Dies wiederrum bringt einige Herausforderungen und Fragen mit sich: So verfügen beispielweise Multikernprozessoren oft über einen gemeinsamen Cache oder greifen konkurrierend auf einen Hauptspeicher zurück. Wird dies nicht korrekt bearbeitet, entstehen häufig Fehler bzw. Ausnahmen bei der Ausführung. Auch hier existieren spezielle parallele Architektur-Muster, die die Entwicklung erleichtern sollen. In diesem Kapitel werden auch Arbeiten analysiert, die sich mit den parallelen Mustern beschäftigen.

Da diese Arbeit modellbasierte Analyse für spätere automatische Parallelisierung nutzt, werden in diesem Kapitel auch zwei weitere Arbeiten betrachtet, die sich mit dem Thema „automatische Parallelisierung“ beschäftigen: [Tou09], [Tou10]. Beide Arbeiten beschreiben einen Parallelisierungsansatz auf Codeebene und nutzten dafür keine Modelle. Die Zusammenfassung am Ende dieser Kapitel vergleicht alle verwandten Arbeiten mit dem eigenen Ansatz und zeigt deutlich alle Gemeinsamkeiten, sowie die Unterschiede nochmal deutlich an.

3.1 Analysenmethoden für UML-Modelle

Die *Unified Modeling Language* (UML) wird von der *Object Management Group* [OMG] entwickelt und ist heute zur Standartsprache im Bereich Spezifikation, Konstruktion und Dokumentation von Software geworden. Die UML wird oft für objektorientierte Analyse und Design (OOAD) im Entwicklungsprozess von Softwaresystemen verwendet. Dabei können große Softwareteile in einer standardisierten Form beschrieben werden können.

Die UML hat außerdem verschiedene Erweiterungsmechanismen wie zum Beispiel die Erweiterung des UML2-Metamodells, basierend auf dem leichtgewichtigen Erweiterungsmechanismus der Profile. Für die Erweiterung vorhandener Modellelemente werden sogenannte Stereotype verwendet. Sie stellen vor allem die möglichen Verwendungszusammenhänge und den Kontext eines UML-Elements dar. Viele Wissenschaftler und Firmen nutzen diese Erweiterungsmöglichkeit der UML, um eigene spezifische Sprachelemente und Konstrukte zu entwickeln, zum Beispiel für die Modellierung der parallelen Softwaresysteme wie in [PF+04].

In diesem Kapitel werden vier Arbeiten vorgestellt, die sich mit der musterbasierten Analyse von UML auseinander setzen: [Woh05b], [PF+04], [Gud10], [Stö05].

3.1.1 Pattern-based Analysis of UML Activity Diagrams [Woh05b]

Diese Arbeit untersucht die Aktivitätsdiagramme der aktuellen Version (UML 2.2). Die Modelle werden in Form einer Sammlung von Mustern betrachtet. Dabei werden Kontroll- und Datenflussmuster in den prozessorientierten Informationssystemen separat betrachtet und analysiert. Zweck dieser Analyse ist, die relativen Stärken und Schwächen der Spezifikation von UML 2.x zu beurteilen und Möglichkeiten der Ansprache potenzieller Mängel zu identifizieren.

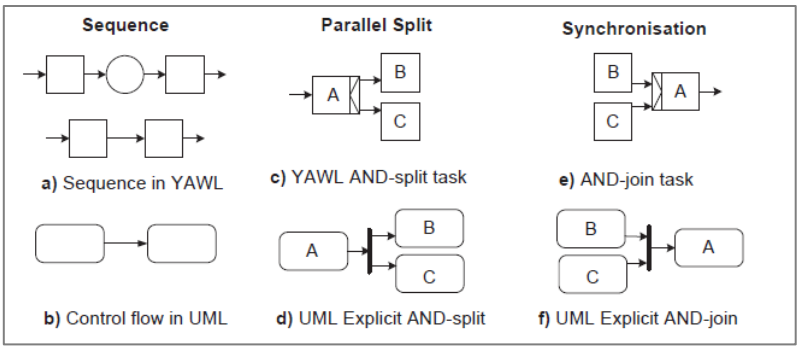


Abbildung 21: Kontrollfluss-Muster in UML und YAWL aus [Woh05b]

In dieser Analyse werden die Muster in UML mit ähnlichen Mustern, die in der Sprache YAWL beschrieben sind verglichen. Die YAWL ist eine formale Sprache und die Musterimplementierungen in YAWL lassen keinen Raum für Unklarheiten. Abbildung 21 zeigt einige Kontrollfluss-Muster in UML und YAWL im Vergleich. In dieser Arbeit wird keine Analyse der Modelle gemacht, sondern nur die Mächtigkeit und Eindeutigkeit von UML Modellen bewertet. Es werden auch unterschiedliche Muster beschrieben und klassifiziert.

Die Arbeit untersucht unter anderem, den Kontrollflussverlauf zwischen Aktionen und Kontrollknoten (siehe Kapitel 2.2.3). Dabei werden alle Aktionen in drei Gruppen unterteilt: *Invocation Actions*, die anderen Aktionen anrufen oder senden und die Signale akzeptieren. Die *Read-Write*-Aktionen, die den Zugriff auf Objekte durchführen und deren Eigenschaften (Werte) verändern. Die dritte Gruppe bilden die *Computation Actions*, die Berechnungen durchführen und Eingabedaten in Ausgabedaten transformieren.

Alle untersuchten Muster werden als einfache (triviale) oder komplexere Muster klassifiziert. Die einfachen Muster, wie zum Beispiel eine Sequenz, können meistens mithilfe einer oder zweier Knoten dargestellt werden (siehe Abbildung 21). Die komplexeren Muster (engl. *multiple instances patterns*) bestehen aus mehreren Aktionen und Kontrollknoten. Diese Muster zeichnen sich dadurch aus, dass hier mehrere parallele Prozesse gleichzeitig existieren können (es existieren also mehrere Kontrolltoken, siehe 2.2.3). Für diese Fälle sind auch Synchronisierungsmechanismen nötig, die teilweise durch UML abgebildet werden können. Zum Schluss wird die Unterstützung von verschiedenen Mustern in UML 2.0 mit UML 1.4 verglichen. Wie man aus folgender Tabelle ersichtlich ist, gibt es dort einige Unterschiede und mit UML 2.0 können viele komplexere Muster modelliert werden, wie zum Beispiel *Implicit Termination*.

Nr	Muster	2.0	1.4	Nr	Muster	2.0	1.4
1	Sequence	+	+	11	Implicit Termination	+	–
2	Parallel Split	+	+	12	MI without Synchronization	+	–
3	Synchronization	+	+	13	MI with a priori Design Time Knowledge	+	+
4	Exclusive Choice	+	+	14	MI with a priori Runtime Knowledge	+	+
5	Simple Merge	+	+	15	MI without a priori Runtime Knowledge	–	–
6	Multi Choice	+	–	16	Deferred Choice	+	+
7	Synchronizing Merge	–	–	17	Interleaved Parallel Routing	–	–
8	Multi-Merge	+	–	18	Milestone	–	–
9	Discriminator	+	–	19	Cancel Activity	+	+
10	Arbitrary Cycles	+	–	20	Cancel Case	+	+

Tabelle 1: Unterstützung von verschiedenen Muster in UML 1.4 und 2.0 nach [Woh05b]

3.1.2 Semantics and Verification of Data Flow in UML 2.0 Activities [Stö05]

Diese Arbeit zeigt, wie die Aktivitätsdiagramme in UML 2.0 für eine formale Analyse des Steuerungs- und Datenflusses verwendet werden können. Es wird gezeigt, wo die übliche UML-Konstrukte nicht so leicht formalisiert werden können und wo man sie für eine formale Analyse doch gebrauchen kann.

Zuerst werden die Unterschiede zwischen UML 1.x und UML 2.0 beschrieben und diskutiert. In der UML 2.0 wurde das Metamodell komplett überarbeitet und ist jetzt viel sauberer, vollständig und orthogonal, als es früher war. Viele Details wurden verbessert, und die UML 2.0 kann jetzt wieder als objektorientiert bezeichnet werden. Die wichtigste Neuerung ist, dass einzelne Elemente jetzt mit viel weniger Einschränkungen als bisher miteinander kombiniert werden können. So können Klassendiagramme eine Referenz auf die Aktivitätsdiagramme haben und somit den notwendigen Kontext für das Aktivitätsdiagramm ausbilden.

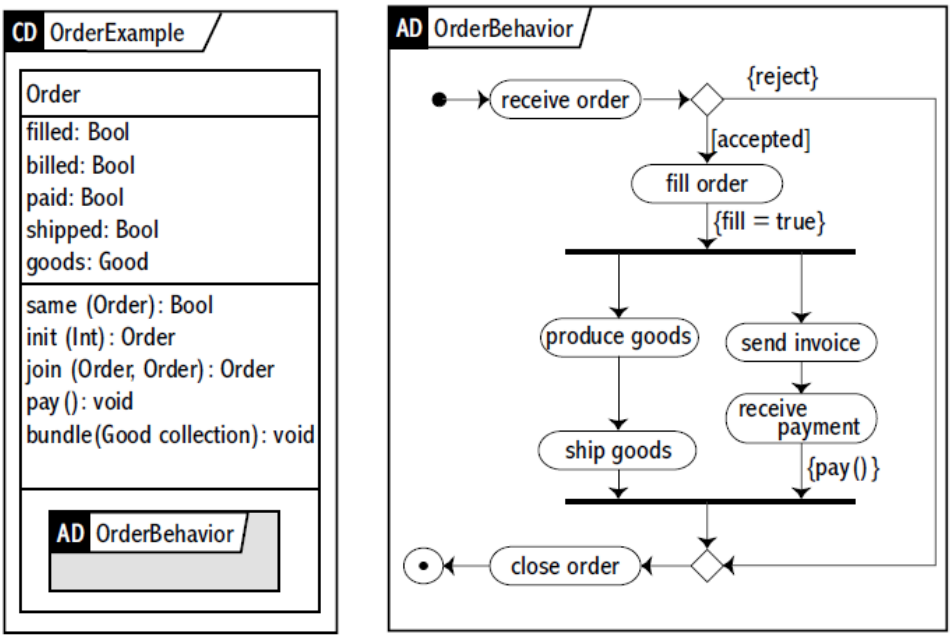


Abbildung 22: Kombination von Klassen- und Aktivitätsdiagramm aus [Stö05]

Auf diesem Aktivitätsdiagramm wird folgendes Verhalten bestimmt: Zuerst wird ein Auftrag (*Order*) erhalten – Aktion „*receive order*“. Danach wird entschieden, ob er akzeptiert oder abgelehnt wird. Falls der Auftrag akzeptiert wird, bekommt der Kontext den Status, der in eckigen Klammern steht, nämlich *accepted*. Als Nächstes wird die Aktion „*fill order*“ ausgeführt. Dabei wird das Attribut *fill* auf *true* gesetzt (links in dem Klassendiagramm heißt das Attribut *filled*, was vermutlich ein Fehler ist). Der nächste Schritt spaltet den Kontrollfluss in zwei nebenläufige Prozesse. Interessant ist, dass der aktuelle Standard keine Aussagen über das Verhalten der Datenobjekte an dieser Stelle macht. Es kann davon ausgegangen werden, dass in Systemen mit gemeinsamem Speicher die Objekte sich nicht vermehren und es wird ein Objekt von beiden Fäden verwendet.

Konkrete Syntax

Die konkrete Syntax in den Aktivitätsdiagrammen hat sich in der Version 2.0 kaum verändert, die Syntax für Datenfluss aber schon. Der Standard erlaubt jetzt drei verschiedene Darstellungsarten für Datenflüsse (siehe Abbildung 23). Erstens gibt es eine Notation ähnlich der UML 1,5 (links im Bild), wobei Datenströme explizit angegeben werden. Der Datenfluss wird jetzt durch *ObjectNodes* (Objektknoten, die durch kleine Rechtecke bezeichnet werden)

und *ObjectFlows* (Datenfluss, der durch Pfeile zwischen einzelnen Aktivitäten definiert wird) bezeichnet.

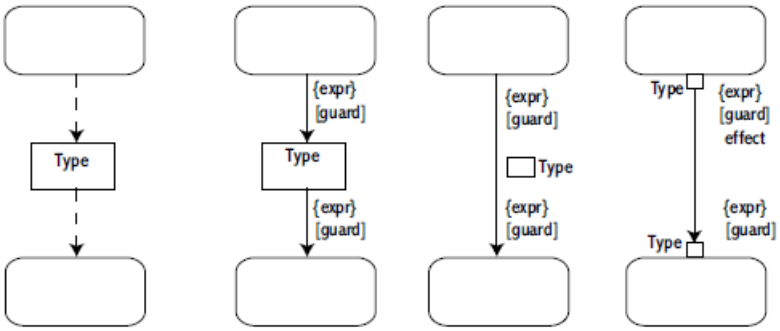


Abbildung 23: UML 1.5 (links) und drei alternative gleichwertige Notationen in UML 2.0

Zweitens gibt es eine vereinfachte Version, die ein Objekt einem Kontrollflusspfeil zuweist. Visuell wird ein Objekt neben dem Pfeil angezeigt. Somit kann man zuerst die Kontrollflüsse allgemein festzulegen und erst später die Datentype definieren. Die dritte, in dem Standard vorgeschlagene Notation, definiert die Pins als eine Unterklasse von *ObjectNode*. Pins sind Eingabe- und Ausgabeparameter für die Aktivitäten.

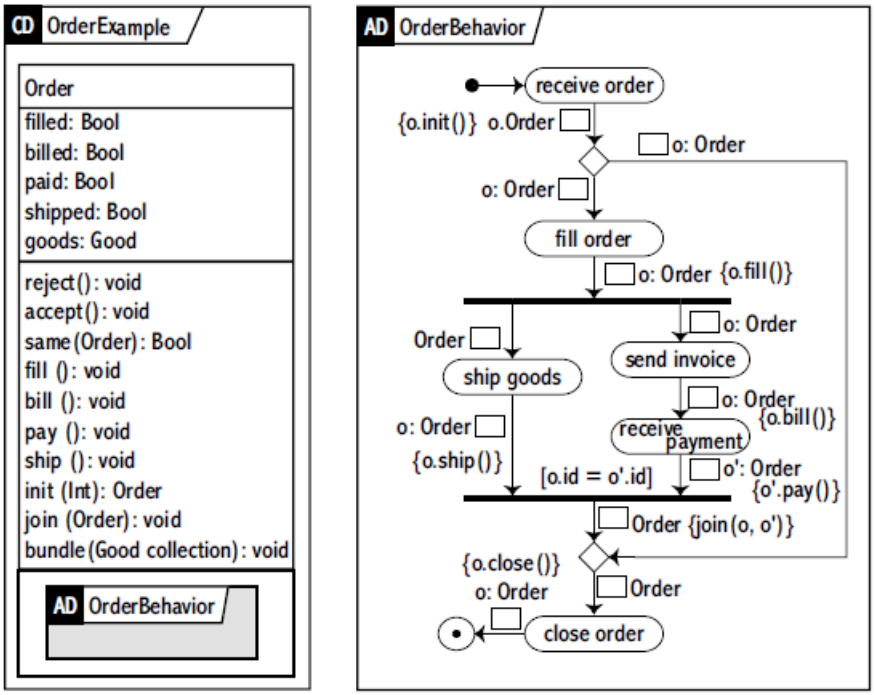


Abbildung 24: Oberes Beispiel mit zusätzlichen Objekt-Notationen

Das obere Beispiel kann nun durch zusätzliche Notationen vervollständigt werden. In dieser Form kann man das Aktivitätsdiagramm für Datenabhängigkeitsanalyse nutzen. Wie man sieht wird das Objekt *Order* in beiden parallelen Faden verwendet.

Analyseansatz

In der neuen UML-Version wird eine Petri-Netz-artige Semantik eingeführt. In der vorliegenden Arbeit wird ein Mapping vorgeschlagen, um die Validierung und die Analyse dem Standardverfahren für die Petri-Netze durchführen zu können (mehr dazu in [Jen92]). Die

formale Semantik ist dabei ziemlich einfach. Jede Aktivität wird als Tupel <*ActivityNodes, ActivityEdges*> auf ein Petri-Netz-Tupel <*N, SigAlg, Farbe, guard, effect*> abgebildet.

Als ein wichtiger Grund für die Verwendung Petri-Netz-basierter Tools in der Entwicklung und Analyse werden die Möglichkeit, Simulationen durchzuführen, sowie die Verwendung von quantitativen Analyse-Techniken genannt. Folgende Abbildung stellt ein Petri-Netz als einen Graphen dar, der aus zwei Arten von Knoten aufgebaut ist, die Stellen (Ovale) bzw. Interaktionen (Rechtecke) genannt werden. Die Knoten sind durch Kanten verbunden, und zwar jeweils von einer Stelle zu einer Interaktion oder umgekehrt. Die Stellen können mit beliebig vielen Marken belegt sein. Eine solche Markierung stellt einen verteilten Zustand des Systems dar.

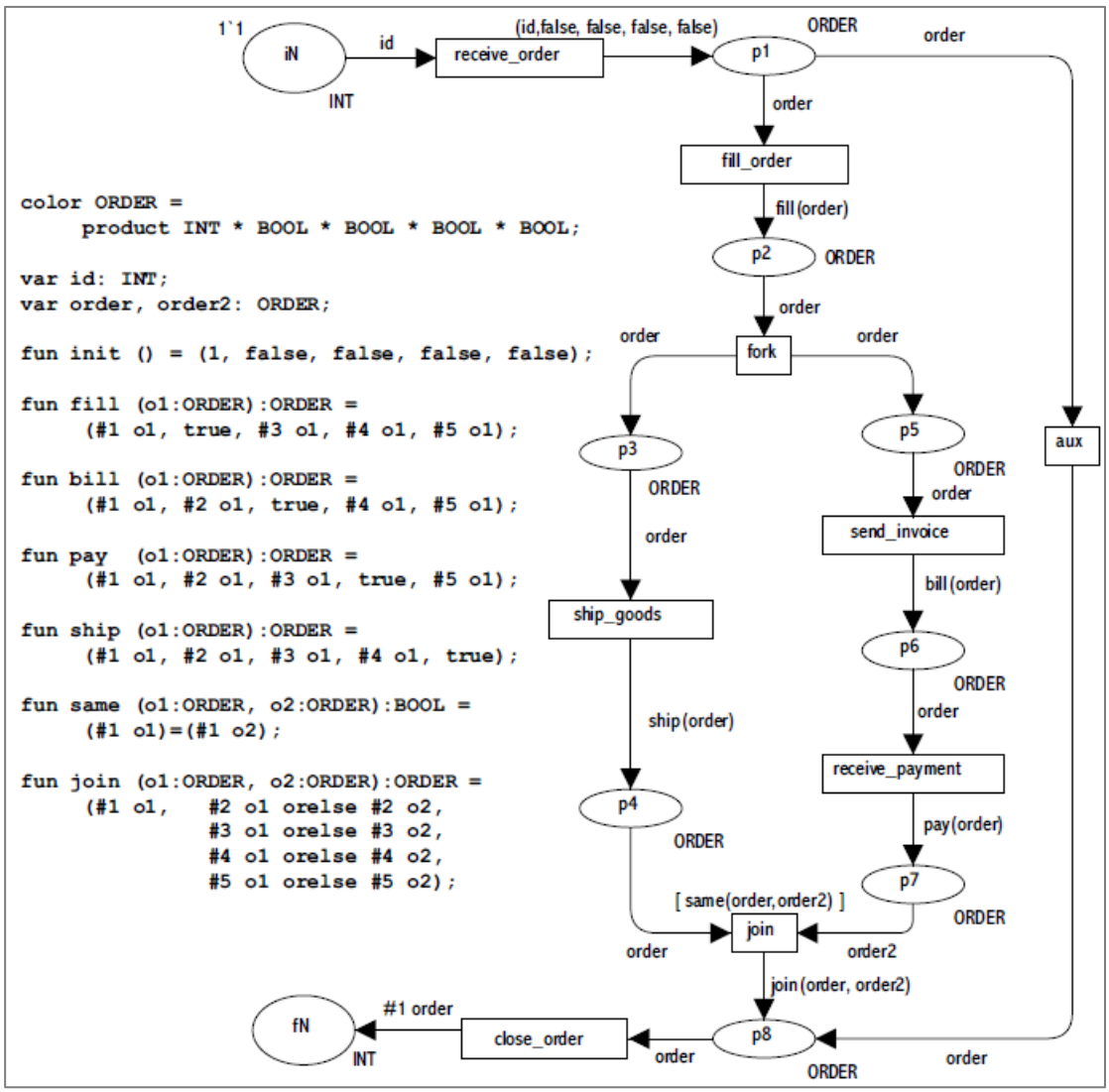


Abbildung 25: Das Petri-Netz, das das obere UML-Beispiel repräsentiert

Defizite dieser Arbeit

Die in dieser Arbeit vorgeschlagene Methode zur Analyse von Aktivitätsdiagrammen basiert auf einen Mapping zwischen UML-Modellen und Petri-Netzen. Es werden Analysetechniken verwendet, die typisch für Petri-Netze sind, wie zum Beispiel *Trasing*. Dabei liefert der aktuelle UML-Standard einige Beschriftungsarten nicht, die für typische Analysetechniken von Petri-Netzen notwendig sind. Deswegen werden syntaktische Erweiterungen für UML-Aktivitätsdiagramme gebraucht um solche Aspekte wie Durchsatz, Frequenz,

Wahrscheinlichkeit und Latenz zu beschreiben. Es gab eine Reihe von Ansätzen zur Bereitstellung und Nutzung solcher Erweiterungen für UML 1.x es ist aber unklar ob diese auf die neue Version der UML übertragen werden können.

In meiner Arbeit will ich ähnliche Kombination des Datenflusses mit Objektknoten verwenden, aber für andere Analysearten. So wird hier der Parallelisierungspotenzial gar nicht untersucht, obwohl die Voraussetzungen für eine solche Analyse gegeben sind. Auch die Datenabhängigkeiten und Wettläufe können dabei theoretisch erkannt werden.

3.1.3 Towards an UML Based Graphical Representation of Grid Workflow Applications [PF+04]

In dieser Arbeit wird ein Ansatz zur grafischen Modellierung und Beschreibung von Grid-Anwendungen auf Basis der Unified Modeling Language (UML) dargestellt. Der Ansatz stellt eine grafische Darstellung auf einem weitverbreiteten Standard (UML) dar, die für die Modellierung von Kontrollfluss, Datenfluss, Synchronisation, Benachrichtigung und Zwängen erweitert wird. Außerdem werden typische Grid-Konstrukte, wie Broadcast und parallele Schleifen unterstützt. Es wird ein grafischer Editor „Teuta“ präsentiert und anhand eines Beispiels (3D-Bildrekonstruktion in einer medizinischen Anwendung) evaluiert.

In der vorliegenden Arbeit werden UML-Aktivitätsdiagramme mit Stereotypen für die Beschreibung von Daten- und Kontrollfluss verwendet. Die Beschreibung erfolgt auf ziemlich abstraktem Niveau, obwohl eine Komposition (Verschachtelung) von *Workflows* auch möglich ist.

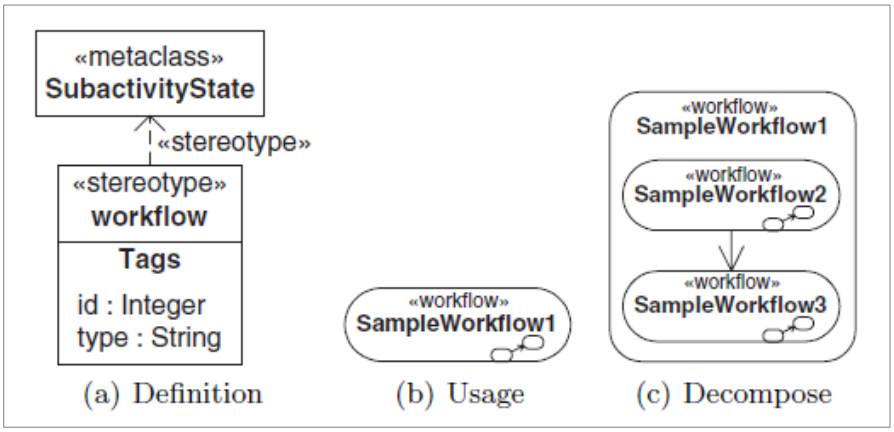


Abbildung 26: Hierarchische Aufbau eines Stereotyps

Es werden drei Standartoperationen definiert:

- **Berechnung** (Compute) – bezeichnet einen rechnungsintensiven Prozess.
- **Datentransfer** (TransferData) – bezeichnet Datentransfer im Grid. Dabei können Quelle und Ziel sowie der Kommunikationstyp als Eigenschaften definiert werden.
- **Sicht** (View) – bedeutet Datenvisualisierung

Ein Berechnungsblock, der zusätzlich mit “*“ gekennzeichnet ist, kann parallel ausgeführt werden. Das bedeutet, dass diese Berechnung auf mehreren Grid-Knoten parallel ausgeführt werden darf. Wie die Berechnung und die dabei beteiligten Datenstrukturen genau aussehen wird in diesem Ansatz nicht beschrieben.

Wie Abbildung 27 zeigt, wird ein Datenfluss nur sehr oberflächlich dargestellt. Die genauere Struktur der Daten wird nicht abgebildet:

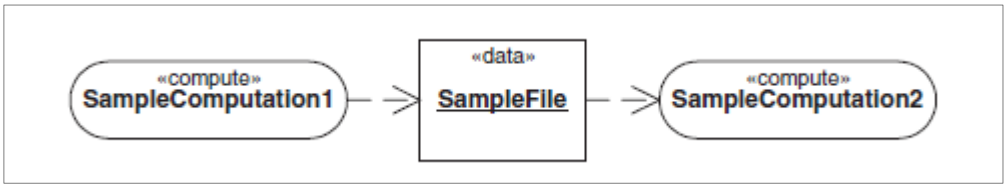


Abbildung 27: Datenfluss Beispiel

Für die Beschreibung des Kontrollflusses werden Standartelemente der UML verwendet. Ein Brunch sieht dabei genauso aus wie eine *if-else*-Anweisung. Analog werden die Schleifen modelliert. Abbildung 28 zeigt eine Verzweigung (links im Bild) und eine Schleife (rechts), die mithilfe von Verzweigungsknoten ♦ modelliert werden.

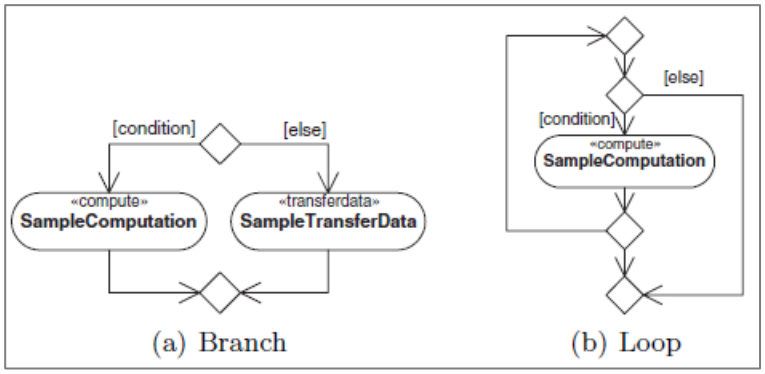


Abbildung 28: Beschreibung der Kontrollflusses in [PF+04]

Zusammenfassung

Der oben vorgestellte Ansatz kann für die Modellierung der Grid-Anwendungen verwendet werden, reicht aber nicht aus für die Analyse der Datenabhängigkeiten. Das ausgewählte Abstraktionsniveau passt nicht für die Modellierung von Anwendungen für Systeme mit verteiltem Speicher.

Für eine Datenabhängigkeitsanalyse soll man die bei der Berechnung beteiligten Datenstrukturen genau kennen. Eine Lösung dafür kann die Kombination von Aktivitäts- und Klassendiagrammen sein:

- Die Berechnung kann durch Markierung von dabei beteiligten Objekten oder Attributen definiert sein. Eine Art Mapping von Objekten und Attributen ermöglicht eine Datenabhängigkeitsanalyse, falls mehrere Berechnungen parallel ablaufen werden.

- Datenfluss kann auch durch zusätzliche Notationen genauer beschrieben werden (Objekte, Datentyp, Strukturen).

3.1.4 A Coordination-Based Model-Driven Method for Parallel Application Development [Gud10]

Diese Arbeit stellt eine Top-Down-Methode vor, die dem gewöhnlichen Software-Entwickler bei der Entwicklung von parallelen Anwendungen helfen soll. Der Ansatz basiert auf der Kombination von Koordination-Modellierung mit Model-Driven-Development. Koordination Modellierung bedeutet die Erstellung eines Koordination-Modells. Ein solches Modell stellt eine hochrangige Abstraktion eines parallelen Programms dar. Im Allgemeinen definiert ein Koordination-Modell das Zusammenwirken von aktiven und unabhängigen Einheiten.

In diesem Ansatz wurden eine Metamodell und einer konkrete Syntax als DSL, sowie entsprechende Prototypwerkzeugen für die Erstellung eines Koordination-Modells entwickelt. Abschließend wurde eine umfangreiche Auswertung der Verfahren durchgeführt. Als Zieldomain für entwickelnde DSL wird „*parallel systems software engineering*“ genannt. Dabei hat man sich auf einzelnen Phasen, wie Entwicklung, Debugging und Leistungsoptimierung besonderes konzentriert.

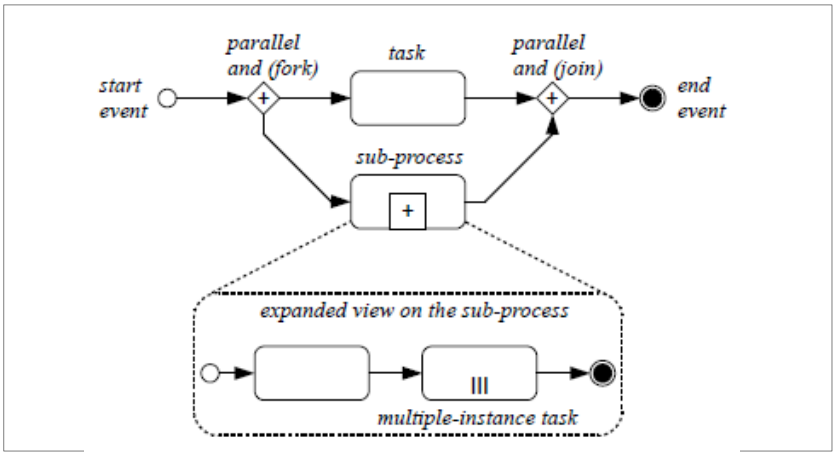


Abbildung 29: Domain-Konzepte aus [Gud10]

Ein Softwaresystem wird als eine Menge der aktiven Einheiten (Prozessen) betrachtet, die in einem globalen Raum liegen und eine einzelne Aufgabe darstellen. Innerhalb eines Prozesses befinden sich weitere Teil-Prozesse oder *Tasks* (Berechnungen). Diese Elemente können sequenziell oder parallel ausgeführt werden. Datenflüsse und passive Datenobjekte sind auch Teile der Domäne.

Folgende Tabelle stellt einzelne Elemente des Metamodells vor:

Bezeichnung	Bedeutung
Process	Eine aktive Einheit innerhalb eines globalen Objektsraum, die Aufgaben und Teil-Prozesse enthält. Anzahl der Prozesse kann sich von der Anzahl der Prozessoren (CPU) unterscheiden und wird bei der Problemzerlegung bestimmt.
Sub-Process	Teil eines übergeordneten Prozesses in dem gleichen globalen Objektraum.
Task	Eine atomare Programmeinheit (Knoten). Sie kann sequenziell oder parallel mit anderen Aufgaben und Sub-Prozessen ausgeführt werden. Ein Task kann erstellen, lesen und Datenobjekte konsumieren.
Data Object	Eine passive Einheit innerhalb des globalen Objektraums. Data Object kann durch Tasks erstellt, gelesen und verbraucht werden.
Control Flow	Angaben, die Ausführung explizit steuern, wie z.B. if, or, parallel
Data Flow	Die Daten-Objekte, die als Eingabe oder Ausgabe eines Tasks definiert sind.

Abbildung 30 zeigt einzelne Phasen des Verfahrens. Der erste Schritt ist die Erstellung eines Koordination-Modells für geplante parallele Anwendung. Danach wird das Koordination-Modell in den Quellcode der Zielplattform transformiert (d.h. die Umsetzung der Koordinierung

im Quellcode). Der dabei entstehende Code wird mit domänenspezifischem sequenziellem Code ergänzt.

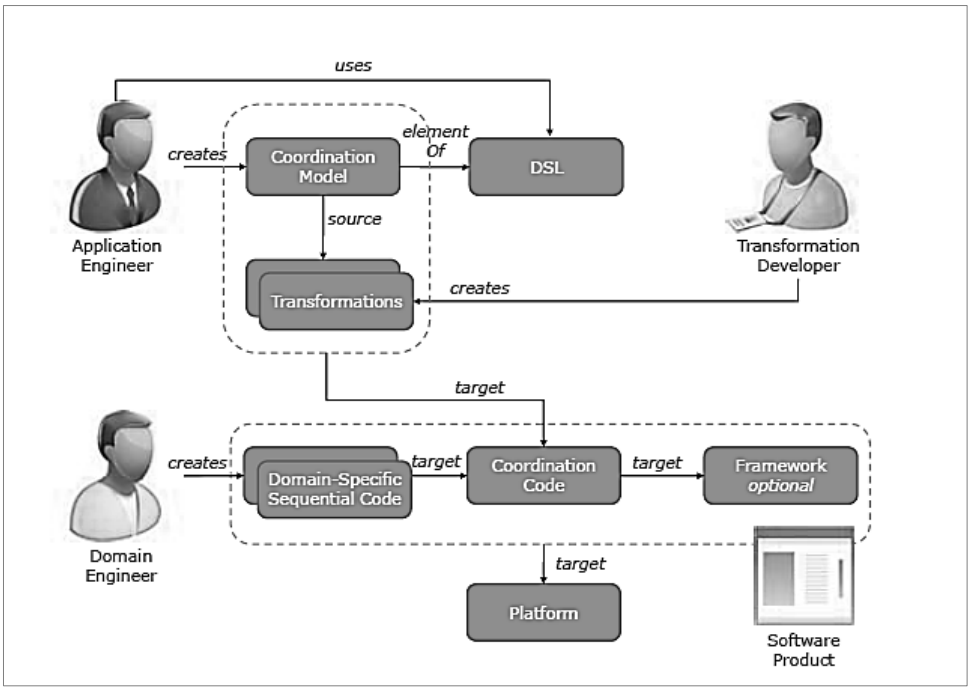


Abbildung 30: Verfahren im Überblick [Gud10]

Defizite dieser Arbeit

In dieser Arbeit wurde vorgeschlagen die parallele Software mithilfe einer DSL zu entwickeln. Die visuellen DSLs sind im Gegensatz zum Quellcode mehrdimensional und sind damit in der Lage, mehrere gleichzeitige Kontrollflüsse ganz natürlich zu präsentieren. Andererseits ist die modellgetriebene Entwicklung komplexer als die klassische Softwareentwicklung und fordert mehr Fachkenntnisse und Erfahrung von dem Entwickler. Die Autoren vermuten, dass die UML-Aktivitätsdiagramme und die Business Process Modeling Notation (BPMN) viele der DSL-Anforderungen erfüllen können. Warum man sich für eine spezielle DSL entscheiden soll ist nicht klar formuliert.

In der vorliegenden Arbeit wollen wir die UML-Modelle mit Stereotypen nutzen um unterschiedliche Daten- und Steuerungsflüsse abzubilden und analysieren zu können (siehe mehr in 4.4.2). Die UML ist weit verbreitet und wird von den meisten Werkzeugen unterstützt. Ziel meiner Arbeit ist es die Entwicklung einfacher zu machen, indem man die parallele Abläufe und Datenabhängigkeiten durch eine automatische Analyse erkennt. Der Entwickler kann dann diese Information nutzen, um mögliche Fehler bei der Parallelisierung zu vermeiden und den Code besser zu strukturieren.

Zusammenfassung

In [Gud10] wird eine modelgetriebene Entwicklung für parallele Softwaresystemen vorgeschlagen. Dabei wird der Code für die Zielplattform mithilfe eines Koordination-Modells erzeugt. Dies bedeutet aber, dass für die verschiedene Plattformen spezielle Code-Generatoren bereitgestellt werden müssen, was wiederum zusätzlichen Kosten und zusätzlichen Aufwand bedeuten kann.

3.2 Ansätze zur automatischen Parallelisierung

In diesem Abschnitt werden verwandten Arbeiten vorgestellt, die sich mit dem Thema „Automatische Software-Parallelisierung“ auf Codeebene auseinandersetzen.

3.2.1 Semi-Automatic Extraction and Exploitation of Hierarchica Pipeline Parallelism Using Profiling Information [Tou10]

Diese Arbeit von Georgios Tournavitis und Björn Franke stellt einen neuartigen Ansatz zur Gewinnung und Nutzung von Pipeline-Parallelität aus sequenziellen Anwendungen vor. Dieser Ansatz unterscheidet sich von den herkömmlichen Parallelisierungsansätzen. Es werden verschachtelte Schleifen und wiederholende Pipeline-Stufen automatisch erkannt und unterstützt. Allerdings zeigt der Ansatz auch einige Schwächen. Da nur Datenabhängigkeiten berücksichtigt werden, die bei der Ausführung der Testfälle anfallen, wird die Korrektheit für weitere Eingaben nicht garantiert. Der Entwickler ist dazu aufgefordert, die Korrektheit selber zu verifizieren.

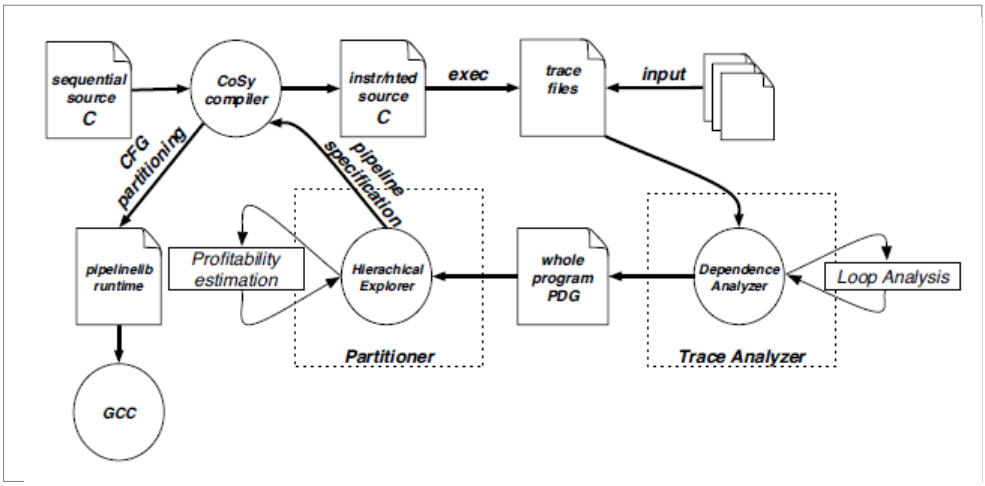


Abbildung 31: Parallelisierungsprozess in [Tou2010]

Eine Übersicht des Parallelisierungsprozesses wird in Abbildung 31 vorgestellt. Die Parallelisierung erfolgt auf Code-Ebene und ist nicht modellbasiert. Das sequenzielle Programm wird durch *CoSy C-Compiler* verarbeitet und instrumentiert. Das Ergebnis wird mit einer oder mehreren repräsentativen Eingaben ausgeführt. Dadurch entsteht eine Reihe von Profil-Dateien, die abschließend für eine Abhängigkeitsanalyse verwendet werden.

Das generierte Abhängigkeitsgraph wird an den Partitionierer übergeben und dort wird eine Pipeline-Spezifikation mit Anmerkungen ausgearbeitet. Das Programm wird dann durch das *CoSy Compiler* zum zweiten Mal bearbeitet. Als Ergebnis wird ein paralleler Code nach der Pipeline-Spezifikation erzeugt. Schließlich ist dieser Code für die Zielplattform mit dem *gcc-Compiler* kompiliert und mit der Pipeline-Laufzeit-Bibliothek verlinkt.

3.2.2 Towards a Holistic Approach to Auto-Parallelization: Integrating Profile-Driven Parallelism Detection and Machine-Learning Based [Tou09]

Parallelisierung auf Instruktionen-Ebene war der erste Schritt in Richtung automatischer Parallelisierung. Die *Compiler* konnten bereits in den 80er Jahren die voneinander unabhängigen Befehle im Code erkennen und parallel ausführen. Da diese Art der Parallelisierung durch *Compiler* erfolgt, wird sie auch *compiler*-basierte Auto-Parallelisierung genannt.

In diesem Bereich wird immer noch geforscht und eine Arbeit des schottischen Wissenschaftler „Towards a Holistic Approach to Auto-Parallelization: Integrating Profile-Driven Parallelism Detection and Machine-Learning Based“ [Tou09] versucht *compiler*-basierte Parallelisierung durch Profildaten und maschinelles Lernen zu verbessern.

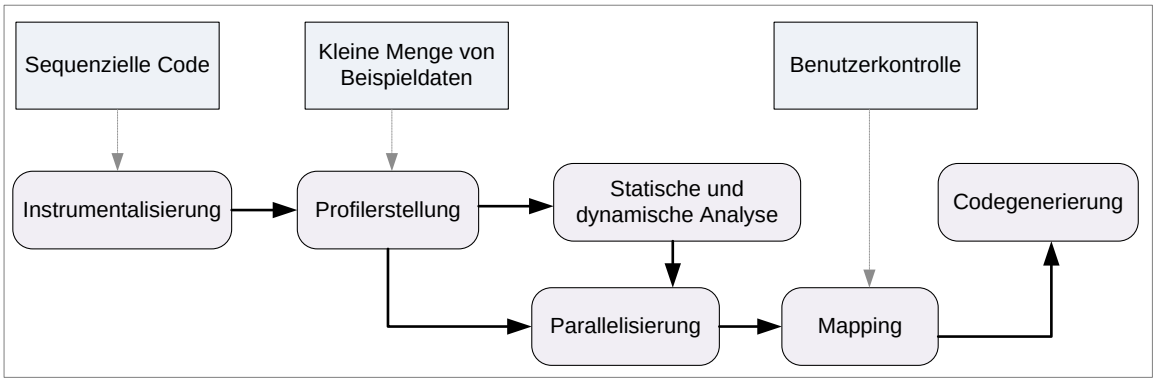


Abbildung 32: Compiler-basierte Auto-Parallelisierung in [Tou09]

Abbildung 32 zeigt wie der Parallelisierungsprozess in [Tou09] aussieht. Zuerst werden Profiling-Daten zu Kontroll-und Datenabhängigkeiten extrahiert und zur Verbesserung der entsprechenden statischen Analysen mit dynamischen Informationen verwendet werden (dynamische Analyse). Danach betrachtet ein, zuvor geschulter, auf maschinelles Lernen basierter Vorhersage-Mechanismus, jeden potenziell parallelen Zyklus. Dabei wird entschieden, ob und wie die Parallelisierung durchgeführt werden soll.

Ein großer Vorteil ist hier, dass durch *compiler*-basierte Auto-Parallelisierung die Entwicklungskosten deutlich reduziert werden können und die Korrektheit des entstehenden Codes formal beweisbar bleibt. Die Parallelisierung selbst erfolgt anschließend durch Verwendung von OpenMP Standard.

3.3 Zusammenfassung:

In diesem Kapitel wurden einige wissenschaftliche Arbeiten beschrieben, die sich mit Modellierung von parallelen Algorithmen und der automatischen Parallelisierung beschäftigen. Die bisherige Unterstützung bei der Parallelisierung beschränkt sich in erster Linie auf die explizite Modellierung oder die automatisierte Refaktorisierung von Quellcode. MAP ermöglicht neben der Erkennung von Parallelisierungspotenzial und Datenabhängigkeiten auf Modellebene zusätzlich eine automatische Parallelisierung durch Modelltransformation. Dabei werden verschiedene Ansätze aus [PF+04], [Gud10] und [Stö05] durch ein erweitertes Metamodell und einen Regelsatz in GrGen.NET implementiert.

Die folgende Tabelle vergleicht die wichtigsten Punkte der hier aufgeführten Arbeiten mit dem in diesem Dokument geplanten Ansatz MAP:

	Woh05b	Gud10	Stö05	ATL	GrGen	MAP
Modellbasierte Analyse	+	+	+	++ *	++ *	++
Betrachtung des Parallelisierungspotenzials	++	+	+			+
Erkennung von Datenabhängigkeiten	-	-	+			++
Parallelisierung auf Modellebene		+	+	+ *	+ *	++
Modelltransformation		+	+	++ **(M)**	++ **(G)**	+ **(G)**
Graphen-/Modellbasiert	**M**	**M**	**M**	**M**	**G**	**M/G**

Tabelle 2: Vergleich von relevanten Arbeiten mit MAP-Ansatz

* durch deklarative Ersetzungsregeln

4 MUSTERBASIERTE MODELLANALYSE UND MODIFIKATION

Diese Arbeit befasst sich mit der Erkennung von Parallelisierungspotenzial auf Modellebene. Dazu wird in diesem Kapitel ein Konzept zur musterbasierten Modellanalyse eingeführt und Ziele, Anforderungen und Voraussetzungen definiert. Ferner werden hier die verwandten Arbeiten aus Kapitel 3 mit dem vorliegenden Konzept verglichen.

Im Allgemeinen bieten Softwaremodelle dem Entwickler unterschiedliche Perspektiven auf das Softwaresystem und erleichtern verschiedene Aspekte von Entwurf und Anforderungen des Systems zu begreifen und zu erörtern. In diesem Kapitel wird ein Ansatz für eine kombinierte Analyse von Klassen-, Aktivitäts- und Sequenzdiagrammen vorgestellt. Durch diese Analyse werden Kontroll- und Datenfluss auf Parallelisierungspotenzial untersucht. Genauso wie Code können auch die Softwaremodelle implizite Parallelität beinhalten. Das sind diejenigen Stellen, die eine Parallelisierung ermöglichen, aber nicht durch spezielle Sprachkonstrukte explizit angegeben ist. Die implizite Parallelität (siehe Kapitel 2.3.1) wird anhand von speziellen Suchmustern erkannt, die in diesem Kapitel definiert werden. Eine zusätzliche Analyse von Datenabhängigkeiten hilft zu erkennen, ob eine Parallelisierung an dieser Stelle möglich und sinnvoll ist. Wenn das der Fall ist, wird das Suchmuster modifiziert und die Parallelität explizit dargestellt, indem Kontroll- und Datenfluss aufgesplittert werden.

Aus dem modifizierten Objektmodell werden abschließend neue parallele UML-Modelle erzeugt. Die Abbildung 33 stellt diesen Prozess schematisch dar:

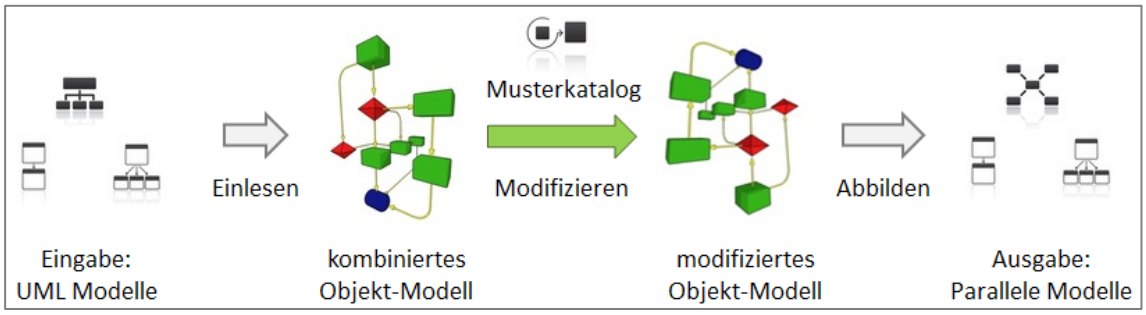


Abbildung 33: Schematische Abbildung des Ansatzes

In dieser Arbeit werden allgemeine, sowie selbst definierte Begriffe verwendet. Diese werden im Kapitel 4.1 definiert. Im Folgenden wird der Aufbau des abstrakten Objektmodells vorgestellt 4.3. Die implizierte Parallelität wird durch eine musterbasierte Analyse entdeckt. Dafür werden in Kapitel 4.4 spezielle Suchmuster definiert und formal beschrieben. Diese Suchmuster bilden einen Katalog von Such- und Ersetzungsregeln, die für spezielle Fälle oder Umgebungen auch erweitert werden können. Das nächste Kapitel 4.5 beschreibt die Transformationsregeln, mit deren Hilfe dieses Muster in ein anderes paralleles Muster transformiert werden kann und die Einschränkungen für jedes Muster. Dadurch wird das Objektmodell modifiziert und die Parallelität explizit dargestellt. Abschließend werden aus dem modifizierten Objektmodell die neuen parallelen UML-Modelle erzeugt 4.5. Eine prototypische Implementierung des Objektmodells und des Suchmusters werden in Kapitel 5 vorgestellt.

4.1 Begriffe und Definitionen

Im Folgenden werden Aufbau und grundlegende Konzepte dieser Arbeit vorgestellt. Dabei werden allgemeine, als auch selbst definierte Begriffe eingeführt.

4.1.1 UML-Elemente und Type

Die Grundbausteine eines Aktivitätsdiagramms (AD) sind die Aktionen (engl. *actions*). Die UML definiert einen Satz von elementaren Aktionen und teilt diese in mehrere Gruppen ein [UML2.3]. In dieser Arbeit werden alle Aktionen nach [Woh05b] in drei Gruppen aufgeteilt. Eine ähnliche Aufteilung findet man auch in anderen Arbeiten, wie zum Beispiel in [PF+04]. Wir werden folgende Definitionen für diese drei Gruppen verwenden:

- ***Invocation Actions*** sind Aktionen zum Aufrufen einer Operation oder zum Senden eines Signals. Ein **Signal** ist eine spezielle Aktion, die für eine asynchrone Kommunikation zwischen zwei Aktionen verwendet wird.

- ***Read and Write Actions*** sind Aktionen, die auf Objekte zugreifen und deren Zustand ändern können, bzw. deren Felder ändern. Mehr dazu wird in Kapitel 4.3 und 4.4 erklärt.

- ***Computation Actions*** sind Aktionen, die Eingabedaten bekommen, Berechnungen durchführen und die Ausgangswerte zurück liefern.

Außer Aktionen findet man in AD auch **Kontrollknoten**, die den Kontrollfluss beeinflussen [Woh05b]. Abbildung 34 zeigt die grafischen Notationen für alle diese Elemente. Die letzten vier sind für die Aufteilung und Zusammenführung des Kontrollflusses verantwortlich. Mehr dazu findet man in 2.2.3.

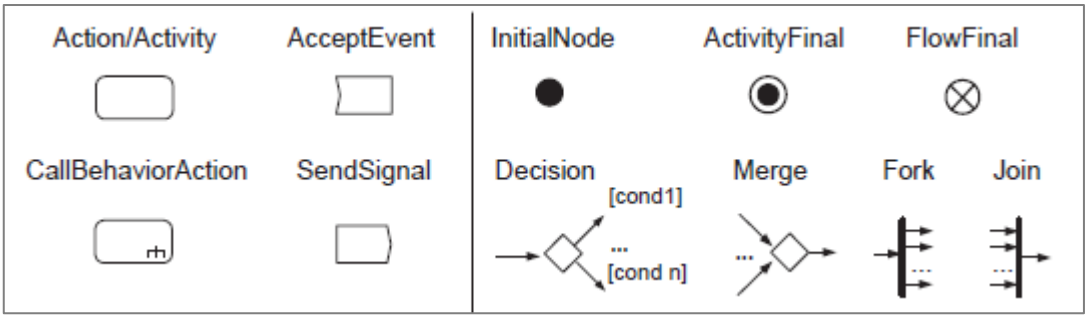


Abbildung 34. UML: Aktivitäten (links) und Kontrollflussknoten in AD aus [Woh05b]

Objektknoten und Pins

Wie man aus dem Grundlagenkapitel 2.2.3 bereits kennt, wird ein Objektfluss (engl. *object flow*) in der UML durch eine spezielle Kante dargestellt. In der UML 2.3 wird eine Aktion nicht direkt mit einem Objektknoten verbunden, sondern immer nur indirekt über Pins, die der Aktion zugeordnet sind [Stö05]. Das macht die Pins sehr nützlich für die Modellierung und Analyse verschiedener Datenflüsse und somit auch für die Bewertung des Parallelisierungspotenzials. Man unterscheidet zwischen In- und Output-Pins, die eingehenden und ausgehenden Datenfluss darstellen.

Folgende Abbildung zeigt einen Objektknoten (engl. *object node*), der mit einer Aktion durch einen Input-Pin verbunden ist. Der weitere Datenfluss zwischen zwei Aktivitäten wird mit zwei Pins modelliert.

Abbildung 35: Objektknoten und Pins in der UML

4.1.2 Objektmodell (AOM) und Suchmuster

Im Rahmen dieser Arbeit wird ein graphenbasierter Ansatz verwendet, um das kombinierte Objektmodell, das aus mehreren UML-Modellen besteht, zu erzeugen. Der Fokus liegt dabei auf die Zusammenarbeit zwischen den Objekten und Aktionen. Die Knoten im Graph repräsentieren die Aktionen (Methoden) sowie die aktiven und passiven Objekte (siehe 2.2.2) aus den UML-Modellen. Aktive Objekte leiten den Kontrollfluss und die passiven Objekte dienen dabei als Datencontainer. Die Kanten sind auch typisiert und beschreiben unterschiedliche Zugriffsarten, zum Beispiel, synchrone und asynchrone Zugriffe. Diese unterschiedlichen Kanten- und Knotentypen werden bei der Definition von Suchmustern und Transformationsregeln verwendet. Der Aufbau des abstrakten Objektmodells wird in dem Kapitel 4.3 ausführlich beschrieben.

Suchmuster

Fast in allen gut strukturierten Systemen finden sich viele Muster. Als Entwurfsmuster (engl. *design patterns*) werden bewährte Lösungsansätze in der Softwarearchitektur bezeichnet.
Suchmuster im Sinne dieser Arbeit sind Teilgraphe im AOM, die implizite Parallelität in sich tragen. Sie werden bei der Analyse des Modells verwendet und durch ihre parallelen Varianten ersetzt, die ein paralleles Architekturmuster (siehe 2.3.4) repräsentieren. Ein Suchmuster besteht aus einer Menge von Knoten und Kanten, die miteinander verbunden sind. Einige Knoten oder Kanten können mehrmals vorkommen und deren Anzahl gilt als Parameter dieses Suchmusters. So kann ein Suchmuster aus einen Knoten vom Typ A und aus theoretisch beliebig vielen Knoten vom Typ B bestehen. Abbildung 36 zeigt ein Suchmuster, das in dem einführenden Beispiel 2.1 vorkommt. Dieses Suchmuster besteht aus einer Aktion mit mehreren eingehenden Datenflüssen von anderen Aktionen (siehe Aktivitätsdiagramm links im Bild). Zwischen diesen Aktionen darf es keinen Datenfluss geben. Das Sequenzdiagramm zeigt die Reihenfolge der Aktionen und welche Objekte dabei zugegriffen werden.

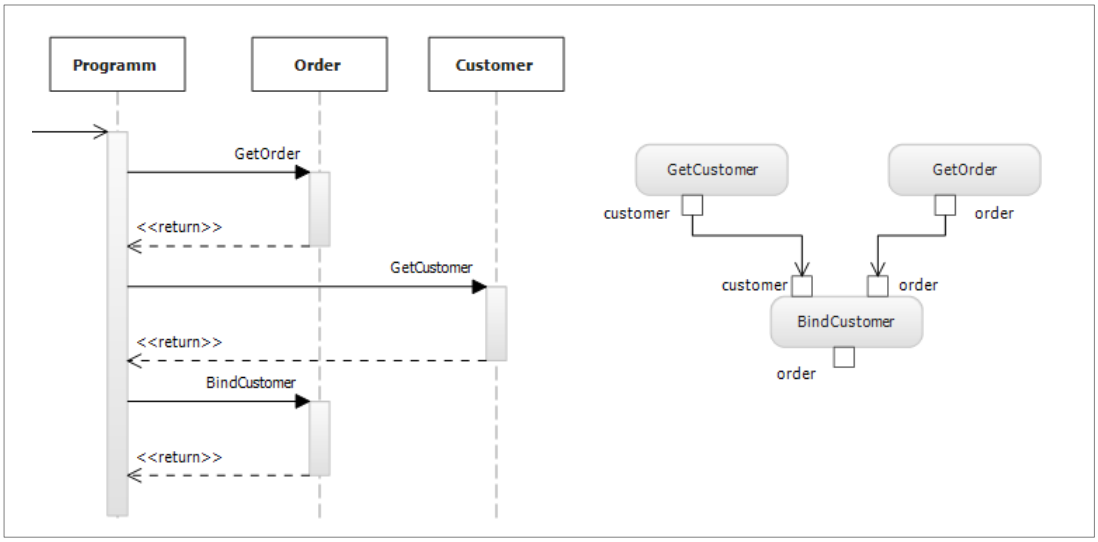


Abbildung 36: Suchmuster Beispiel

Dieses Suchmuster kann in einem Pseudocode folgendermaßen beschrieben werden:

```
1. Finde alle Aktionen mit Anzahl Input-Pins >= 2
2. Füge jeder Aktion eine Liste der Nachbaraktionen zu
3. Für alle Aktionen in der Liste:
4. Gibt es Datenflüsse zwischen Aktionen in dieser Liste,
   dann beide entfernen
5. Ist Anzahl der Listenelementen >= 2, dann Suchmuster gefunden
```

Such- und Ersetzungsregeln

In dieser Arbeit wird eine regelbasierte Modellmodifikation durchgeführt. Dafür wird eine Menge von Such- und Ersetzungsregeln definiert. Diese Regeln beschreiben Suchmuster und wie diese schrittweise geändert (transformiert) werden. Dabei kann ein Suchmuster durch mehreren Regeln beschrieben werden. Es werden auch allgemeine Hilfsregeln definiert, zum Beispiel, um alle eingehenden Datenflüsse für einen gegebenen Aktivitätsknoten zu finden. Diese Hilfsregeln können in mehreren Suchmustern verwendet werden. Analog zu Methoden in einem Programm können sie Eingabe- und Ausgabeparameter besitzen. Abbildung 37 zeigt die Komposition zwischen Suchmuster und Regeln.

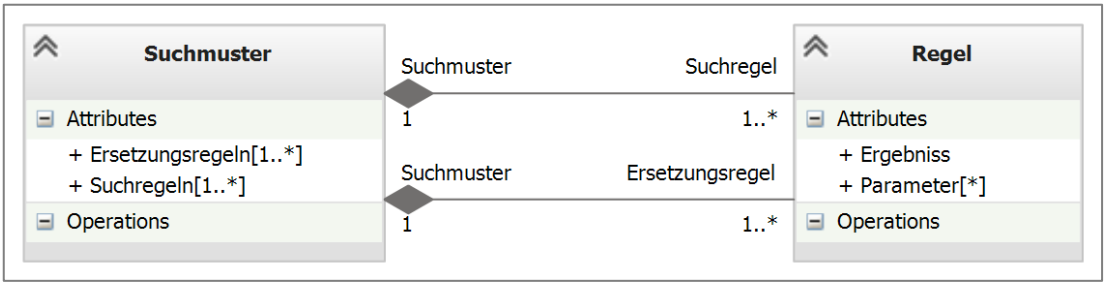


Abbildung 37: Such- und Ersetzungsregeln

Die Grundlagen für die Suchregeln werden in Kapitel 4.4 beschrieben. Die in dieser Arbeit definierten Suchmuster werden in Kapitel 4.5 formal beschrieben und graphisch dargestellt. In Kapitel 5 werden konkrete Implementierungen von Suchmustern mit Graphenersetzungssystem GrGen.NET vorgestellt.

4.2 Ziele und Anforderungen

Nachdem gerade ein Überblick über AOM und ein kleines Beispiel eines Suchmusters gegeben wurden, soll das Konzept formal definiert werden. Das Objektmodell soll mit Hilfe eines Suchmusterkatalogs modifiziert und für die Erzeugung von neuen parallelen Modellen verwendet werden. Im Folgenden werden die Anforderungen und Voraussetzungen dafür definiert.

4.2.1 Anforderungen

Um das Ziel dieser Arbeit, nämlich die Erkennung des Parallelisierungspotenzials auf Modellebene zu erreichen, werden folgende Anforderungen definiert:

Anforderung A_1: Es soll ein **eigenes abstraktes Objektmodell (AOM)** für die Analyse benötigte Kombination gewisser Daten aus vorliegenden Modellen entwickelt werden. Damit wird eine Abstraktion und Unabhängigkeit von den vorliegenden Modellen und deren Implementierung erreicht. Das Einlesen und Erzeugen von UML-Modellen wird durch eine von der Mustersuche unabhängige Schnittstelle implementiert. Der Ansatz kann also mit unterschiedlichen UML-Formaten (wie XMI oder Microsoft UML) arbeiten, wie die Implementierung später zeigen wird. Außerdem wird damit automatisch der Suchraum reduziert, indem man mehrere Modelle in einem verbindet und nur gewisse Teile aus allen Modellen nimmt. Kapitel 4.2 beschreibt den Aufbau des Objektmodells ausführlich.

In dieser Arbeit wird der Ansatz für drei UML-Diagrammtypen implementiert. Für den Aufbau des Objektmodells werden Daten aus Klassen-, Aktivität- und Sequenzdiagrammen verwendet. Klassendiagramme beschreiben dabei die statischen Strukturen – Klassen, Schnittstellen (engl. *interface*) und Assoziationen und bilden somit die Grundlage des Objektmodells. Aktivitätsdiagramme stellen die Objekte und Methoden graphisch dar, und bilden Daten- und Kontrollflüsse ab. Die Sequenzdiagramme beinhalten eine zeitliche Anordnung der Abläufe im Softwaresystem und vervollständigen somit die Aktivitätsdiagramme. Diese Arbeit berücksichtigt auch, dass manche Informationen in verschiedenen Modellen angegeben sein können, wie zum Beispiel Verzweigungen. Mehr dazu in Kapitel 4.1.2. Die genaue Spezifikation der erwähnten Schnittstellen und des Objektmodells wird in Kapitel 5 gegeben.

Anforderung A_2: **Musterbasierte Analyse**. Um implizite Parallelität in Modellen zu entdecken, wird eine musterbasierte Analyse durchgeführt. Dafür wird ein Katalog von den Suchmustern für das Objektmodell entworfen. Diese Suchmuster werden formal beschrieben und durch Suchregeln definiert. Es können auch allgemeine Regeln bzw. Hilfsmethoden definiert werden, zum Beispiel für die Analyse des Kontrollflusses. Sie können bei der Definition von unterschiedlichen Suchmustern verwendet werden, zum Beispiel um die Anzahl der eingehenden oder ausgehenden Kanten zu berechnen. So kann der Musterkatalog durch neue Muster jederzeit erweitert werden und der Ansatz sehr flexibel und leicht erweiterbar bleiben. Einen musterbasierten Ansatz findet man in vielen wissenschaftlichen Arbeiten, die sich mit der Analyse von UML-Modellen beschäftigen.

In [Woh05b] werden verschiedene Muster für die UML Aktivitätsdiagramme definiert und mit ähnlichen Muster, die in Sprache YAWL definiert sind, verglichen 3.1.1. Dabei werden Vor- und Nachteile, sowie die Mächtigkeit der UML untersucht. Die Ergebnisse finden man in Tabelle 1 in Kapitel 3. In [Stö05] werden UML-Aktivitätsdiagramme in Petri-Netze transformiert und deren Datenfluss wird mit Standardverfahren für die Petri-Netze auf Korrektheit überprüft 3.1.2. Hier werden die Daten aus den Klassendiagrammen mit den Daten aus Aktivitätsdiagrammen kombiniert. Dadurch kann der Datenfluss besser beschrieben werden, weil man auch den Aufbau der Objekte und deren Eigenschaften und Methoden kennt. Analog werden in dieser Arbeit Daten aus mehreren UML-Modellen in einem Objektmodell kombiniert. Dabei bezieht sich ein Suchmuster auf die Daten aus mehreren Modellen und nicht nur aus einem wie in [Woh05b]. Die Umsetzung dieser Anforderung erfolgt in Kapitel 4.4.

Anforderung A_3: Regelbasierte Modellmodifikation. Das Objektmodell soll unter Anwendung von Suchmustern und Ersetzungsregeln modifiziert werden. Jedes Suchmuster soll einen Satz von Ersetzungsregeln haben, die schrittweise den Modifikationsprozess beschreiben. Durch die Modifikation bekommt das Objektmodell neue Knoten, die den Kontrollfluss aufspalten und neue Kanten, die parallele Daten- und Kontrollflüsse abbilden. Somit wird die Parallelität explizit dargestellt. Wenn eine explizite Parallelität in den UML-Modellen bereits vorhanden ist, wird sie nicht verändert, sondern es wird versucht weitere Stellen mit Parallelisierungspotenzial zu finden. Die Ersetzungsregeln werden in Kapitel 4.5 genauer beschrieben.

Anforderung A_4: Automatische Erzeugung paralleler UML-Modelle aus dem modifizierten Objektmodell. Der Benutzer soll eine Wahlmöglichkeit haben, ob das ursprüngliche Modell direkt modifiziert oder ein komplett neues Modell erzeugt wird. Diese UML-Modelle werden eine explizite Darstellung der parallelen Muster (siehe 2.3.4), sowie zusätzliche Notationen für den Softwareentwickler beinhalten. Die Notationen können in Form von Kommentaren direkt in das UML-Modell eingebaut werden. Außerdem soll der gesamte Modifikationsprozess in einer Log-Datei schrittweise gespeichert werden.

4.2.2 Voraussetzungen

In diesem Abschnitt werden die Voraussetzungen definiert, die aus verschiedenen Gründen für die Analyse und die spätere Parallelisierung notwendig sind.

Konsistenz und Widerspruchsfreiheit der Modelle

Das UML-Metamodell definiert neben den Modellelementen auch Regeln zur Verwendung der Modellelemente und die daraus resultierende Semantik und Bedingungen, welche die Verwendungsmöglichkeiten auf Grund der Spezifikation der Modellelemente einschränken. Zum Beispiel werden Klassen durch ihre Attribute und Operationen spezifiziert und durch Assoziationen miteinander verbunden. Dabei sollen bestimmte Bedingungen, wie eine mehrfache Deklaration eines Attributs in einer Klasse und einer Oberklasse, nicht verletzt werden.

Ein Gegenbeispiel: Im folgenden Beispiel (Abbildung 38) werden Modellelemente angezeigt, deren Existenz die Existenz anderer Modellelemente voraussetzt. Die folgende Abbildung zeigt ein Klassendiagramm aus den zwei Klassen *Customer* und *Order*. Die Klasse *Order* besitzt eine Methode *BindCustomer*, die als Parameter *customerID* vom Typ Integer erhält. Diese Methode verbindet eine Bestellung mit einem Kunden anhand seiner ID-Nummer. Wenn die Aktion *GetCustomer* eine andere Aktion *BindCustomer* aufruft und ein Objekt als Parameter übergibt, so wird vorausgesetzt, dass das Objekt auf einer existierenden Klasse basiert und die aufgerufene Methode den entsprechenden Parameter hat.

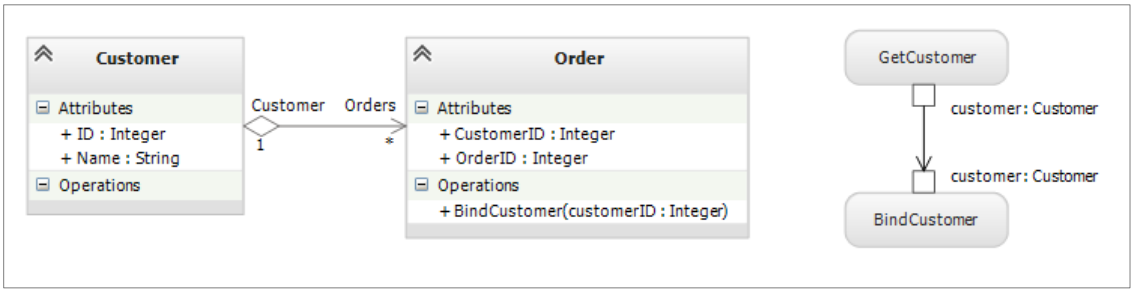


Abbildung 38: Inkonsistenz in den UML-Modellen

Das daneben dargestellte Aktivitätsdiagramm zeigt ein anderes Bild. Dort ist ein Datenfluss zu sehen und die Aktion *BindCustomer* bekommt als Eingabe ein Objekt vom Typ *Customer*. Solche Inkonsistenzen können dazu führen, dass das Objektmodell nicht korrekt aufgebaut wird und als Folge alle weiteren Modifikationen falsch werden.

Alle vorliegenden Modelle müssen also konsistent sein: Methoden und Objekte müssen gleiche Namen und Parameter haben, sowie gleiche Attribute in allen Diagrammen besitzen. Die in den Verhaltensdiagrammen (AD und SD) beschriebenen Aktivitäten müssen in den Klassendiagrammen als Methoden bestimmter Klassen definiert werden. Das Gleiche gilt für die Datentypen. Dies ist kritisch für den Aufbau des abstrakten Objektmodells, denn dort werden alle diese Elemente miteinander verbunden. Fällt ein Objekt oder eine Methode aus, so kann der Kontroll- bzw. Datenfluss nicht mehr korrekt dargestellt und analysiert werden. Als mögliche Folge werden einige Muster nicht erkannt, bzw. Muster werden fälschlicherweise als positiv bewertet. Die Widerspruchsfreiheit bedeutet, dass gleiche Elemente (Objekte, Methoden) in verschiedenen Diagrammen gleiche Attribute und Beziehungen zu den anderen Elementen haben müssen.

Der in dieser Arbeit entwickelte Ansatz ermöglicht die Bearbeitung von mehreren Modellen als ein Ganzes. So kann beispielsweise ein großes Klassendiagramm mit zwei Aktivitäts- und zwei Sequenzdiagrammen in ein Objektmodell eingelesen werden. Es ist dabei zu beachten, dass durch Einlesen und Parsen von UML-Modellen in dem Objektmodell mehrere separate Graphen gebildet werden können, wenn die Modelle keine gemeinsamen Elemente besitzen. In diesem Fall werden im Objektmodell mehreren Graphen gebildet. Diese stellen offenbar unterschiedliche Teile des Softwaresystems dar und werden daher separat betrachtet und modifiziert. Zusätzlich dazu kann der Benutzer selbst entscheiden, ob die Modelle zusammen oder separat analysiert und transformiert werden sollen und wie sie im Softwaresystem zeitlich und räumlich zueinander stehen.

Keine syntaktischen Überprüfungen in Aktivitätsdiagrammen

Beim Einlesen von Aktivitätsdiagrammen wird nicht überprüft, ob der Objektfluss von einem Output-Pin zu einem Input-Pin gültig ist oder der Ausdruck eines Wächters für die darin verwendeten Typen definiert ist. In der UML ist es erlaubt, dass die Ein- und Ausgangsparameter nicht vom selben Typ sind, solange sie ineinander konvertierbar sind [UML2.3]. In dieser Arbeit wird davon ausgegangen, dass diese statischen, syntaktischen Überprüfungen schon im Vorfeld bei der Erstellung von UML-Modellen stattgefunden haben.

Keine Schleifen zwischen Kontrollknoten

Die Suchregeln zur Analyse von Kontroll- bzw. Objektflüssen erkennen die Rücksprünge von einem Kontrollknoten zum anderen Kontrollknoten nicht. Ein Rücksprung von einem Verzweigungsknoten ist somit nur zu einer Aktion erlaubt. Mehr dazu findet man in der Definition des Objektmodells im Kapitel 5.4. Die Abbildung 39 zeigt eine korrekte Darstellung einer Schleife in UML.

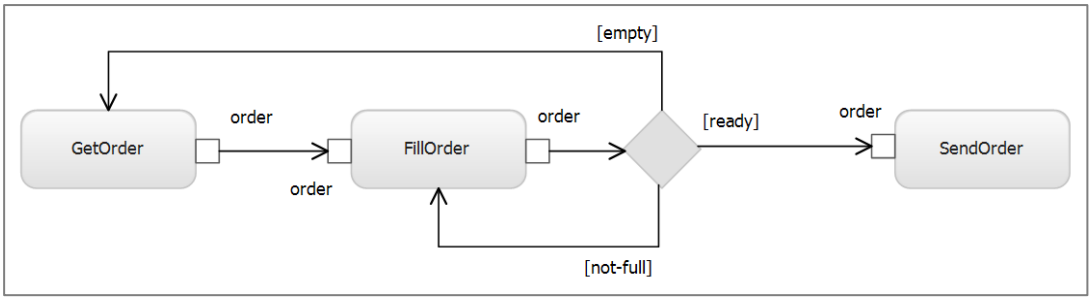


Abbildung 39. UML: Korrekte Darstellung von Schleifen

Der Schleifenkörper kann als separate Aktion mit stereotyp «loop» dargestellt werden.

Zeitereignisse werden nicht unterstützt

Ein Zeitereignis (engl. *time event*) steht in der UML für Ereignisauftritte, die zu einem bestimmten Zeitpunkt eintreffen [UML2.3]. Zeitereignisse werden in dieser Arbeit nicht berücksichtigt. Dies liegt daran, dass die entwickelte Analyse statisch ist und die zeitlichen Aspekte sowie die Interaktion mit Benutzer nicht betrachtet werden können.

Höchstens eine Ausführung pro Aktivität

Das Verhalten eines Softwaresystems wird in UML AD als Verhalten von kooperierenden Instanzen aktiver Klassen 2.2.2 modelliert. Diese Instanzen heißen aktive Objekte. In der UML gibt es eine Möglichkeit, die Anzahl paralleler Ausführungen einer Aktion zu beschränken [UML2.3]. Dies wird über das Attribut *isSingleExecution* der UML-Metaklasse *Activity* geregelt, welches in dieser Arbeit standardmäßig als *true* angenommen wird. Für jede Instanz einer aktiven Klasse wird im Objektmodell also genau eine Instanz der ihr zugeordneten Aktivitäten erzeugt. Wenn bei der Parallelisierung mehrere Instanzen einer Aktion parallel ausgeführt werden sollen (Replikation), müssen sie auch explizit dargestellt sein, wie in Abbildung 40 gezeigt ist.

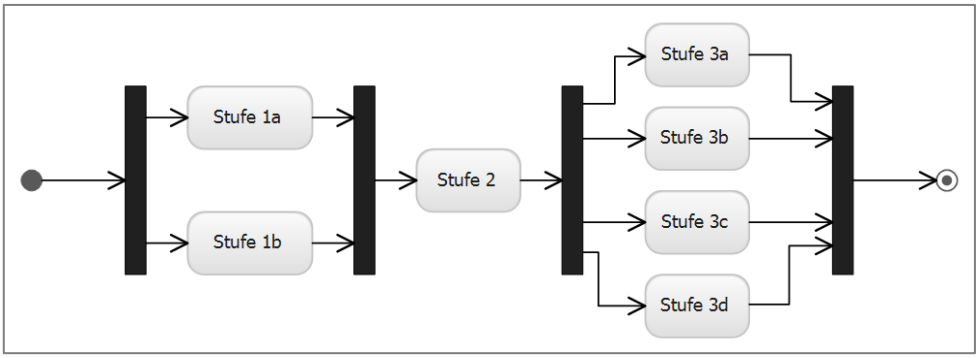


Abbildung 40: Dreistufiges Pipeline in AD

Als Beispiel dafür kann man eine replizierbare Stufe in einem Pipeline-Muster 2.3.4 nennen. Abbildung 40 zeigt eine dreistufige Pipeline mit zwei replizierbaren Stufen. Jede Replikation einer Stufe wird dabei explizit als eine Aktion dargestellt.

4.3 A1: Aufbau des abstrakten Objektmodells (AOM)

Nachdem im vorherigen Kapitel die Ziele und Anforderungen dieser Arbeit definiert wurden, wird nun der Aufbau des kombinierten Objektmodells (Anforderung A_1) beschrieben. Zunächst wird das Metamodell vorgestellt, das in dieser Arbeit verwendet wird und die Elemente aus allen drei UML-Modellen verbindet. Danach werden Aufbau des Objektmodells und deren Elemente beschrieben. Zum Schluss wird die graphische Notation des Objektmodells vorgestellt, die in dieser Arbeit verwendet wird.

4.3.1 Das Metamodell zu AOM

In dieser Arbeit wird folgendes Metamodell als eine Erweiterung des Metamodells eines Aktivitätsdiagramms entwickelt. Es bildet die Grundlage für den Objektmodellgraphen. Hier werden die Grundtypen von Knoten und Kanten in dem Objektmodell definiert. Eine grafische Darstellung findet man in folgenden Kapiteln. Die Abbildung 41 zeigt einen Ausschnitt aus dem Metamodell:

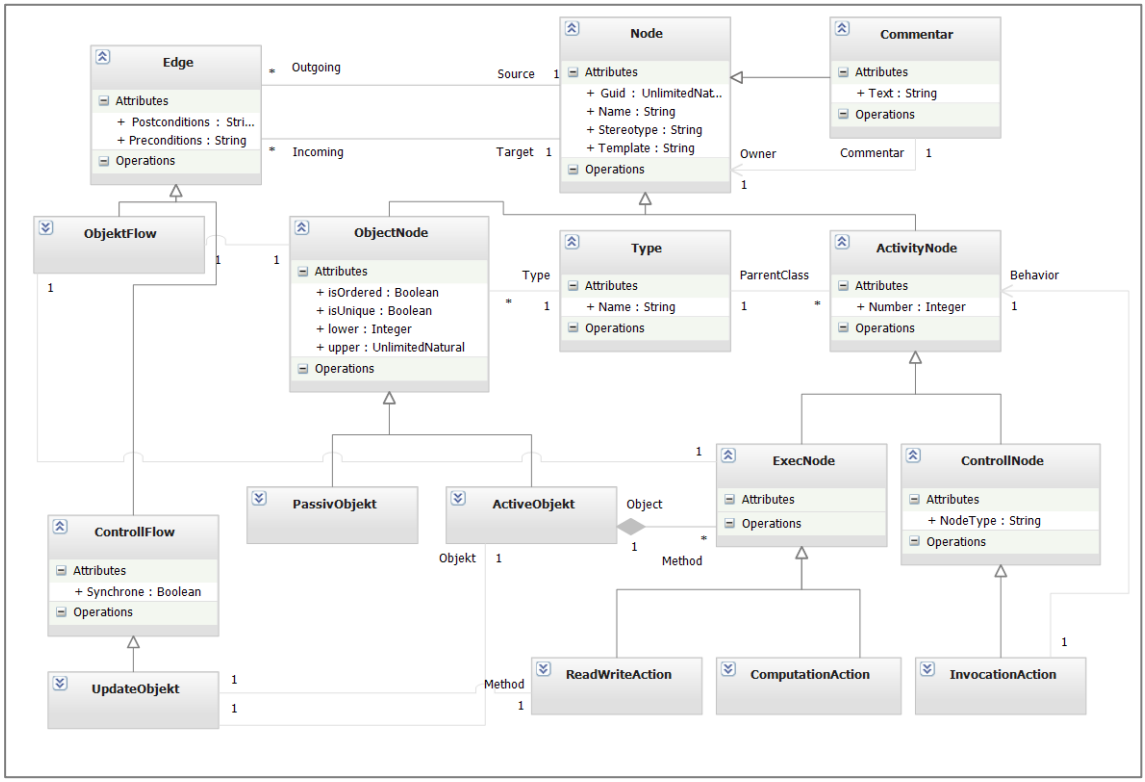


Abbildung 41: Das Metamodel des kombinierten Objektmodells (AOM)

Knoten

Das Objektmodell AOM ist ein typisierter gerichteter Multigraph. Alle Knoten erben von der Knotenklasse *Node*, von dem die weiteren Knotenklassen *ObjectNode* und *ActivityNode* erben. Alle Knoten verfügen über weitere allgemeine Attribute, wie *Guid*, *Name* und *Typ*, die aus dem UML-Modell kopiert werden. Jeder *ObjektNode* ist mit einem Typknoten verbunden, so wie die Objektknoten in der UML. Der Objektknoten in UML ist mit einem Attribut *Multiplicity* ausgestattet, das die zulässige Anzahl der Kardinalitäten für eine Instanziierung dieses Elements angibt. In der UML ist dieses Attribut durch *MultiplicityElement* beschrieben (siehe [UML2.3] in §§ 9.12.1) und hat folgende Eigenschaften, die in dem Objektmodell direkt übernommen wurden:

- ***isOrdered: Boolean***
  Gibt an, ob die Reihenfolge der Elementen eine Rolle spielt. Der Standardwert ist *false*. Wenn dieses Attribut gleich *true* ist, dann kann die Parallelisierung erschwert oder sogar nicht möglich sein, da die Objekte in eine bestimmte Reihenfolge bearbeitet werden müssen.

- ***isUnique: Boolean***
  Gibt an, ob die Werte in einer Instanziierung dieses Element eindeutig sind. Der Standardwert ist *true*. Dieses Attribut kann verwendet werden, um solchen Muster wie *Singelton* zu erkennen. Wird in dieser Arbeit jedoch nicht berücksichtigt.

- ***lower: Integer***
  Gibt die untere Grenze des Multiplizitäts an. Standardwert ist 1. Dieses Attribut zeigt, ob es sich um ein Objekt oder eine Menge geht, die parallel verarbeitet werden kann. Wenn dieses Wert klein ist (meistens = 1), dann macht die Parallelisierung keinen Sinn, da es nur um ein Objekt geht.

- ***upper : UnlimitedNatural***
  Die obere Grenze des Multiplizitäts. Standardwert ist 1. Dieses und das obere Attribut werden bei einigen Suchmustern (z. Bsp. *Producer-Consumer 4.5.4*) als Parameter verwendet.

Analog zur UML 2.2.2 werden alle Objektknoten in zwei weitere Typen aufgeteilt: passive und aktive Objekte. Diese Unterteilung findet anhand der *isActive*-Attribute im Klassendiagramm statt. Passive Objekte spielen eine wichtige Rolle in dem Datenfluss. Instanzen der aktiven Klassen dagegen besitzen mehrere Methoden und leiten den Kontrollfluss. Die Nebenläufigkeit wird in der UML hauptsächlich durch mehrere aktive Objekte dargestellt, die unabhängige Threads oder Tasks repräsentieren. Da in dieser Arbeit vorausgesetzt wird, dass alle Aktivitäten als Methoden im Code zu betrachten sind, so gehört jede *ActivityNode* auch zu einer Klasse (*Type*). Im Objektmodell soll der zeitliche Aspekt aus der Sequenzdiagramm nicht verloren gehen. Deswegen besitzen alle Aktivitätsknoten einen Attribut *Number*, der die Ausführungsreihenfolge angibt. So wie in [Stö05] werden die Aktivitätsknoten in zwei Gruppen aufgeteilt: Kontrollknoten (*ControllNode*) und Berechnungsknoten (*ExecutableNode*). Wie in Kapitel 4.1 schon erwähnt wurde, werden in dieser Arbeit alle Aktionen nach [Woh05b] in drei Gruppen aufgeteilt: ***Invocation Actions***, ***Read-Write Actions*** und ***Computation Actions*** 4.1.2. Wie die Aktionen auf entsprechenden Graphenknoten abgebildet werden ist im nächsten Kapitel beschrieben.

Ein spezieller Knotentyp ist der **Kommentar-Knoten**. Dieser Knotentyp wird zum einen für die Speicherung von in der UML bestehenden Kommentaren verwendet. Zum anderen können damit neue Kommentare hinzugefügt werden, die dem Entwickler helfen neue parallele Modelle besser verstehen zu können. Mehr über die Aktivitätsknoten in der UML und im Objektmodell findet man im weiteren Verlauf dieses Kapitels.

Kanten

In einem Graphen werden zwei Knoten immer durch eine oder mehrere Kanten verbunden. Durch die Kanten werden Relationen zwischen verschiedenen Knotentypen ausgedrückt. Im Objektmodell AOM gibt es zwei grundlegenden Kantentypen, die den Kontroll- und Datenfluss entsprechend repräsentieren. Im Gegensatz zu UML verbindet ein Datenfluss in AOM immer einen Aktions- und Objektknoten miteinander und nicht zwei Pins. Hier findet eine Spaltung des Kontroll- und Datenflusses statt. Dadurch können beide Flüsse separat betrachtet und analysiert werden. Ein Kontrollfluss verbindet dabei immer zwei Aktivitätsknoten (*ActivityNode*). Folgende Abbildung zeigt den Unterschied zwischen den Darstellungen des Datenflusses in den UML-Modellen und in dem Objektmodell.

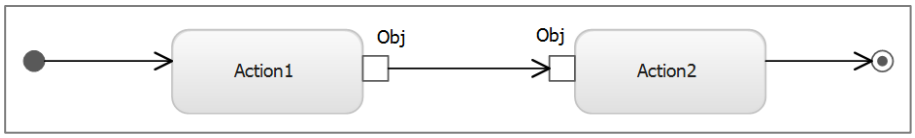


Abbildung 42. UML: Ein Objektfluss zwischen zwei Aktionen

Ein Objektfluss zwischen zwei Aktivitäten im abstrakten Objektmodell (AOM) zeichnet sich dadurch aus, dass der Objektknoten immer explizit dargestellt wird. In den folgenden Abschnitten wird gezeigt, dass der Unterschied zwischen Daten- und Kontrollfluss in UML nicht immer klar ist und erst aus dem Kontext verständlich wird. Durch die explizite Darstellung wie in Abbildung 43 umgeht AOM dieses Problem.

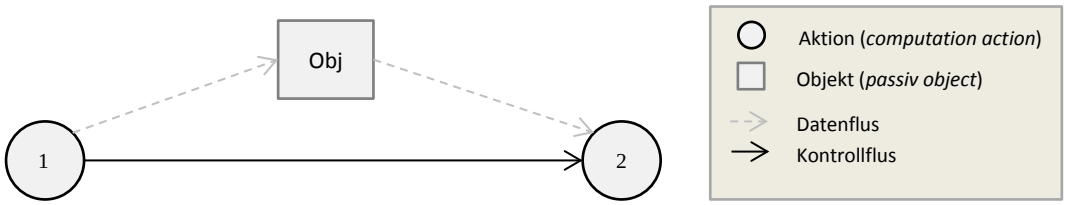
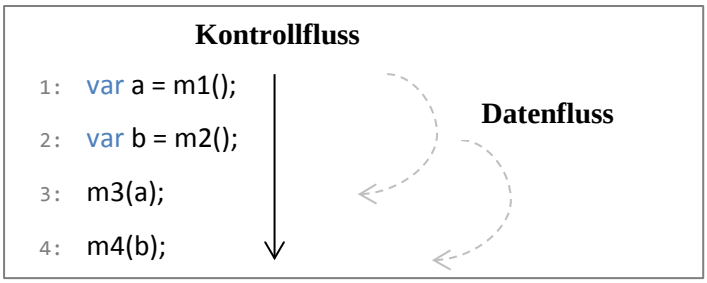


Abbildung 43. Kontroll- und Datenflüsse in UML und AOM

In nebenläufigen Systemen kann die Zuordnung von Nachrichten zu einem bestimmten Kontrollfluss durch Vergabe von Nachrichtennummern erfolgen [Vol05]. Dieses Konzept wird übernommen und verwendet zum Unterscheiden der Ausführungsreihenfolgen in AOM. Diese Nummern werden im Attribut *Number* gespeichert. In dieser Arbeit werden Aktionsknoten in AOM immer mit ihrer Nummer dargestellt, damit die Reihenfolge klar wird.

Die Tatsache, dass der Datenfluss nicht immer mit dem Kontrollfluss übereinstimmt lässt sich leicht an folgendem Code-Beispiel demonstrieren. Zunächst werden zwei Objekte *a* und *b* mit Hilfe der Methoden *m1* und *m2* erzeug (Zeilen 1, 2). Danach werden sie als Parameter in der Methoden *m3* und *m4* jeweils übergeben (Zeilen 3,4). Wie man sieht, ist der Kontrollfluss hier linear und die Daten fließen jeweils zwischen Statement 1-3 und 2-4:



UpdateObjekt-Kante

Ein spezieller Fall von Kanten ist die *UpdateObjekt*-Kante. Damit wird eine Interaktion zwischen einem aktiven Objekt und seiner Methode gezeigt, die den Zustand dieses Objektes ändert. Dabei handelt es sich immer um synchrone Operationen, die in vielen Suchmustern eine wichtige Rolle spielen 4.5.

Assoziation

Eine Assoziation verbindet zusammengehörige Elemente, damit sie bei einer späteren Generation von neuen UML-Modellen verbunden bleiben. So kann etwa einen Kommentarknoten mit einem Objekt- oder Aktionsknoten verbunden werden. Diese Kanten beschreiben keinen Fluss, sondern spielen eine semantische Hilfsrolle. Aus diesem Grund werden sie auch nicht visualisiert.

4.3.2 Modellelemente

Wie in Kapitel 2.2.1 bereits beschrieben bestehen alle UML-Modelle aus Elementen und Beziehungen zwischen diesen. Beim Einlesen eines UML-Modells werden alle Elemente auf die entsprechenden Elemente des Objektmodells AOM abgebildet werden. Im Folgenden wird das Mapping zwischen UML-Diagrammen und den Elementen des Objektmodells vorgestellt. Die wichtigsten Klassen und deren Eigenschaften für die kombinierte Modellanalyse werden hier detailliert beschrieben.

Abbildung von Aktivitätsdiagrammen auf AOM

Ein Aktivitätsdiagramm beschreibt die dynamischen Aspekte eines Systems als Reihe von Aktivitäten [UMLWiki]. Die Verbindungpfeile stellen dar, wie die Kontrolle nacheinander von einer Aktion an die nächste übergeben wird. Eine Aktion kann nur dann gestartet werden, nachdem die vorherige Aktion abgeschlossen wurde. Die Folge von Aktionen kann auch Verzweigungen und Schleifen beinhalten 2.2.3, die für eine Analyse von besonderer Bedeutung sind. Die Eingabe- und Ausgabe-Pins beschreiben den Datenfluss zwischen Aktionen und definieren den Type der Daten, der durch ein Klassendiagramm konkretisiert werden kann. Für die parallelen Daten- und Kontrollflüsse, die später durch Modelltransformation entstehen werden, sind die Gabelung- und die Parallelisierungsknoten sehr wichtig. Sie beschreiben Orte, an denen Parallelität angebracht werden kann. Mehr dazu in Kapitel 4.4.1.

Für den Aufbau des Objektmodells wird in dieser Arbeit das Metamodell eines Aktivitätsdiagramms aus [Stö05] als Grundlage verwendet (Abbildung 44) und durch zusätzliche Elemente aus Sequenz- und Klassendiagrammen erweitert. In den letzten Jahren wurde das Aktivitätsmodell in UML erheblich überarbeitet. Während beispielsweise die Aktivitäten der UML 1.x eine spezielle Form der Zustandsautomaten waren, haben die Aktivitäten nun keine Beziehung mehr zu den Automaten. Sie entsprechen viel mehr der Semantik der Petri-Netze und unterstützen wesentlich besser die Ablaufsemantik [Stö05]. Das vollständige Modell ist im Anhang zu finden.

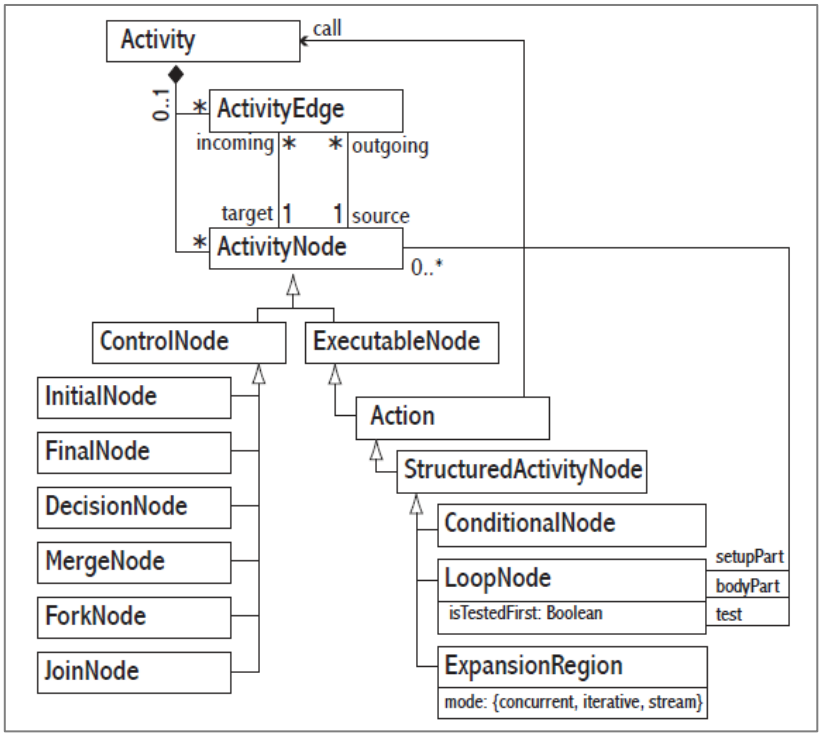


Abbildung 44: Metamodell eines Aktivitätsdiagramms aus [Stö05]

Aus der obigen Abbildung folgt, dass eine Oberklasse *Activity* mehrere *ActivityEdge* (Kanten) und *ActivityNodes* (Knoten) haben kann, die sich in *ExecutableNodes* und *ControlNodes* aufspalten. Außerdem werden hier die Vererbungshierarchie und die Attribute der Elemente definiert. Wie in Kapitel 2 bereits beschrieben wurde, können in einem Aktivitätsdiagramm sowohl Kontroll- als auch Objekt- bzw. Datenflüsse zwischen Aktionen beschrieben werden. Zusätzlich zu dieser allgemeinen Art von Aktionen stehen Spezialisierungen zur Verfügung, wie z. B. Signale, die für eine asynchrone Kommunikation verwendet werden können.
Die Semantik eines Aktivitätsdiagramms basiert auf einem Tokenfluss, der zwischen Aktionen über den Kanten fließt 2.2.3. Eine Aktion bekommt einen Token auf eingehenden Kanten, wird dann ausgeführt und bietet auf den ausgehenden Kanten Token an. Der Kontrollfluss und der Objektfluss werden entsprechend durch Kontroll-Token und Objekt-Token geleitet. Die Unterscheidung ist im Diagramm nicht direkt ersichtlich, sondern ergibt sich aus dem Kontext. Das erschwert eine musterbasierte Analyse der Modelle. Aus diesem Grund wurden die Aktivitätsdiagramme mit den Sequenzdiagrammen kombiniert. Die ausführliche Beschreibung der Syntax und der Semantik findet man in der Spezifikation [UML2.3]. Folgende Abbildung aus [Stö05] zeigt eine vereinfachte Form des Metamodells eines UML-Aktivitätsdiagramms. Das vollständige Modell ist in Anlage A zu finden.

Die Aktivitäten können einen oder mehrere Input- und Output-Pins für Objektflüsse besitzen. Für die Analyse des Datenflusses müssen zusätzlich Objekt-Token, die durch Ein- und Ausgabe-Pins fließen, berücksichtigt werden. Im Folgenden wird die Abbildung der Elemente eines Aktivitätsdiagramms auf die Knoten im Objektmodell AOM beschrieben. Abbildung 46 beinhaltet alle wichtigen Elemente, die in dieser Arbeit berücksichtigt und auf AOM abgebildet werden:

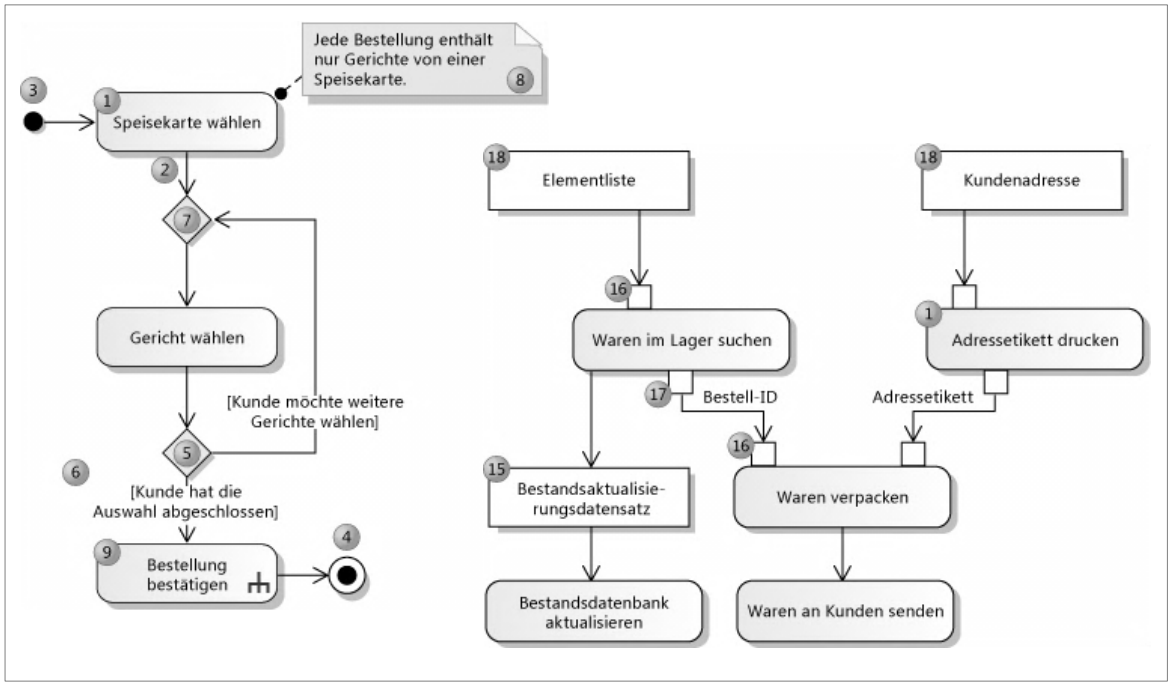


Abbildung 45. UML: Elemente eines Aktivitätsdiagramms aus [MSDN_AD]

Folgende Tabelle beschreibt die UML Elemente, die in der Abbildung 45 zu sehen sind, und die entsprechende Elemente des Objektmodells:

ID	UML Element	Bedeutung	AOM Element
1	Aktivität	Ein Schritt der Aktivität. Zum Beispiel eine Methode.	`Node:ActivityNode`
2	Kontrollfluss	Eine Kante, die den Kontrollfluss zwischen Aktionen darstellt.	`Edge:ControllFlow`
3,4	Start/Endknoten	Gibt die erste bzw. die letzte Aktion an. Die Startaktion darf nur einmal vorkommen.	`Node:ControllNode`
5-7	Entscheidungs- und Zusammenführungsknote	Eine bedingte Verzweigung und Zusammenzuführung in dem Kontrollfluss.	`Node:ControllNode`
16-17	Datenfluss mit Pins	Stellt Daten dar, die im Fluss übertragen werden. Die Pins definieren Daten, die eine Aktion erzeugt/empfängt. ***Type*** – Der Typ der übertragenen Objekte.	`Edge:ObjektFlow` `Node:PassivObject`
8	Kommentar	Stellt weitere Informationen zu Elementen bereit, mit denen er verknüpft ist.	`Node:Comment`
18	Objektknoten	Daten in einem Datenfluss	`Node:ObjectNode`
9	Aktion zum Aufrufen eines Verhaltens	Eine Aktion, die in einem anderen Diagramm ausführlicher definiert ist. ***Behavior*** – Die aufgerufene Aktivität.	`Node: InvocationAction`

Tabelle 3: Mapping von Aktivitätsdiagrammen

Abbildung von Sequenzdiagrammen auf AOM

Das Sequenzdiagramm (SD) ist ein Verhaltensdiagramm und es zeigt die dynamischen Aspekte des modellierten Systems. Ein SD repräsentiert die Interaktion und den Austausch von Nachrichten zwischen Lebenslinien (engl. *lifelines*). Diese Lebenslinien repräsentieren die Instanzen der aktiven Klassen. Das modellierte System kann also als Netzwerk (Graphen) von aktiven Objekten betrachten, die untereinander Nachrichten austauschen. Die Objekte sind typisiert und somit hat man eine implizite Verbindung mit dem Klassendiagramm. Dadurch kann man die Informationen aus beiden Modellen miteinander verbinden und im AOM gemeinsam betrachten. Setzt man voraus, dass die Nachrichten gleich Aktivitäten in AD sind und die Parameter den Pins entsprechen, so bekommt man nicht nur räumliche (AD), sondern auch die zeitliche Sicht auf das modellierte System.

In der UML wird zwischen drei Nachrichtentypen unterschieden, die durch unterschiedliche Pfeile dargestellt werden (weitere Varianten wären als Erweiterungen von UML zulässig):

- **synchroner Aufruf** (Kontrollfluss): Die Nachricht wird als Prozeduraufruf interpretiert. Der Sender wartet auf die Antwort des Empfängers.

- **asynchroner Aufruf** (Kontrollfluss): Die Nachricht wird als Signal 4.1.1 betrachtet, und der Sender der Nachricht wartet nicht auf die Antwort des Empfängers

- **unspezifizierter Kontrollfluss:** wird in dieser Arbeit immer als synchrone Kontrollfluss betrachtet.

Im AOM werden synchrone und asynchrone Aufrufe unterstützt und in einigen Suchmustern verwendet. Bei synchronen Nachrichten wird die Kontrolle an das gerufene Objekt übergeben (siehe passive und aktive Objekte 2.2.2). Asynchrone Aufrufe geben sofort die Kontrolle an das rufende Objekt zurück. Folgende Abbildung stellt alle SD-Elemente dar, die von AOM unterstützt werden:

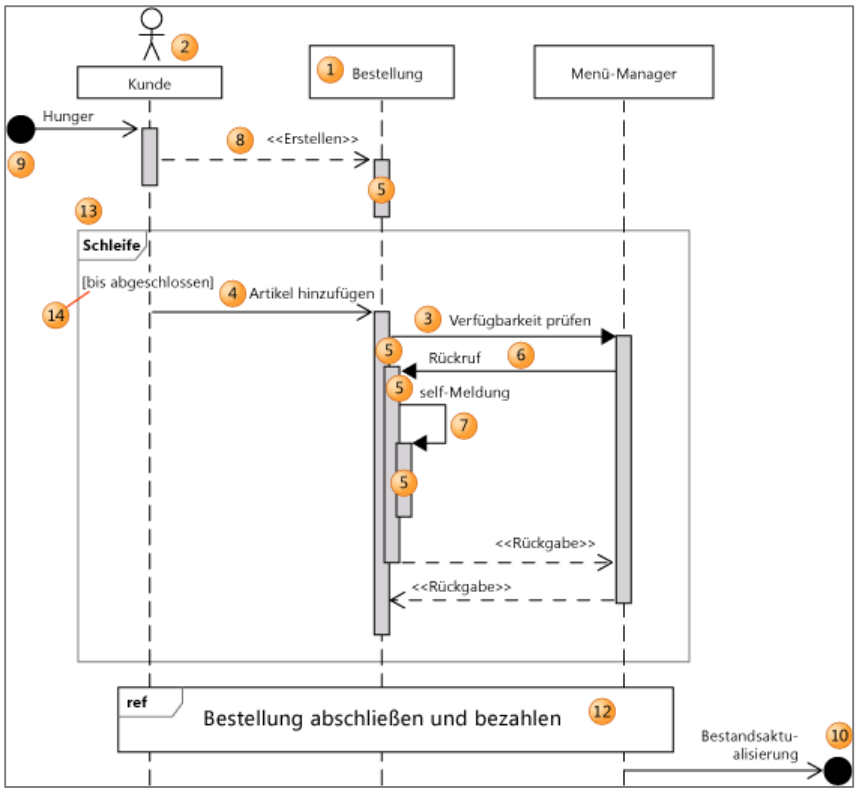


Abbildung 46. UML: Elemente eines Sequenzdiagramms aus [MSDN_SD]

Folgende Tabelle beschreibt die Abbildung der Elemente einer SD auf die entsprechenden Elemente des Objektmodells:

ID	UML Element	Bedeutung	AOM Element
1	Lebenslinie	Zeigt eine Sequenz von Ereignissen, die zwischen verschiedenen Teilnehmer auftreten.	`Node:AktiveObjekt`
3	Synchrone Nachricht	Zeigt einen Aufruf und die Rückgabe an. Die gewöhnlichen Methodenaufrufe innerhalb eines Programms.	`Edge:ControllFlow` *`synchrone=true`*
4	Asynchrone Nachricht	Asynchrone Kommunikation zwischen separaten Threads oder der Erstellung eines neuen Threads.	`Edge: ControllFlow` *`synchrone=false`*
5	Ausführungsvorkommen	Zeigt die Ausführung einer Operation von dem Teilnehmer.	`Node:ExecNode`
6	Rückrufmeldung	Eine Meldung, die zu einem Teilnehmer zurückkehrt, der auf die Rückkehr von einem früheren Aufruf wartet	`Edge:ObjektFlow` oder `Edge:ControllFlow`
8	Kommentar	Stellt weitere Informationen zu Elementen bereit, mit denen er verknüpft ist.	`Node:Comment`

Tabelle 4: Mapping von Sequenzdiagrammen

Die Interaktionen in SD können je nach modelliertem System sehr komplex werden. Um diese komplexe Sachverhalte zu modellieren, wurde in UML 2.0 das Konzept der „*inline expressions*“ unter dem Namen **kombinierte Fragmente** übernommen [UMLWiki]. Das können zum Beispiel Schleifen, Verzweigungen und andere Alternativen seien (siehe Abbildung 46, 13-Schleife). Ein kombiniertes Fragment besteht aus einer oder mehreren Interaktionen, in die jeweils eine oder mehrere Sequenzen eingeschlossen sind. In dieser Arbeit werden folgenden Typen von kombinierten Fragmenten unterstützt:

- ***Opt*** – Optional. Schließt eine Sequenz ein, die eintreten oder nicht eintreten kann.
- ***Alt*** – Alternative Sequenzen. Es tritt nur jeweils eine Sequenz ein (nicht zu verwechseln mit Gabelung).
- ***Loop*** – Die Sequenz wird einmal oder mehrere Male wiederholt. Im Wächter kann die Bedingung angeben, unter der es wiederholt werden soll.
- ***Par*** – Es sind zwei oder mehrere parallele Sequenzen. Dieses kombinierte Fragment kann für die Modellierung vom parallelen Muster verwendet werden.

Im AOM werden kombinierte Fragmente durch spezielle Knoten modelliert, die von der *InvocationAction* erben. Diese Knoten spielen die Rolle von Wächtern und speichern in sich die Bedingungen, die den Kontrollflussleiten. Die entsprechenden Bedingungen werden auf den Kanten als *Precondition*-Attribut gespeichert.

Abbildung 47 zeigt ein Szenario, indem ein *Opt*-Fragment vorkommt. Die Methode *BindCustomer* wird nur dann aufgerufen, wenn die *Opt*-Bedingung stimmt. Links im Bild ist ein Ausschnitt aus dem Sequenzdiagramm zu sehen. Rechts ist seine Abbildung im AOM mit Kanten-Attributen dargestellt. Der graue Bereich rechts zeigt die Grenzen des Opt-Fragments im AOM. Der Knoten 2 repräsentiert dabei den Eintritt in das Fragment und ist ein *InvocationAction*-Knoten. Die Bedingungen werden im AOM direkt in Kanten im Attribut *Preconditions* gespeichert.

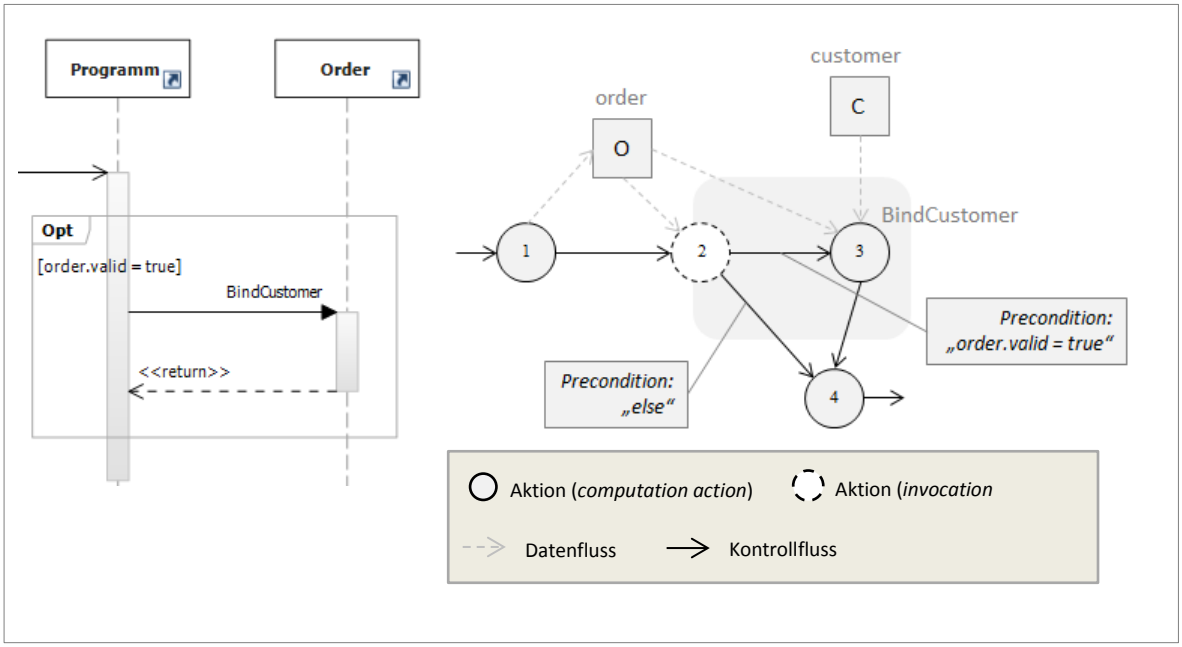


Abbildung 47: Kombinierte Fragmente im AOM

Klassendiagramme

Wie im zweiten Kapitel bereits beschrieben legen Klassendiagramme in der UML die statische Struktur eines Softwaresystems fest. Hier wird eine Klassenhierarchie definiert sowie Eigenschaften und Methoden der Klassen beschrieben. Außerdem kann jede Klasse eine Reihe von Attributen besitzen, wie etwa *isAbstract* oder *isActive*. Attribute sind vom UML-Profil abhängig und können variieren. Da in dieser Arbeit das UML-API der Firma Microsoft verwendet wird, werden nur die entsprechenden Profile verwendet, wie zum Beispiel das *CSharpProfile*. Die Stereotypen in diesem Profil definieren zusätzliche sprachspezifische Attribute, die Transformationen eines Modell-Elements in C#-Programmcode ermöglichen sollen.

Alle Objektknoten aus der UML werden durch Abbildung nach AOM in zwei Gruppen aufgeteilt: aktive und passive Objekte. Ein aktives Objekt ist eine Instanz einer **aktiven Klasse** und wird im UML-Modell durch ein Attribut *isActive* markiert. Die Besonderheit einer aktiven Klasse besteht darin, dass sie einen unabhängigen Steuerungsfluss erzeugen und die Operationen parallel zu den anderen Kontrollflüssen ausführen kann. Die Klassen, die keine Steuerungsaktivität auslösen, werden implizit passiv genannt. Sie werden als Datencontainer verwendet.

4.3.3 Grafische Darstellung des Objektmodells

In dieser Arbeit wurde für die Darstellung des Objektmodells eine eigene grafische Notation entwickelt. Im Folgenden werden grafische Elemente des Metamodells 4.3.1 dargestellt. Die Elemente des Objektmodells AOM haben eine gewisse Ähnlichkeit mit den Elementen des Aktivitätsdiagramms: Die Aktivitätsknoten werden als Kreise und die Objektknoten als Rechtecke dargestellt. Die Kinderklassen in AOM erben auch die Grundform der Vaterklasse und unterscheiden sich in Farbe und Schattierungen.

Knoten

Um die gleiche Semantik zu unterstützen werden alle Aktionen in Form eines Kreises dargestellt:

- *AktivityNode* – ein Kreis ohne Füllung
- *ComputationAction* – ein Kreis mit hellgrauer Füllung
- *ReadWriteAction* – ein Kreis mit dunkelgrauer Füllung
- *InvocationAction* – ein gestrichelter Kreis

Ein weiterer Knotentyp sind Objektknoten und davon abgeleitete Typen. Sie werden alle als Rechtecke dargestellt:

- *ObjektNode* – ein Rechteck ohne Füllung
- *PassivObject* – ein Rechteck mit hellgrauer Füllung
- *ActiveObject* – ein Rechteck mit dunkelgrauer Füllung

Kanten

Die Kanten sind in dem Objektmodell auch typisiert und unterscheiden sich in ihrer grafischen Repräsentation.

- *ControllFlow* – ein einfacher Pfeil

--> *ObjectFlow* – ein hellgrauer gestrichelter Pfeil

◄► *UpdateObjekt* – ein doppelter Pfeil (zeigt somit, dass der Zugriff synchron ist)

Wie man im Kapitel 5 später sehen wird, kann das Objektmodell vor und nach Modifikationen als Graph gespeichert und visualisiert werden. Dadurch können Modifikationen und Suchmuster leicht analysiert und auf Fehlern untersucht werden. Bei der Definition der Suchmuster in kommendem Kapitel wird diese grafische Notation ebenfalls verwendet.

4.4 A2: Mustersuche im Objektmodel

In diesem Kapitel werden typische Muster beschrieben, die in einem Kontrollfluss vorkommen. Diese Muster werden durch Suchregeln bestimmt, die für die Analyse verwendet werden. Ein Kontrollfluss besteht aus mehreren Aktionen und Kontrollflusskanten, über die keine Objekt-Token fließen. Das Kontrollfluss beginnt immer von einem Startknoten [UML2.3]. Somit ist der Startknoten der Ausgangspunkt für die Analyse des Kontrollflusses.

Bei der Modellierung des Kontrollflusses in Aktivitätsdiagrammen spielen sogenannten Kontrollknoten 4.1.1 eine wichtige Rolle. Sie können die Kontrollflussrichtung ändern oder spalten den in mehrere Flüsse. Dazu gehören auch der Startknoten (*InitialNode*), der Endknoten (*ActivityFinalNode*), die Entscheidungsknoten (*DecisionNode*) und die Zusammenführung (*Merge-Node*). Für die Modellierung von Nebenläufigkeit existieren spezielle Knoten: die Splitting (*ForkNode*), die Synchronisation (JoinNode) und das Ablaufende (*FlowFinalNode*), das einen Kontrollfluss darstellt. Die grafische Darstellung dieser Knoten findet man in Abbildung 34. Der Kontrollfluss im kombinierten Modell wird generell durch folgende Elemente beschrieben:

1. **Aktive Objekte** in Klassendiagrammen (2.2.2)
2. **Kontrollflusspfade** und -**knoten** in Aktivitätsdiagrammen (2.2.3)
3. **Nachrichten** und **Signale** in Sequenzdiagrammen (2.2.4)

Die folgenden Unterkapitel beschreiben einzelne Muster näher und definieren ihre charakteristischen Merkmale (Suchregeln), die später in Kapitel 5 für die Formulierung der Transformationsregeln in GrGen.NET verwendet werden.

4.4.1 Einfache Kontrollflussmuster

Im Jahr 1999 wurde eine „*Workflow Patterns Initiative*“ von der Eindhoven University of Technology (Professor Wil van der Aalst) und Queensland University of Technology (Professor Arthur ter Hofstede) gestartet. Es wurden verschiedenen Ansätze und Muster [RtHEvdA06] auf ihren relativen Stärken und Schwächen beurteilt, sowie bestimmte auf Geschäftsanforderungen geprüft. Laut [Woh05b] und [RtHEvdA06] werden in der UML 2.3 folgende einfache Muster direkt unterstützt (siehe Abbildung 34 rechts):

1. Eine Sequenz
2. Parallelisierung (*Fork*)
3. Zusammenführung (*Join*)
4. Verzweigungsknoten (*Decision*)
5. Verbindungsknoten (*Merge*)
6. Multi-Choice
7. Multi-Merge

Diese Muster lassen sich kombinieren und bilden komplexere Muster, die in folgendem Kapitel beschrieben werden. Alle diese Muster können durch entsprechende Suchregeln in AOM gefunden werden.

Sequenz

Obwohl die Sequenz nach [RtHEvdA06] zu den einfachen Mustern gehört, kann sie unterschiedliche Formen und Größen haben: Zwischen zwei Aktivitäten kann es Kontrollflüsse (einfache Pfeil) oder Datenflüsse (als Pfeil zwischen zwei Pins oder mit einem ObjectNode) geben. Die Länge einer Sequenz kann auch unterschiedlich sein.

Die Ausführungsreihenfolge der Aktionen in Sequenzen spielt auch eine wichtige Rolle. In dieser Arbeit wird zwischen kausal und nicht kausal bedingten Ausführungsreihenfolgen unterschieden. Eine **kausal bedingte Ausführungsreihenfolge** liegt vor, wenn die Ausführungsreihenfolge der Bearbeitungsschritte einer Sequenz für dessen Resultat von Bedeutung ist. Das ist zum Beispiel der Fall, wenn zwischen einzelnen Bearbeitungsschritten (Aktivitäten oder Teilsequenzen) eine Datenabhängigkeit existiert. Eine Reihenfolgeverletzung, die durch die Parallelisierung entstehen kann, führt in diesem Fall zu einem Fehlverhalten.

Nicht kausal bedingte Ausführungsreihenfolge: Die Ausführungsreihenfolge der Bearbeitungsschritte einer Sequenz ist nicht für dessen Resultat von Bedeutung. Die Reihenfolge ist willkürlich festgelegt und kann beliebig vertauscht werden. Aufgrund dessen können die Bearbeitungsschritte (Teilsequenzen oder einzelnen Aktivitäten) auch nebenläufig ausgeführt werden.

Die einfachste Form einer Sequenz ist die Kontrollflusssequenz aus zwei Aktivitäten. Die Aktivitäten sind linear miteinander verbunden und es findet kein Datenaustausch statt (Abbildung 48). Hier sind auch Gabelungen und implizite Schleifen möglich, wenn zum Beispiel ein Weg von der Action2 zu Action1 existiert. Da in diesem Fall keine Datenabhängigkeiten vorliegen, geht man davon aus, dass es hier um eine nicht kausal bedingte Ausführungsreihenfolge handelt.

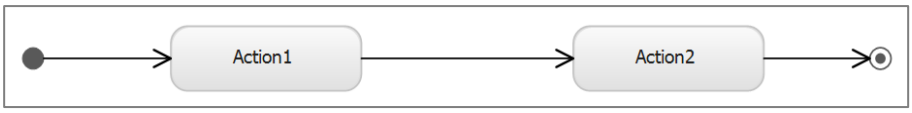


Abbildung 48. UML: Einfache Kontrollfluss-Sequenz

Im Objektmodell AOM werden Start- und Endknoten durch *InvocationAction*-Knoten mit der kleinsten und den größten Nummer dargestellt:

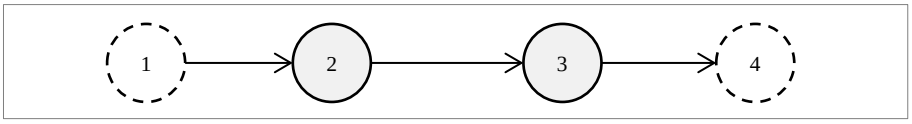


Abbildung 49. Objektmodell: Einfache Kontrollflusssequenz in AOM

Im Vergleich zu den trivialen Fällen, die gerade vorgestellt wurden, tragen die meisten Kontrollflüsse auch Daten zwischen einzelnen Aktivitäten mit. Zum Beispiel generiert die Action1 ein Objekt, das von der Action2 weiterverarbeitet wird. So entsteht zwischen beiden Methoden ein Datenfluss, der in AD durch In- und Output-Pins dargestellt wird. Die Aktivitäten können auch mehrere Input- oder Output-Pins haben. Mehr dazu findet man bei der Beschreibung von ObjectMerge-Muster (4.5.1).

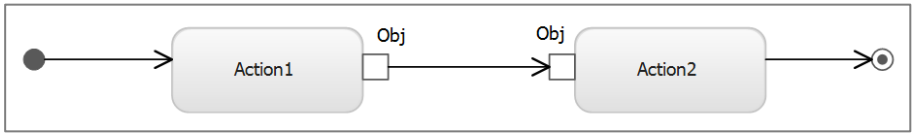


Abbildung 50. UML: Datenfluss mit Pins in AD

Eine weitere Möglichkeit, passive Objekte darzustellen, die in einem Fluss übertragen werden ist der sogenannte Objektknoten. Das Objektfluss-Modellelement wurde in der UML 2 neu eingeführt. Der Objektknoten gibt auch an, ob die Daten direkt im Fluss übertragen werden müssen oder zwischengespeichert werden können. Der Objektknoten wird in einigen Suchmustern als Puffer verwendet. Mehr dazu findet man in den Kapiteln 4.4.3 und 4.5.4

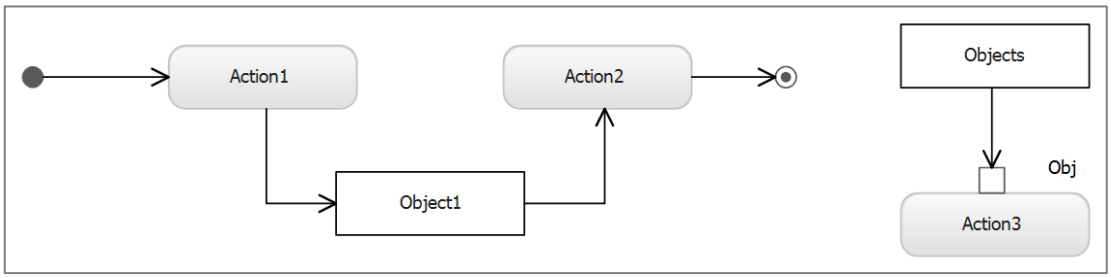


Abbildung 51. UML: Datenfluss mit Objektknoten

4.4.2 Komplexe Strukturmuster

In diesem Abschnitt werden wir die Muster betrachten, die implizite Zyklen und Abrechnungen beinhalten, sowie Muster mit mehreren Objekt-Instanzen. Diese Muster sind interessant, weil sie eine Grundlage für verschiedene Parallelisierungsszenarien bilden. In [Woh05b] werden mehr als 40 komplexere Muster in Aktivitätsdiagrammen beschrieben. In dieser Arbeit betrachten wir folgende Muster, die teilweise darauf basieren und spezielle Merkmale besitzen.

Implizite Schleifen

Die folgenden Muster befassen sich mit Schleifen in einem Kontrollfluss. In [Woh05b] werden zwei Arten von Schleifen im Kontrollfluss definiert: willkürliche Zyklen (engl. *arbitrary cycles*) und strukturierte Schleifen (engl. *structured loop*). Der Unterschied besteht darin, dass willkürliche Zyklen mehrere Ein- und Ausgangsmöglichkeiten haben können. Strukturierte Schleifen haben dagegen nur eine Stelle mit dem Rücksprung, die in einem AD durch eine Verzweigung bestimmt wird.

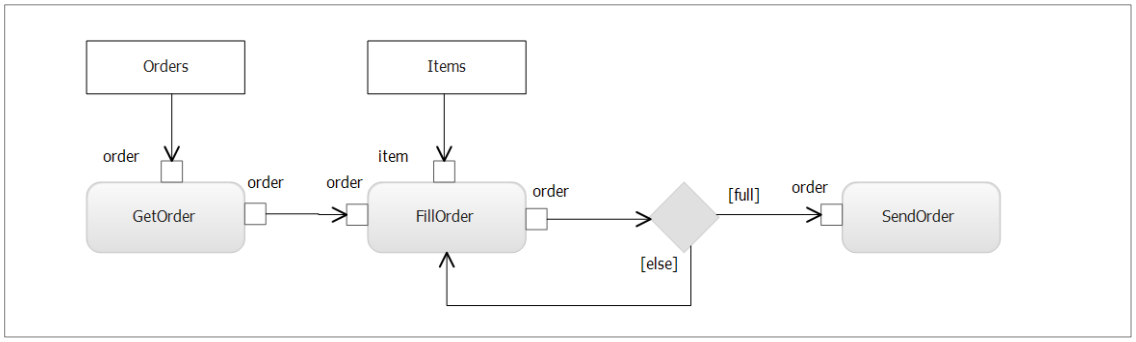


Abbildung 52: UML: Strukturierte Schleife in AD

Verzweigungen und Verbindungsknoten

Verzweigungen spalten den Kontrollfluss. Die genauere Richtung wird erst zur Laufzeit durch die Bedingungen an den Kanten ermittelt. Die Bedingung in Form eines booleschen Ausdrucks wird für jeden einzelnen Steuer- und Daten-Token ausgewertet. In folgendem Beispiel muss die Bestellung durch `CheckOrder`-Methode als "akzeptiert" oder "abgelehnt" markiert werden, wodurch sich der Kontrollfluss in Abhängigkeit dieser Bedingung aufspaltet.

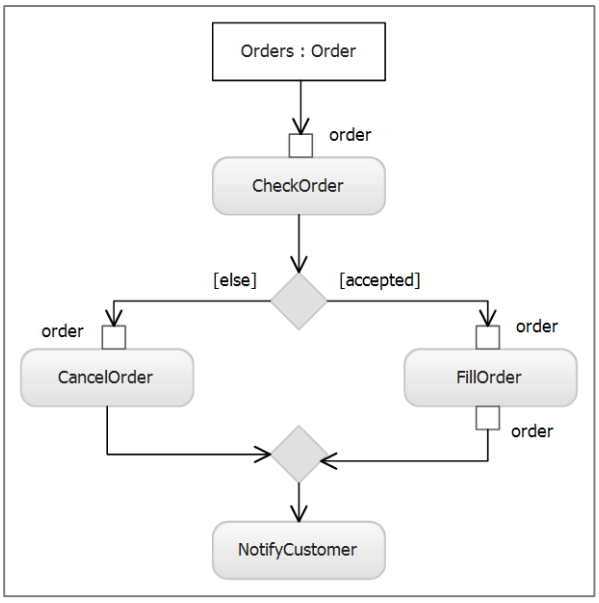


Abbildung 53: Verzweigungen und Vereinigungen in AD

Die Vereinigungsknoten (*Merge*) vereinen mehrere Flüsse wieder zu einem. Alle ankommenden Daten werden durch einen *Merge*-Knoten direkt weitergeleitet. Diesen Knoten haben die gleiche grafische Notation wie Verzweigungen, unterscheiden sich aber auf Codeebene. Im obigen Beispiel werden zwei mögliche Kontrollflüsse durch einen *Merge*-Knoten zusammengebracht, um die Bestellung zu schließen. Folgende Abbildung zeigt dieses Muster in einem Sequenzdiagramm:

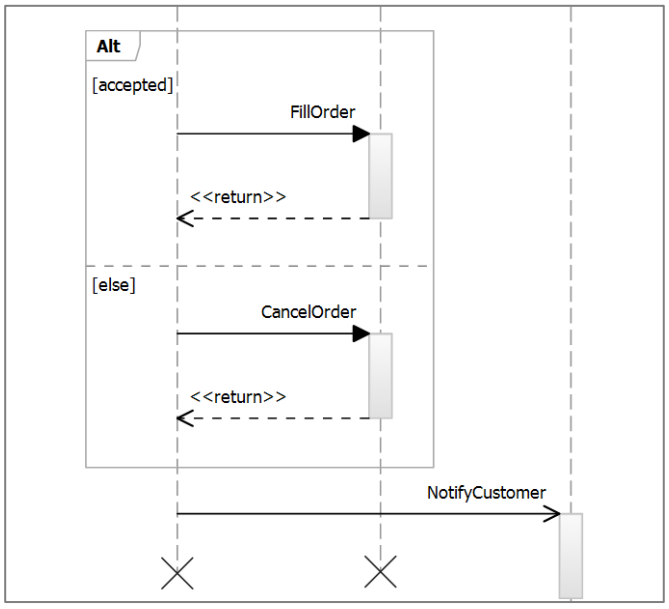


Abbildung 54: Verzweigung in einem SD

In einem Verzweigungsknoten kann man auch komplexere Logik aus einer Aktivitätsgruppe verschachteln. Die Aktivitätsgruppe (engl. *ActivityGroup*) ist ein abstraktes Element, das eine Menge von Knoten und Kanten einer Aktion gruppiert (Abbildung 55). In diesem Fall wird die Aktivitätsgruppe als Teilgraph vor dem Kontrollknoten eingefügt. Der Wächter bekommt dann einen Kontrollfluss und ein oder mehrere Objekt-Flüsse, die für die Entscheidung notwendig sind. Hier wird ein *order*-Objekt in eine Aktivitätsgruppe überprüft (siehe Abbildung). Das Ergebnis (true/false) dieser Überprüfung wird von dem Wächter verwenden und abhängig

davon, wird der Kontrollfluss entsprechend weitergeleitet. Die Aktivität *IsOrderAcceptable* (unten im Bild) beschreibt die Abläufe innerhalb den Verzweigungsknoten.

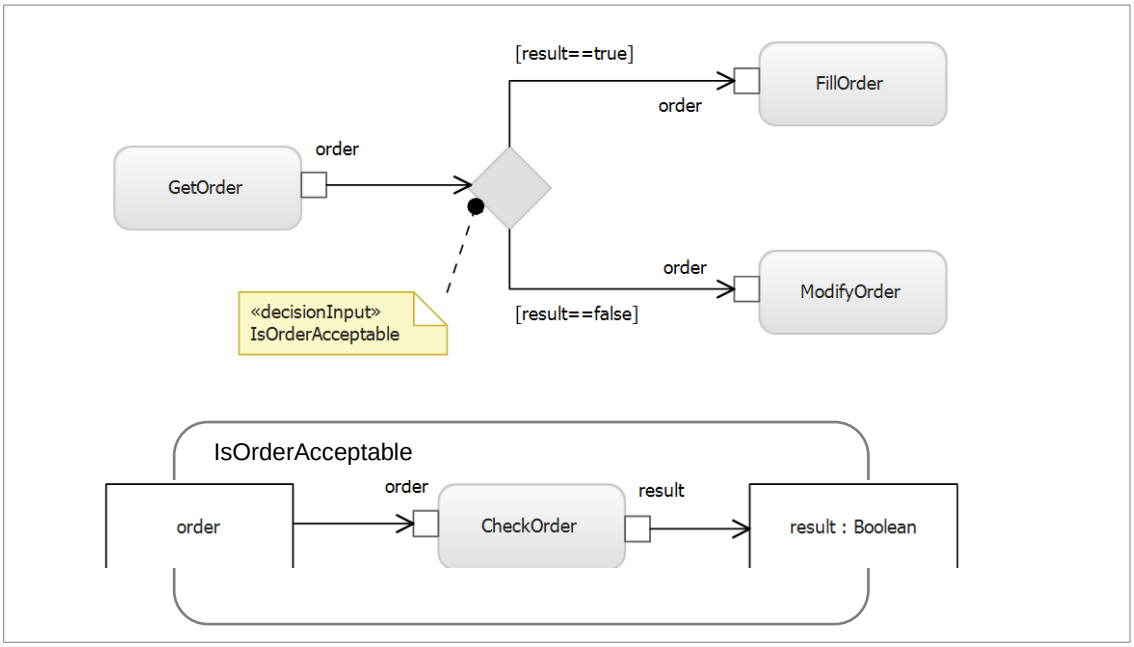


Abbildung 55. UML: Aktivitätsgruppe in einem Wächter

Ist die Überprüfung des order-Objektes erfolgreich wird das Objekt weiter zur Aktion *FillOrder* übergeben, sonst zur Aktion *ModifyOrder*. Die Verbindung zwischen dem Wächter und der Aktivitätsgruppe wird durch einen Kommentarknoten mit dem Stereotyp «*decisionInput*» markiert. In diesem Kommentar steht nur der Name der Aktivitätsgruppe, die selbst auch in einer anderen Datei definiert werden kann.

Folgende Abbildung 57 zeigt wie die oben gezeigte Aktivitätsgruppe als Teilgraph in einem Objektmodell-Graph dargestellt wird. Der Knoten 2 bezeichnet dabei den Eintritt in die Aktivitätsgruppe und der Knoten 4 den Austritt. Die Knoten 1, 3 und 5 stellen die Aktionen *GetOrder*, *CheckOrder* und *Fill-/ModifyOrder* dar. Die Knoten 2 und 4 bezeichnen Ein- und Austritt aus der Aktivitätsgruppe.

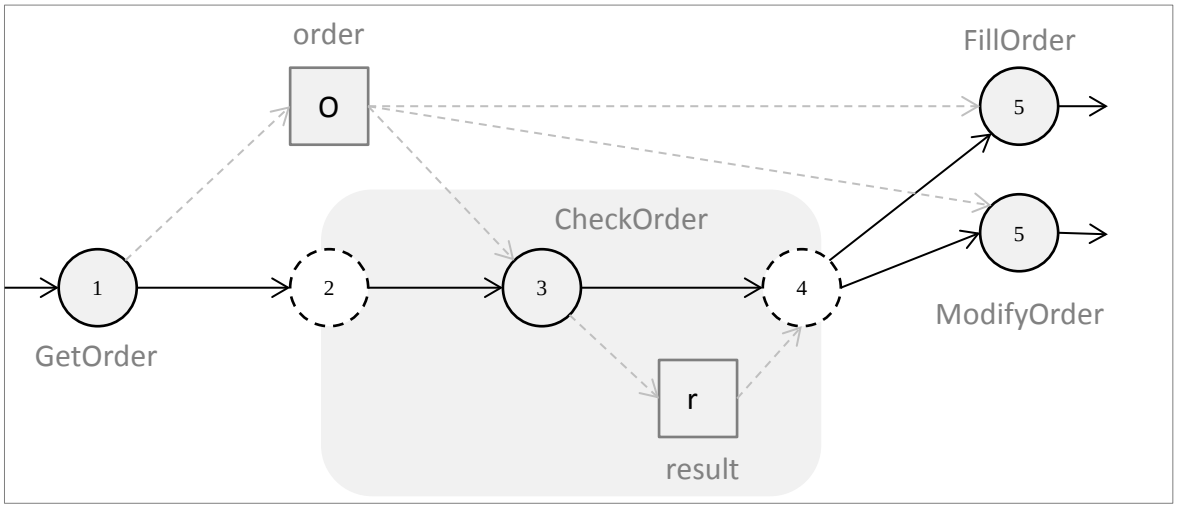


Abbildung 56. Objektmodell: Aktivitätsgruppe als Teilgraph

Durch die Einbindung der Aktivitätsgruppe kann der gesamte Graph analysiert werden. Die Suchmuster können außerhalb, als auch innerhalb einer Aktivitätsgruppe gefunden und

transformiert werden. Einer Rücktransformation in UML steht auch nichts im Wege, da alle Informationen in Knoten und Kanten des Arbeitsgraphen in AOM gespeichert bleiben.

4.4.3 Datenflussanalyse und Erkennung von Abhängigkeiten

Um Datenflüsse analysieren zu können und mögliche Datenabhängigkeiten zu finden, werden in dieser Arbeit verschiedene UML-Diagramme zusammen betrachtet. So beschreiben Aktivitätsdiagramme Daten- und Kontrollflüsse, liefern aber keine Informationen über die Struktur der Daten und Ausführungsreihenfolgen innerhalb einer Aktivität. Dafür werden sie zusammen mit Klassen- und Sequenzdiagrammen analysiert.

Wie in vorherigem Kapitel bereits beschrieben wurde, bilden die passiven Objekte, die in SD als Parameter von Nachrichten oder in AD als In- und Output-Pins, einen Datenfluss im unserem kombinierten Modell AOM. Ein weiteres Merkmal für die Bewertung des Parallelisierungspotenzials und Erkennung von Datenabhängigkeiten sind die Beziehungen (Relationen und Kordialitäten) zwischen einzelnen Klassen in einer Klassenhierarchie. Diese drücken aus, wie viele dieser Objekte in Relation zu den anderen Objekten dieser Assoziation stehen. Liegt eine Komposition oder Aggregation vor mit dem Multiplizität 1..n, so kann es ein Hinweis auf parallele Bearbeitung sein.

Datenflussanalyse

Wie in Abschnitt 4.3 beschrieben, besteht ein Datenfluss aus einem Objekt- und Aktivitätsknoten. Die Verbindungskante zeigt dabei, ob ein passives Objekt als Rückgabewert einer Methode weitergegeben wird oder als Parameter in eine Methode reinfließt. Die aktiven Objekte haben einen besonderen Beziehungstyp mit ihren Methoden, der mit der *UpdateObjekt*-Kante repräsentiert wird. Wenn eine Aktion mehrere Input-Pins hat, dann sollen zuerst die Beziehungen zwischen den einfließenden Objekten und dieser Aktion analysiert werden. Dafür wird im Klassendiagramm nachgeschaut, ob diese Aktivität zu einem von diesen Objekten (Pins) gehört. Das kann zum Beispiel sein, wenn ein aktives Objekt seinen Zustand ändert und andere passive Objekten als Parameter dafür verwendet. Folgendes Beispiel zeigt eine Aktion mit mehreren reingehenden Objekt-Flüssen und das dazugehöriges Klassendiagramm:

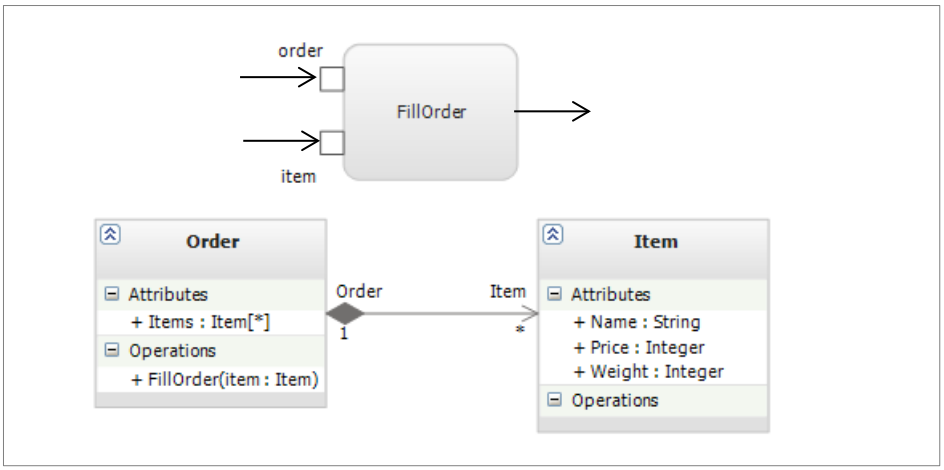


Abbildung 57. UML: Datenfluss in AD und dazugehörige Klassen

Wie man sieht, haben beide Objekte verschiedene Bedeutung und Rolle. Das *order*-Objekt ist ein aktives Objekt, das die Aktion startet. Das item-Objekt ist dagegen passiv und wird als Parameter verwendet.

Im Objektmodell AOM wird dieses Muster wie folgt aussehen:

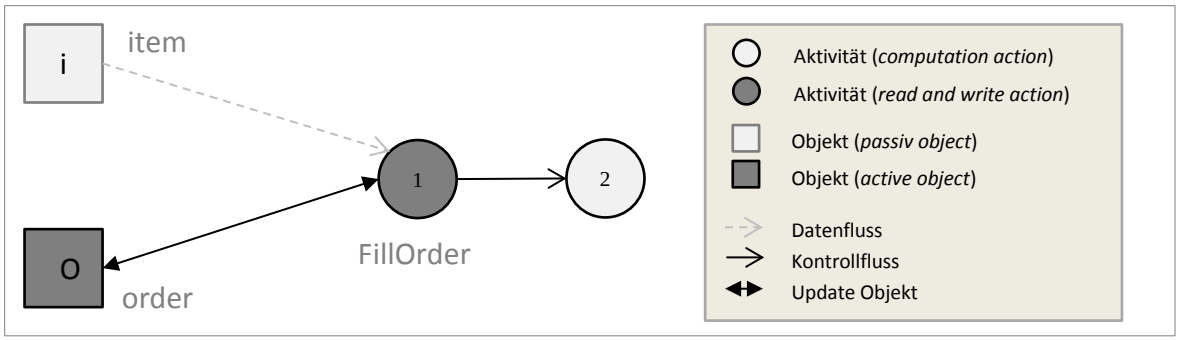


Abbildung 58. AOM: Datenfluss und aktives Objekt

Eine Aktion vom Typ *UpdateObjekt* wird immer als ein Synchronisierungspunkt betrachtet, weil dadurch ein Objekt geändert wird. Die passiven Objekte, die als Parameter übergeben werden, können oft unabhängig voneinander erzeugt werden (implizite Parallelität). Aber zu dem Zeitpunkt als das aktive Objekt durch die *UpdateObject*-Aktion geändert wird, sollen alle Parameter bereits vorhanden sein.

Für die Realisierung eines Synchronisierungspunktes wird im Objektmodell AOM ein zusätzlicher Knoten vom Typ *InvocationAction* hinzugefügt. Er soll garantieren, dass der Kontrollfluss erst weitergeleitet wird, wenn parallele Flüsse zu Ende sind und somit die Objekte, die für *UpdateObject*-Action gebraucht werden, auch vorhanden sind. Der Knoten ist gleich den Zusammenführungsknoten *(Join)* in dem UML-Diagramm. Laut [UML2.3] hat der *Join*-Knoten diese Eigenschaft auch und somit sind keine weiteren Konstrukte nötig.

Datenabhängigkeiten und Pufferknoten

Wie im Grundlagenkapitel erwähnt, können die WAR- und die WAW-Abhängigkeiten durch die Verwendung neuer Ressourcen (Objekten) aufgelöst werden. Dies kann zum Beispiel durch Verwendung einer Pipeline möglich sein, indem die Aktivitäten die einzelnen Stufen repräsentieren. Der Objektknoten kann dabei als Puffer verwendet werden. Folgende Abbildung zeigt die beiden Datenabhängigkeitstypen WAR und WAW als Muster im AOM.

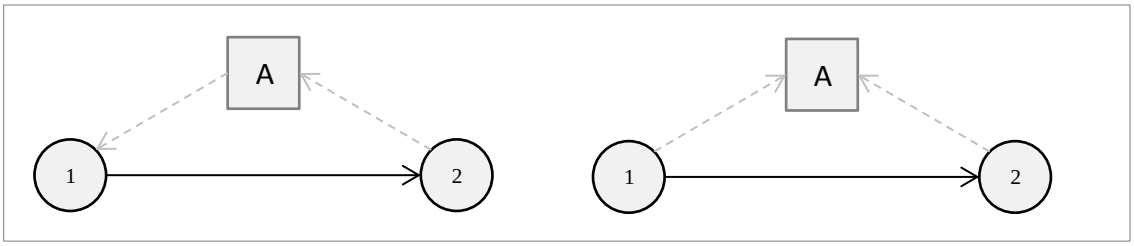


Abbildung 59: WAR- und WAW-Abhängigkeiten in AOM

In Kapitel 4.5.4 wird ein Suchmuster vorgestellt, das mit Hilfe eines Puffers ein Erzeuger-Verbraucher-Muster realisiert. Analog können auch Pipelines modelliert werden.

Abhängigkeiten auf Modellebene

Durch nicht eindeutige Notation in den UML-Diagrammen können später bei der Implementierung unterschiedliche Fehler auftreten. Die Abbildung 60 zeigt ein Aktivitätsdiagramm mit drei Aktionen. Man sieht hier zwei implizit parallele Abläufe, die voneinander unabhängig sind. Erst wenn man genau weiß, wie die Datenflüsse zueinander stehen und zeitlich verlaufen kann, die Situation eindeutig sein. Angenommen das Object *order*, welches von der Aktion *GetOrder* erzeugt wird, kann nicht weitergeleitet werden, weil die *ProcessOrder* nicht bereit ist, das Objekt aufzunehmen. Das wird erst der Fall sein, wenn

CheckCustomer am Ausgabe-Pin das Objekt *customer* bereitstellt, was wiederum erst passiert, wenn *GetOrder* das Objekt *customer* weiterleitet. Somit kommt es zur einer gegenseitigen Abhängigkeit, also einem Deadlock.

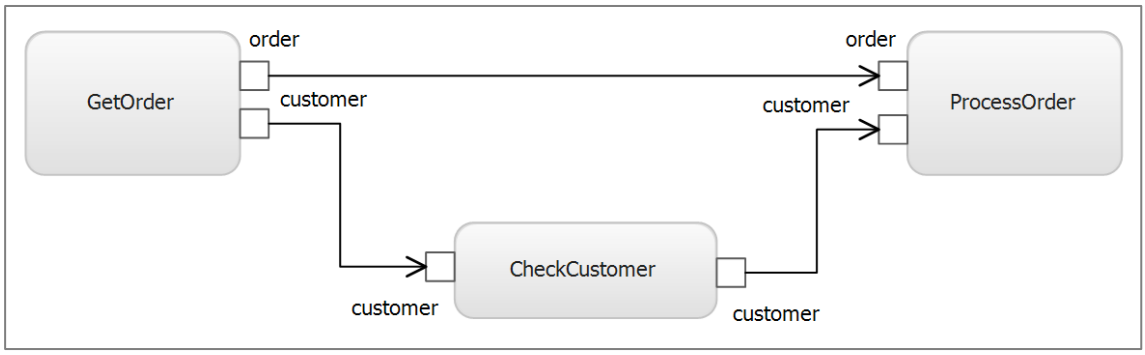


Abbildung 60. UML: Beispiel für Wettlauferkennung

Eine mögliche Lösung dafür ist die Verwendung eines Pufferknoten. Für das oben genannte Beispiel könnte eine mögliche Lösung folgendermaßen aussehen. Die Verwendung des Pufferknoten wird in 4.5.4 anhand des Producer-Consumer-Suchmusters erklärt.

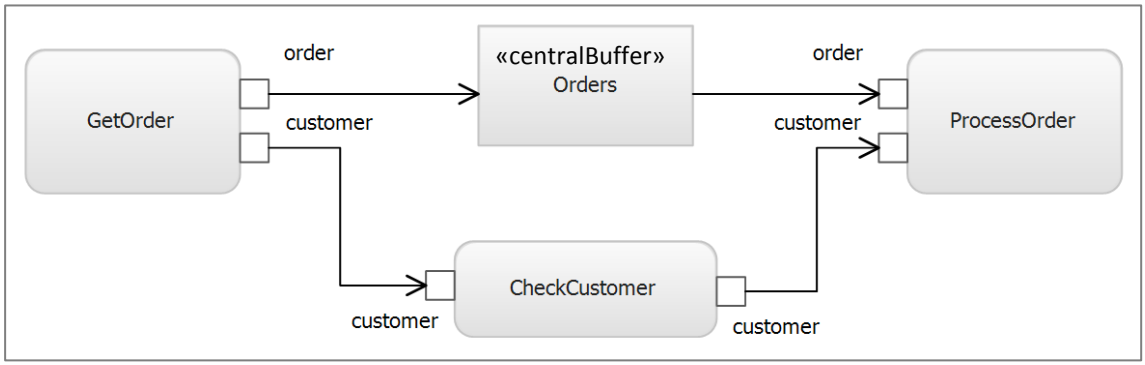


Abbildung 61. UML: Beseitigung des Wettlaufs durch Einfügen eines Puffers

4.5 A3: Suchmusterkatalog und Mustererkennung

In vorherigen Kapiteln wurde der Aufbau des kombinierten abstrakten Objektmodells (AOM) vorgestellt und die Ansätze zur musterbasierte Analyse des Kontroll- und Datenflusses beschrieben. In diesem Kapitel werden nun die Suchmuster definiert, die eine zentrale Rolle in der Modellmodifikation spielen. Jedes Suchmuster wird anhand seiner Eigenschaften in allen drei UML-Modellen formal beschrieben und mit einem kleinen Code-Beispiel illustriert. Danach kommt seine Definition im Objektmodell AOM. Das Suchmuster wird als Teilgraph dargestellt. Abschießend werden die Transformationsregeln beschrieben, die das Suchmuster verändern und Parallelität explizit darstellen.

In dieser Arbeit werden fünf Suchmuster definiert, die sich in Klassen-, Sequenz- und Aktivitätsdiagrammen erkennen lassen und Parallelisierungspotenzial haben. Diese Muster können nur durch eine kombinierte Analyse von allen drei Modellen erkannt werden. Findet man einen Teil des Musters in einem Aktivitätsdiagramm, müssen die anderen Teile auch passen, da das Muster sonst nicht korrekt ist, bzw. parallelisiert werden kann, weil die Voraussetzungen dafür nicht erfüllt sind (es lieg zum Beispiel eine Datenabhängigkeit vor). Erst durch die Kombination, wie sie AOM ermöglicht, werden diese Muster explizit gemacht.

4.5.1 ObjectMerge Suchmuster

In großen Softwareprojekten werden oft Daten aus mehreren Quellen bezogen und für die weitere Verarbeitung oder Darstellung in einem Objekt gekapselt. In modernen Datenbanksystemen wie Oracle oder SQL-Server können komplexe Abfragen formuliert und ausgeführt werden. Doch wenn man mehrere heterogene Datenquellen hat, wie zum Beispiel Webservices, Datenbanken und Dateisystemen, dann werden die Berechnungen und die Zusammenführung der Daten im Code durchgeführt.

In dem einführenden Beispiel 2.1 kommt die Information über eine Bestellung durch einen Webservice und die Kundendaten werden aus der Datenbank geladen. Danach werden diese beiden Datensätze in einem Bestellungsobjekt verbunden und weitergegeben. Theoretisch kann noch die Information über kundenspezifische Rabatte oder früheren Bestellungen hinzukommen. Das Bestellungsobjekt kann neu erzeugt werden oder es wird durch diese neuen Daten nur vervollständigt. Das Prinzip bleibt das Gleiche: Es werden mehrere Objekte produziert und durch eine Aktion konsumiert, die dann ein einziges Objekt rausgibt. Das bezeichnen wir als *ObjectMerge*-Muster.

Folgendes Code-Beispiel illustriert dieses Suchmuster:

```
1: var a = getA();
2: var b = getB();
3: var c = getC();

4: x.update(a, b, c);
5: return x;
```

Wie man sieht gibt es hier folgende Datenabhängigkeiten zwischen Zeilen: 1→4, 2→4 und 3→4. Die Zeilen 1 bis 3 sind aber voneinander unabhängig und können parallel ausgeführt werden, wenn die Methoden *getA*, *getB* und *getC* keine Datenabhängigkeiten miteinander haben. Es können auch mehr als nur drei Objekte, wie in diesem Beispiel sein.

Folgende UML-Modelle enthalten dieses Suchmuster und beschreiben den obigen Code.

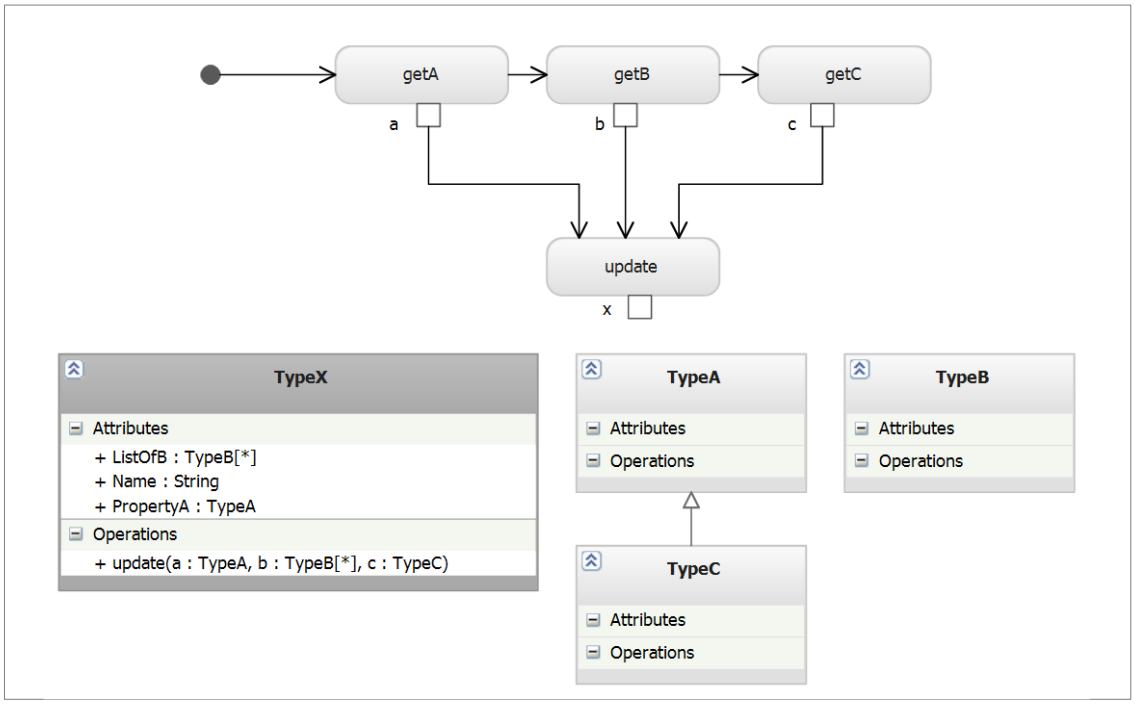


Abbildung 62. UML: ObjectMerge-Muster

Die Methode *update* gehört zu dem Objekt ***x*** und kann seinen Zustand verändern. Dabei muss *x* ein aktives Objekt sein, denn unsere Voraussetzung ist, dass Objekteigenschaften nur durch Objekt selbst geändert werden können. Ein aktives Objekt steht für einen unabhängigen Steuerungsfluss und führt die Operationen durch. Außerdem gehört die Methode *update* zu dem *TypeX*.

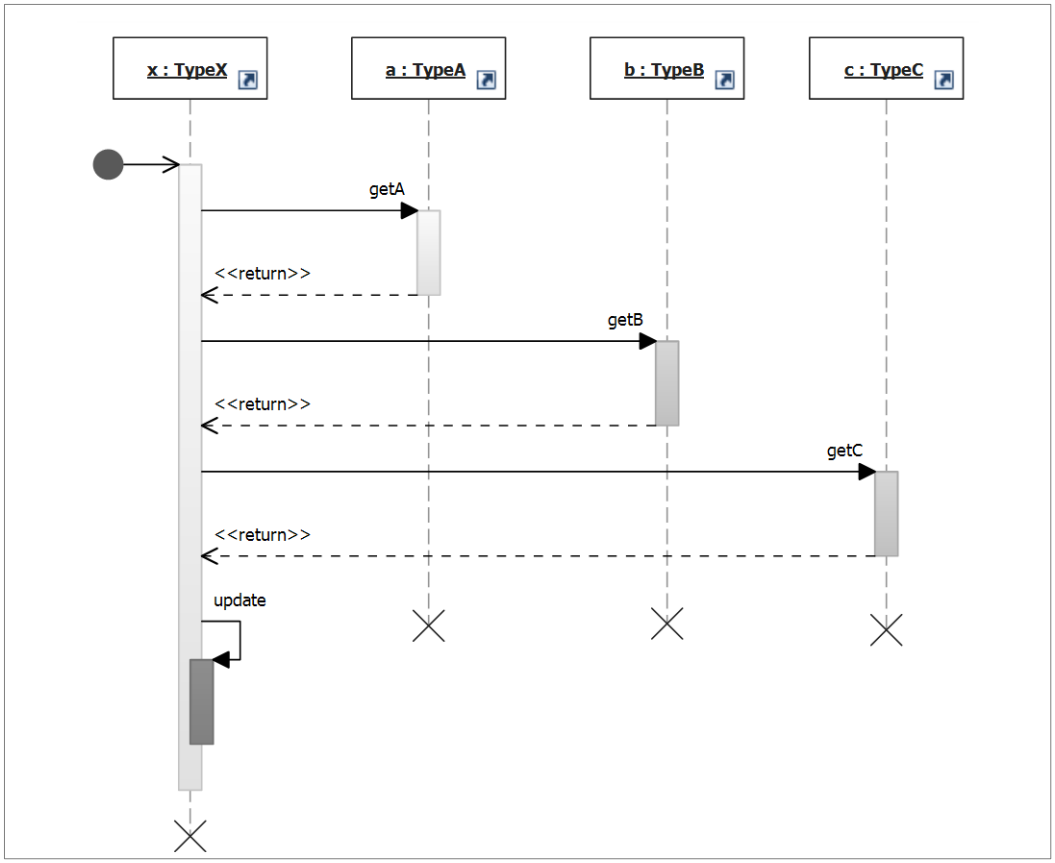


Abbildung 63. UML: ObjectMerge Muster in SD

Formale Beschreibung des ObjectMerge-Musters

Ein *ObjectMerge*-Muster besteht aus mehreren Sequenzen, die zu einer Aktion führen, und kann in einem kombinierten Modell wie folgt formal beschrieben werden:

1. Im AD existiert eine Aktion, die mehrere eingehende Datenflüsse hat und ein Objekt erzeugt.

2. In CD gibt es ein aktives Objekt, das seinen Zustand ändert, und mehrere passive Objekte, die erzeugt und durch das aktive Objekt konsumiert werden.

3. In SD greifen die Aktivitäten auf unterschiedliche Lebenslinien, so dass es keine direkten und indirekten Datenabhängigkeiten gibt.

Folgendes Beispiel zeigt einen Objektmodell-Graph des *ObjectMerge*-Muster mit drei passiven Objekten. Eine Voraussetzung zur Parallelisierung ist, dass Anzahl den passiven Objekten ohne Abhängigkeiten größer gleich zwei ist.

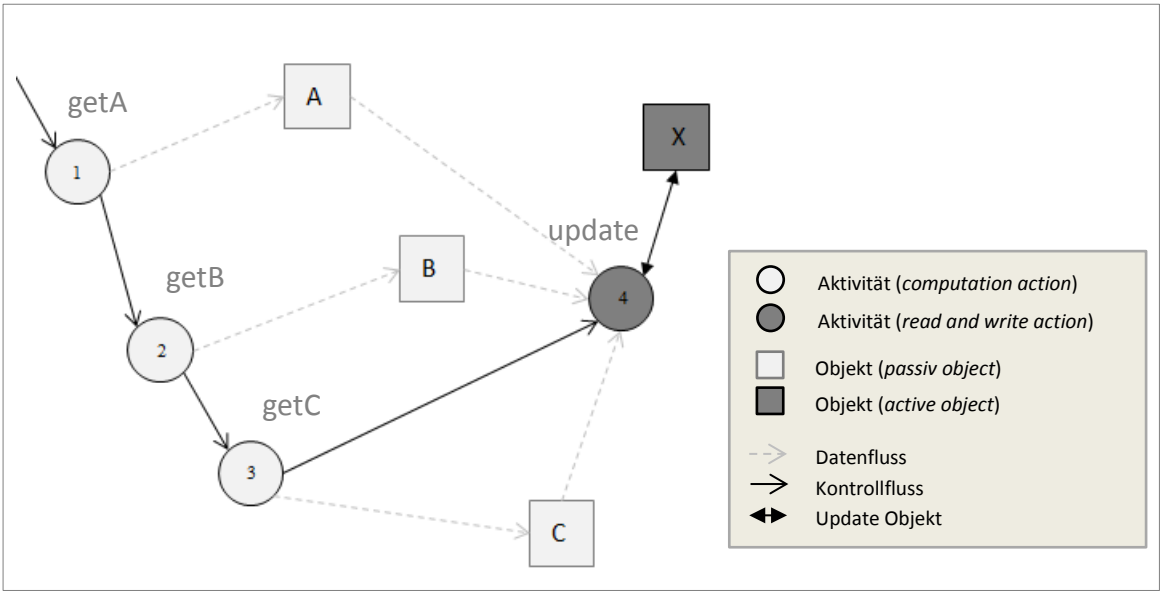


Abbildung 64. AOM: *ObjektMerge*-Suchmuster

Die Aktivitätsknoten sind hier entsprechend der Ausführungsreihenfolge durchnummeriert. Im Objekt-Graph werden alle Knoten auch weitere Attribute besitzen, die hier aus Platzgründen nicht zu sehen sind. Die Aktivitätsknoten sind, wie in Kapitel 4.3 bereits beschrieben wurde, in drei Gruppen aufgeteilt. Hier haben wir zunächst mehrere *computation actions* und eine *read-write action*, die passive Objekte konsumiert und das aktive Objekt verändert.

Dieses Muster wird durch folgendes Teilgraph ersetzt: Zunächst wird eine neue Aktion eingefügt, die als erstes markiert wird und den Kontrollfluss übernimmt. Diese Aktion ist von dem Typ *invocation action* und splittet den Kontrollfluss in mehrere parallele Pfade. Hier entspricht sie den Verzweigungsknoten in einem Aktivitätsdiagramm. Ein weiterer Knoten sammelt abschließend die Pfade wieder zusammen. Der Objektfluss ändert sich dabei nicht.

Folgende Abbildung zeigt den neuen Teilgraph:

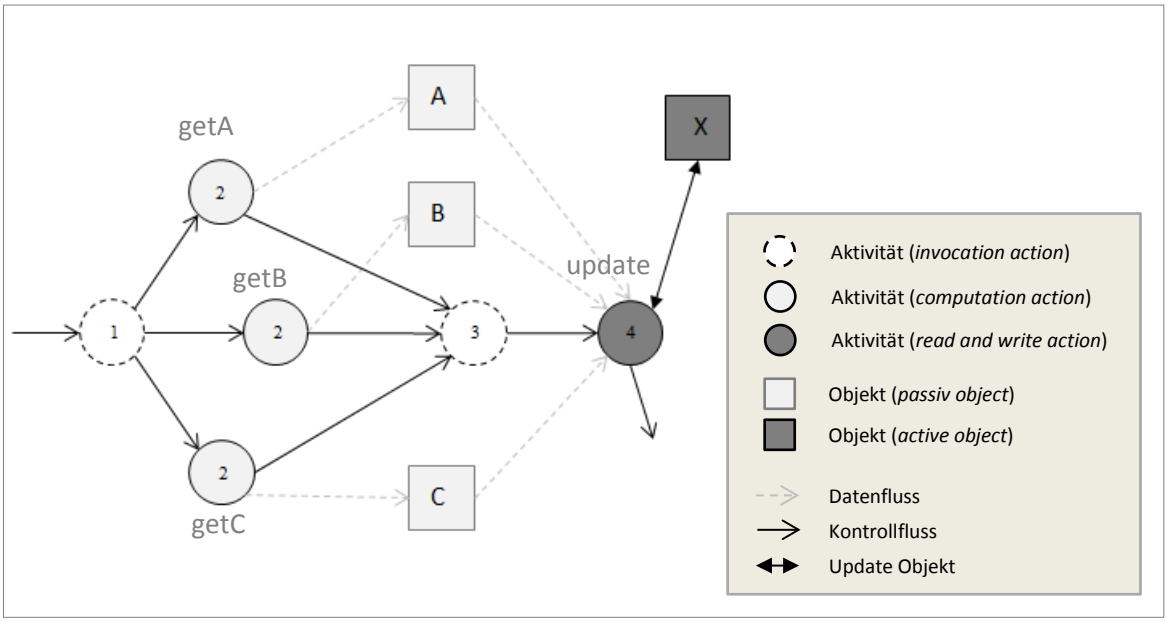


Abbildung 65. AOM: Ersatzmuster für *ObjectMerge*-Suchmuster

Die Parallelisierung eines *ObjectMerge*-Suchmusters ist nur dann möglich, wenn die konsumierten Objekte unabhängig voneinander erzeugt werden. Es darf keine Abhängigkeiten zwischen entsprechenden Methoden geben. In diesem Fall können die Methoden *getA()*, *getB()* und *getC()* als nebenläufige Threads oder Tasks gestartet werden. Danach wird der Hauptfaden warten, bis alle Ergebnisse vorliegen (eine Synchronisierungspunkt) und zu der Methode *x.update()* übergehen.

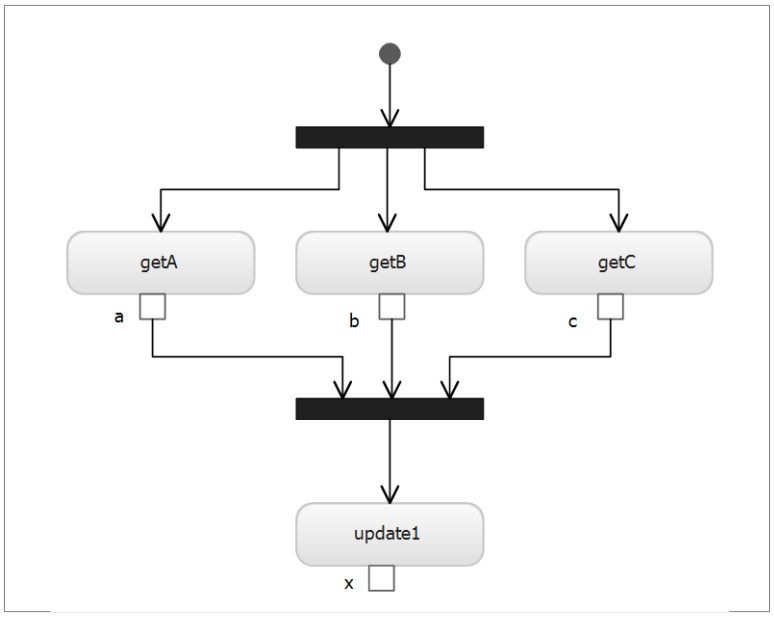


Abbildung 66. UML: Parallelisierung von ObjectMerge-Muster

Im obigen Beispiel hat die Zusammenführung (*Join*) mehrere eingehende Kanten und eine ausgehende Kante. Diese Zusammenführung ist eine Barriere und leitet den Kontrollfluss erst dann weiter, wenn alle eingehenden Token angekommen sind.

Im Gegensatz zum Verbindungsknoten ♦ (*Merge*), werden hier die Abläufe basierend auf drei Regeln synchronisiert [UML2.3]:

- An allen eingehenden Kanten müssen Objekt-Token zur Verfügung stehen, erst dann kann die ausgehende Kante den Token weiterleiten.

- Wenn an den eingehenden Kanten Objekt- und Kontrolltoken eintreffen, werden nur die Objekttoken weitergeleitet. Die Weiterleitung geschieht in derselben Reihenfolge, in der sie an der Synchronisation eintreffen.

- Wenn alle eingehenden Token Kontrolltoken sind, wird an der ausgehenden Kante genau ein Kontrolltoken zur Verfügung gestellt.

Wie man hier sieht, kann das Suchmuster *ObjectMerge* als ein *Master-Worker*-Entwurfsmuster 2.3.4 modelliert und im Anschluss daran implementiert werden. Der Master warten auf die Ergebnisse von Worker-Methoden *getA, getB, getC* und erst dann führt die Methode *update* durch.

4.5.2 ImplicitTermination

Im einführenden Beispiel (Abbildung 1, Kapitel 2.1) ruft die Aktion *GenerateInvoice* die Aktionen *SendMessage* und *GenrateDelveryNote* und übergibt an *SendMessage* ein Objekt. Aus dem AD ist unklar, ob beide Aktionen (Methoden) aus der *GenerateInvoice*-Methode aufgerufen werden oder kommen sie nacheinander. Ein Blick in das Sequenzdiagramm (Abbildung 68 unten) zeigt, dass *SendMessage* ein Endknoten ist und hat keine ausgehenden Pfeile. Sie kennzeichnet also Ende einer Sequenz.

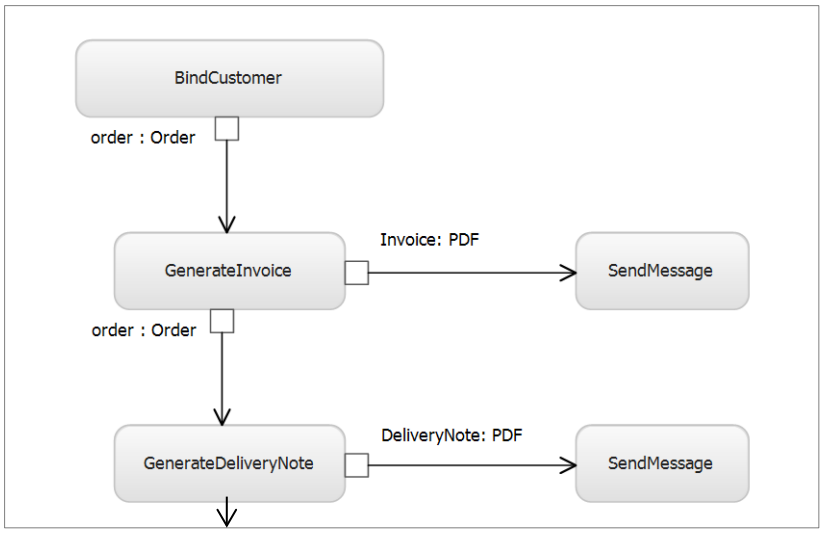


Abbildung 67. UML: ImpicitTermination-Muster

In der UML 2.x gibt es ein spezielles Symbol für den Ablaufende - ⊗ ***FlowFinaNode***. Hier hat der Architekt keinen expliziten *FlowFinal* angezeigt. Aber eine kombinierte Analyse kann es trotzdem finden, weil es in SD deutlich ist.

Nach [UML2.3] beendet das Ablaufende (*FlowFinalNode*) einen einzelnen Ablauf. Bei dem Ablaufenden wird nur das eintreffende Token zerstört. Ein Token ist eine Marke, die den Kontroll- oder Objektfluss markiert 4.1.1. Alle anderen Token in der Aktivität werden dadurch nicht beeinflusst. Somit kann eine Sequenz, die zur Ablaufende führt und keinen weiteren Daten außer eigenem Objektfluss bezieht, asynchron gestartet werden.

Folgendes Code-Beispiel illustriert dieses Suchmuster, das **ImplicitTermination-Muster** genannt wird:

```
1: var Invoice = GenerateInvoice();
2: sendMessage(Invoice);

3: var DeliveryNote = GenerateDeliveryNote();
4: sendMessage(DeliveryNote);
```

Das Sequenzdiagramm (Abbildung 68) zeigt die beiden Teilsequenzen und deren Ausführungsreihenfolge:

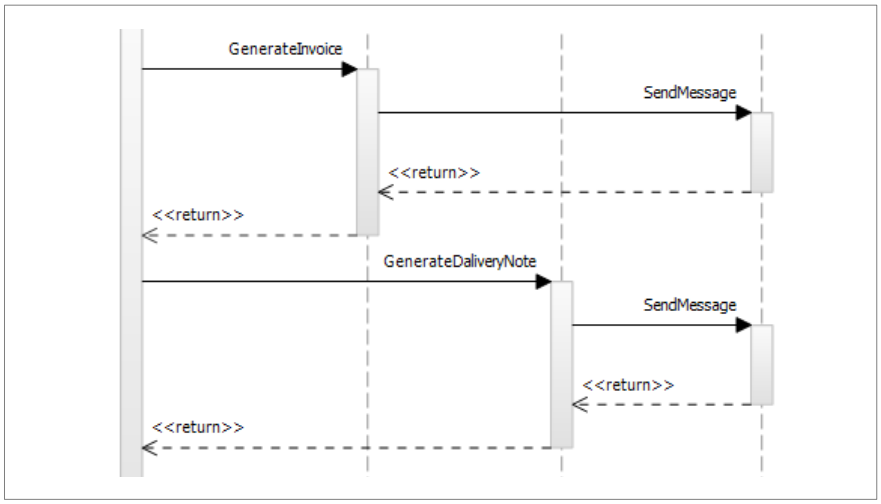


Abbildung 68. UML: ImpicitTermination-Muster mit zwei terminierenden Sequenzen

Formale Beschreibung des ImplicitTermination-Musters

Ein *ImplicitTermination*-Suchmuster kann wie folgt formal beschrieben werden:

1. In AD ist es zwei Aktivitäten bzw. eine Sequenz, wobei die letzte Aktion keine ausgehenden Pfade hat.

2. Die letzte Aktion liefert keine Ergebnisse zurück (Rückgabetyp `void` im Klassendiagramm).

3. Die letzte Aktion soll keinen Objekte zugreifen, mit denen die vorletzte Aktion interagiert (keine *Read and Write Actions*).

Falls dieses Muster gefunden wird, stellt sich die Frage, ob man die letzte Aktion bzw. Sequenz asynchron starten kann und wenn ja, wie?

Im obigen Beispiel werden *GenerateInvoice* und *SendMessage*, sowie *GenerateDeliveryNote* und *SendMessage* jeweils zusammengebündelt. Diese Bündelung ist ein Zwischenschritt bei der Parallelisierung, der in dieser Arbeit oft vorkommt. Methoden, zwischen denen ein direkter Kontroll- und Datenfluss besteht, können nicht parallel ausgeführt werden und mussen daher in eine Aktion zusammengefasst werden. Dieses Suchmuster nennen wir *ActionsBunch*, auf welches im nächsten Kapitel eingegangen wird. Im AOM-Graph sieht das *ImplicitTermination* - Muster wie folgt aus:

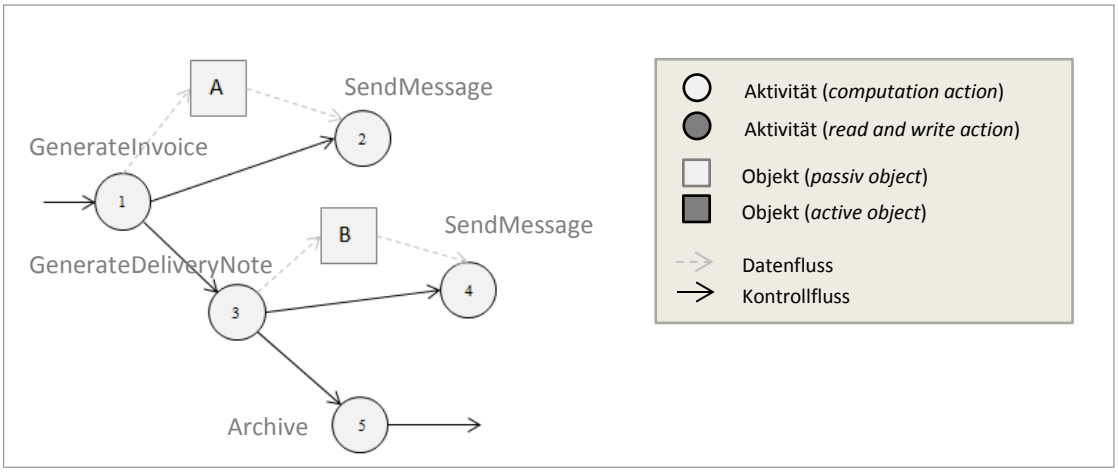


Abbildung 69. AOM: *ImplicitTermination*-Suchmuster

Hier sind zwei Aktionen zusehen (2 und 4), die keine ausgehenden Kanten haben. Das folgt aus dem Aktivitätsdiagramm (siehe Abbildung 67). Folgende Abbildung zeigt eine asynchrone Variante mit zwei *ImplicitTermination*-Mustern. In Kapitel 5.3 findet man eine genauere Beschreibung des Parallelisierungsprozesses.

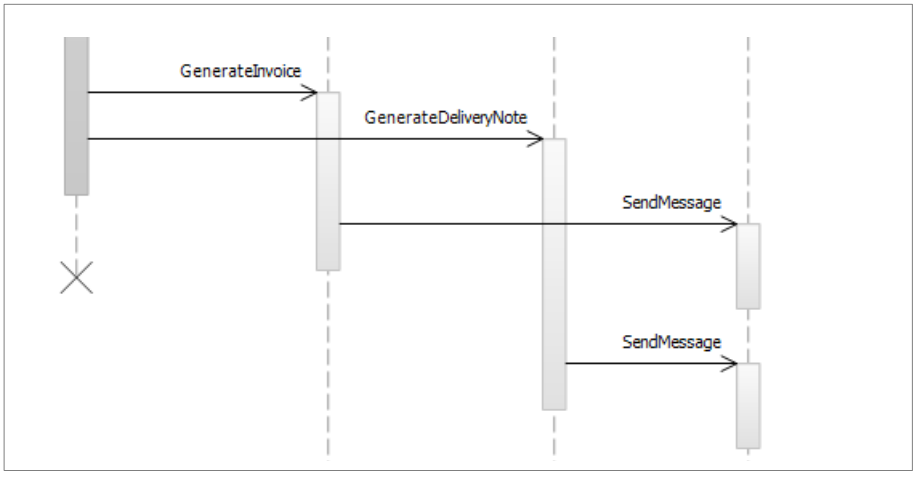


Abbildung 70. UML: Asynchrone Variante von zwei *ImplicitTerminations*

4.5.3 ActionBunch Suchmuster

Das *ActionBunch*-Suchmuster zeichnet sich aus durch mehrere Aktionen, zwischen denen es einen direkten Datenfluss gibt und keine weiteren fremden Daten hineinfließen. Oft wird eine Datenmenge für die Erzeugung von unterschiedlichen Untermengen oder Objekten verwendet. So kann man aus der Menge aller Bestellungen in 2012 die Monatsstatistiken generieren. Die gegebene Datenmenge wird nach Monatsnummer aufgeteilt und in Teilmengen verarbeitet. Die Verarbeitung kann in einem oder mehreren Schritten stattfinden. Die ursprüngliche Untermenge wird durch diese einzelnen Schritte transformiert und am Ende in neuer Form ausgegeben.

Eine weitere Möglichkeit ist, dass die Datenmenge nicht aufgeteilt wird, aber durch verschiedene Filter oder Transformationen durchläuft (Pipeline-Muster). So können Daten aus einer Bestellung für die Erzeugung und Versenden von einer Rechnung, einem Lieferschein und weiteren Dokumenten verwendet werden. Ein Dokument wird generiert, dann als PDF Dokument gespeichert und abschließend per Email versendet. Diese drei Schritte wiederholen sich auch für die anderen Dokumente, die aus den gegebenen Daten erzeugt werden.

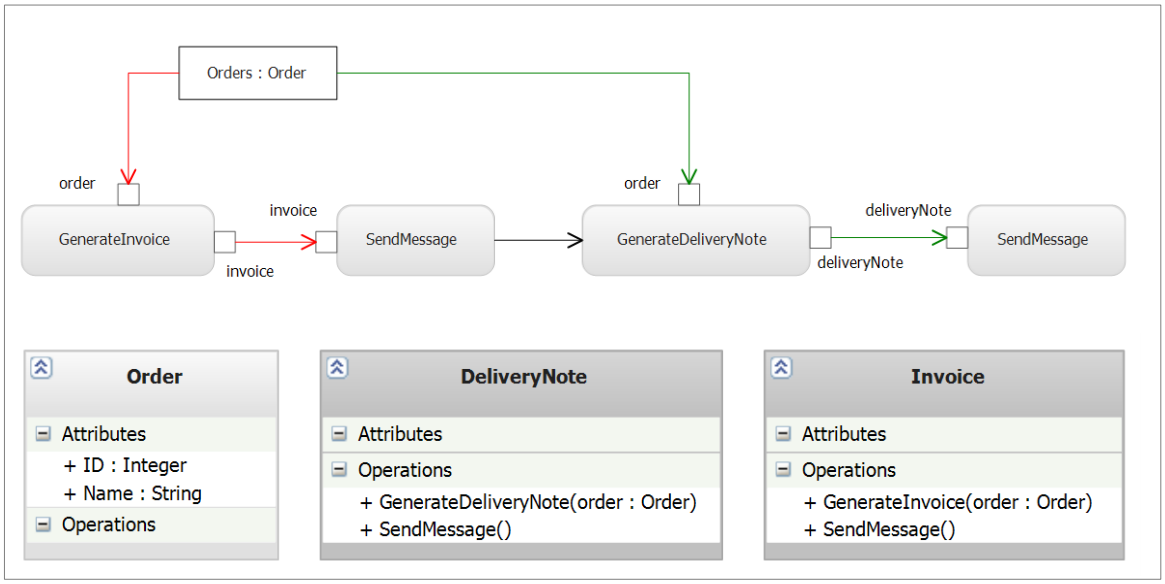


Abbildung 71. UML: *ActionBunch*-Suchmuster

Abbildung 71 zeigt ein Muster, in dem ein Objekt aus der Menge als Parameter in zwei Sequenzen einfließt. Dabei sind zwei unabhängige Datenflüsse zu sehen (hier rot und grün markiert). Innerhalb der Sequenzen gibt es einen direkten Datenfluss – das Objekt wird transformiert und weitergegeben. Zwischen den Sequenzen dagegen gibt es keinen Datenfluss, sondern nur einen Kontrollflussübergang. Pro Sequenz soll es mindestens ein eigenes aktives Objekt geben, wie in der obigen Abbildung: *DeliveryNote* Objekt und *Invoice* Objekt. Dieses Muster ist eine Grundlage für den Aufbau eines Pipelines oder mehreren parallelen Tasks.

Folgendes Code-Beispiel illustriert dieses Muster:

```
1:  var a = getA();

2:  var b = m1(a);
3:  var c = m2(b);

4:  var x = m3(a);
5:  m4(x);
```

Formale Beschreibung des ActionBunch-Suchmusters

Dieses Suchmuster hat folgende Eigenschaften, die ihm formal beschreiben:

1. In AD sind es zwei oder mehrere Sequenzen, die in sich einen direkten Datenfluss haben. Zwischen den Sequenzen gibt es keinen Datenfluss.

2. In SD sollen die Sequenzen nacheinander ausgeführt werden und greifen auf keine weiteren Objekte außerhalb eines eingehenden Parameters zu.

3. In Klassendiagrammen soll es für jede Sequenz ein eigenes aktives Objekt geben.

In dem Objektmodell-Graph sieht das ActionBunch -Muster wie folgt aus:

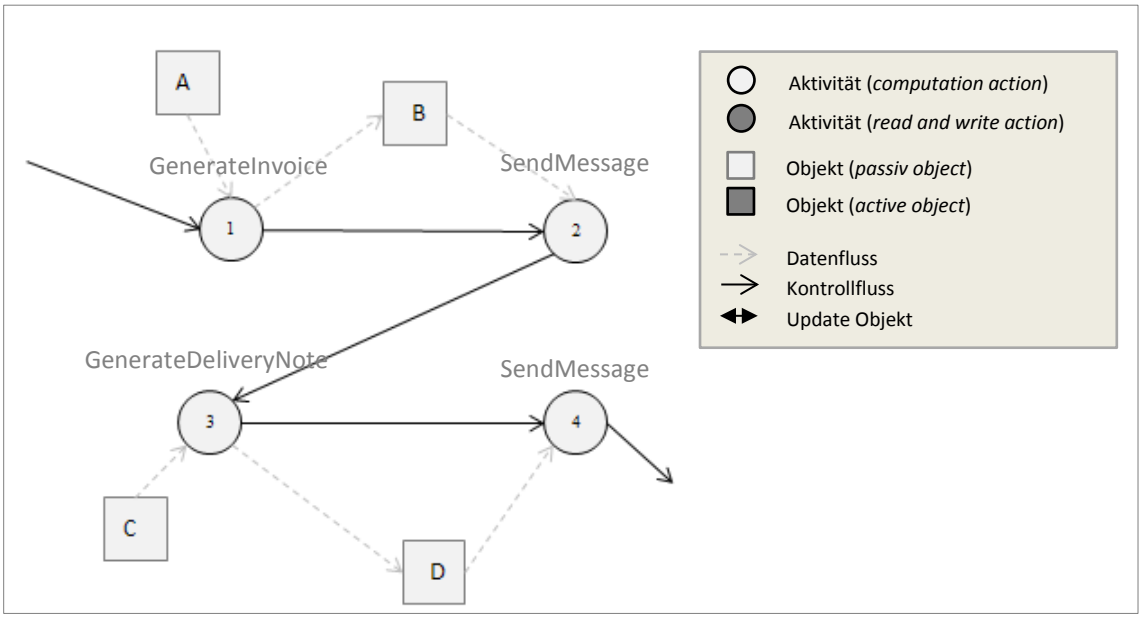
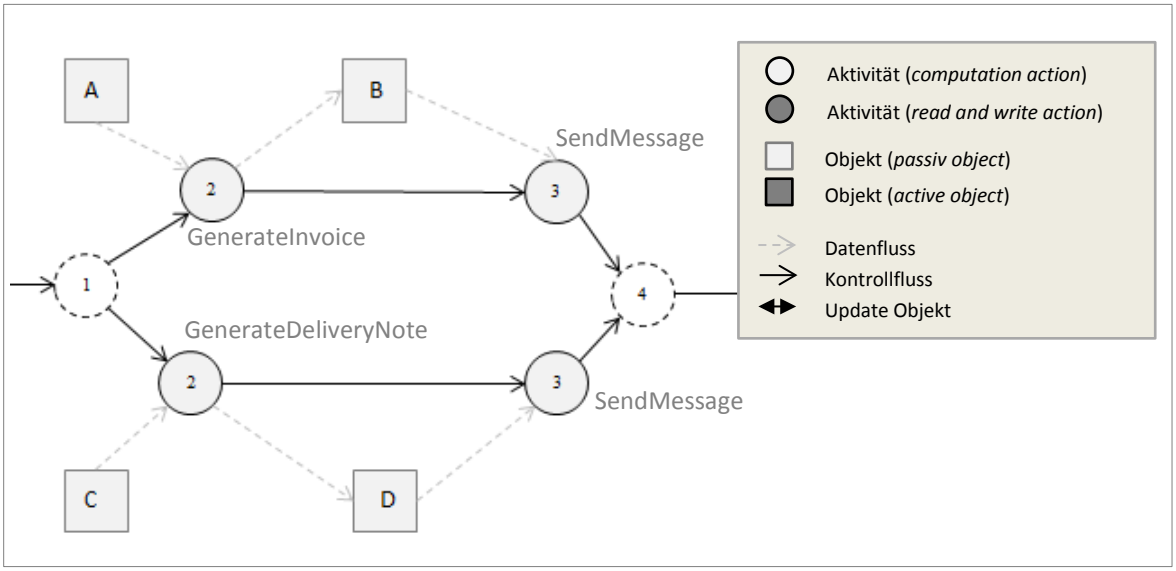


Abbildung 72. AOM: ActionBunch-Suchmuster

Sind diese Bedingungen erfüllt, so werden, wie der Name des Suchmusters schon sagt, die einzelnen Sequenzen zusammen gebündelt und parallel ausgeführt. Folgende Abbildungen zeigen die parallelen Modelle.



In dem Aktivitätsdiagramm werden die Knoten 1 und 4 zu den *Fork-Join*-Knoten gemacht:

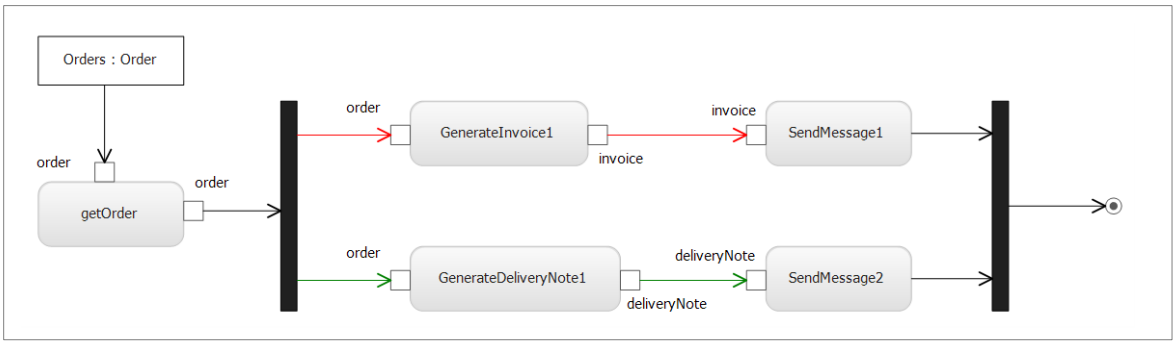


Abbildung 73. UML: Parallele Variante des Musters

In dem Sequenzdiagramm sollen parallele Sequenzen auf unterschiedlichen Lebenslinien ablaufen. Eine mögliche Variante hier ist die Verwendung von Par-Region:

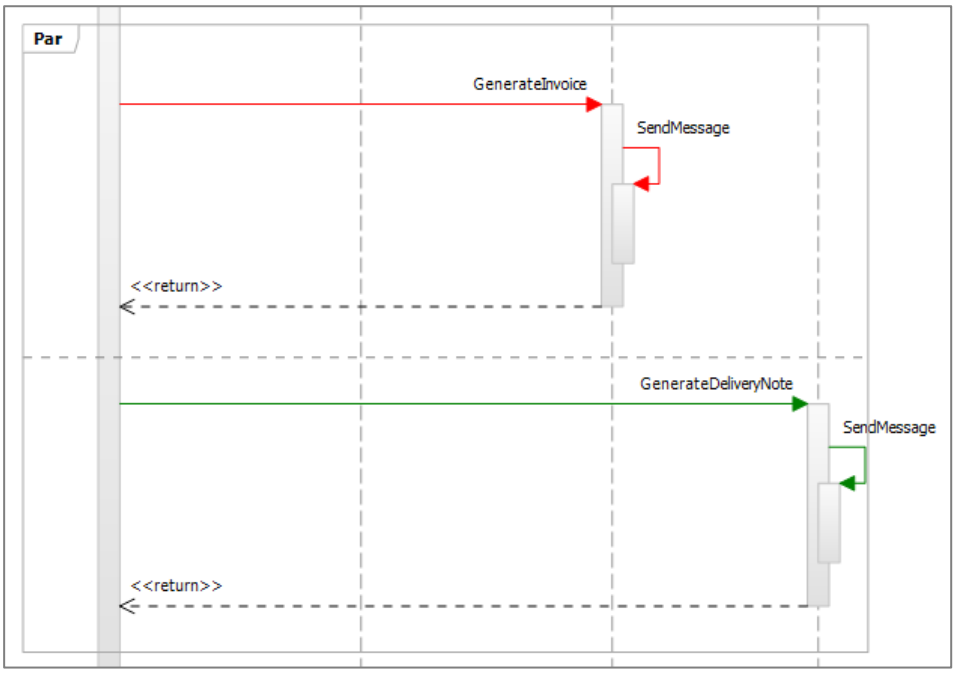


Abbildung 74. UML: Parallele Variante von ActionBunch-Suchmusters

Generell muss es nicht das gleiche passive Objekt sein, das von den Sequenzen bearbeitet wird. Es können unterschiedliche Objekte aus einer Menge sein oder sogar Objekte, die unterschiedliche Typen haben.

4.5.4 Producer-Consumer-Muster (mit Pufferknoten)

Wie der Name schon sagt, beschreibt dieses Muster das Erzeuger-Verbraucher-Problem (engl. *producer-consumer)*. In der UML gibt es vier Arten von Objektknoten [UML2.3]. Einer davon ist der sogenannte Pufferknoten (engl. *CentralBufferNode*), in dem Objekte unabhängig von Aktionen zwischengespeichert werden können. Der Pufferknoten wurde neu in der UML 2.0 eingeführt und ist nahe verwandt mit dem Pin, einem anderen Objektknoten. Der Pufferknoten kann eine bestimmte Anzahl von Objekten zwischenspeichern, das heißt, er hat die Möglichkeit, Objekte im Fluss durch eine Aktion zu puffern. Um einen Pufferknoten von einem anderen Objektknoten in einer Aktion zu unterscheiden, besitzt er das Stereotyp «*centralBuffer*». Die Abbildung 75 zeigt einen modifizierten Ausschnitt aus dem Beispiel 1 indem der Datenfluss mit einem solchen Pufferknoten dargestellt ist. Die Input- und Output-Pins besitzen außerdem eine Eigenschaft *Multiplicity*, die die Werte 0..1, 1, * und 1..* annehmen kann.

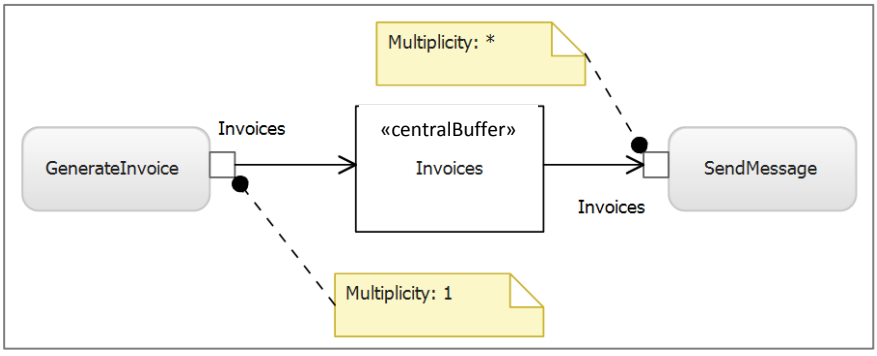


Abbildung 75. UML: Datenfluss mit einem *CentrallBuffer*

Im obigen Beispiel hat eine Methode der Output-Pin den Wert 1 und auf der anderen Seite hat der Input-Pin den Wert *. Das kann so interpretiert werden, dass in den Pufferknoten immer ein Objekt hinzugefügt werden kann und mehrere Objekte gleichzeitig durch *SendMessage* Aktion entnommen werden können.

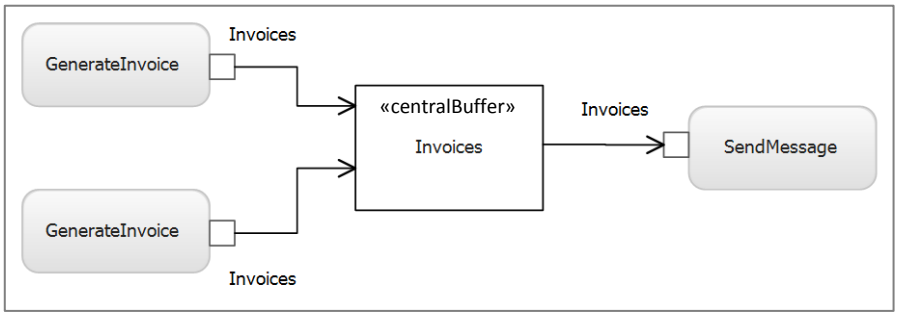


Abbildung 76. UML: *CentrallBuffer* mit mehreren Objektflüssen

In der Abbildung ist ein Pufferknoten dargestellt, der über zwei Objektflüsse Objekte von zwei Aktionen erhält. Dieses Muster modelliert die Tatsache, dass zwei Aktionen nebenläufig Invoices generieren und diese in einem Zwischenlager ablegen, aus dem sich die weiteren Aktionen für den nachfolgenden Arbeitsschritt bedienen. In dem Objektmodell gibt es für den *CentrallBuffer* keinen speziellen Knotentyp, stattdessen wird er durch zwei Objekte mit einem Datenfluss dazwischen repräsentiert. Dabei soll mindestens einen Objektknoten *Multiplicity*-Eigenschaft größer als eins sein (sonst ist das ein direkter Datenfluss zwischen zwei Aktivitäten).

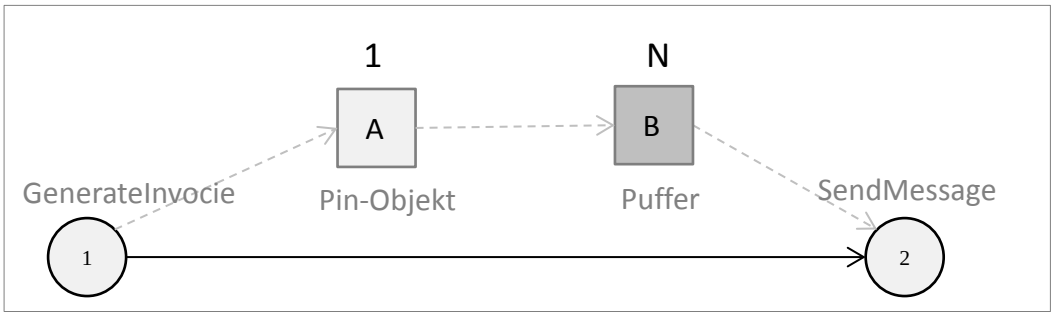


Abbildung 77. AOM: *CentrallBuffer* in dem Objektmodell

Formale Beschreibung des Producer-Consumer-Suchmusters

Dieses Suchmuster wird hauptsächlich durch *CentrallBuffer*-Knoten und Pin-Multiplitäten spezifiziert.

1. Es gibt einen Objektknoten mit Stereotyp *CentrallBuffer*.

2. Die eingehenden und ausgehenden Datenflüsse vom gleichen Typ haben unterschiedliche Werte im *Multiplicity*-Attribut.

Die Parallelisierung nach Erzeugen-Verbraucher-Muster macht da Sinn, wo *Multiplicity*-Eigenschaft kleiner ist. So können mehrere Aktivitäten die Rolle des Erzeugers spielen, falls der Verbraucher mehr verbraucht als die Erzeuger produzieren. Oder es können beide Seiten repliziert werden, wenn die Zahlen dies erlauben.

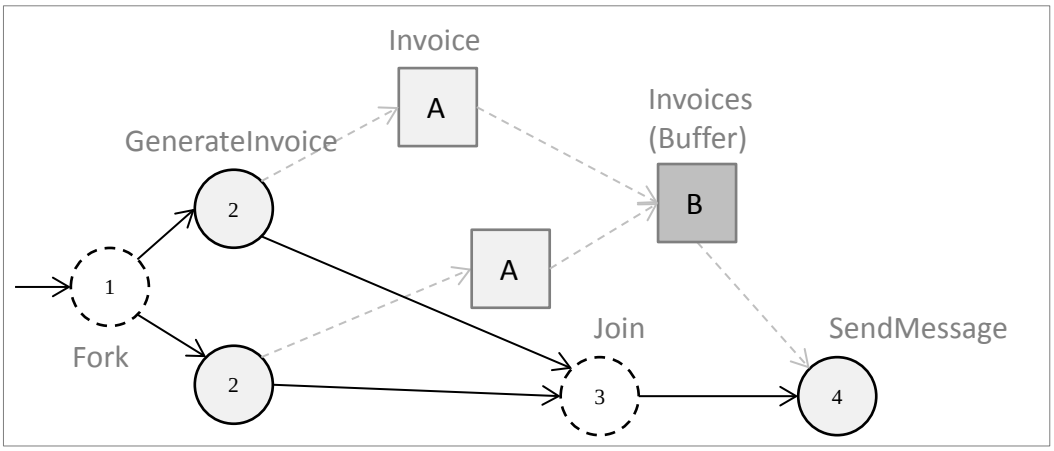


Abbildung 78. AOM: Erzeuger-Verbraucher-Muster mit einem CentrallBuffer

Die Zugriffe auf den Pufferknoten sind kritische Abschnitte und müssen in parallelen Sequenzdiagrammen entsprechend markiert werden (siehe Abbildung 81). Bei der Implementierung soll eine zusätzliche Zugriffsregelung verhindern, dass eine verbrauchende Aktivität auf den Pufferknoten zugreift, wenn er keine Elemente enthält oder die erzeugenden Aktionen mehr Objekte produzieren, als der Pufferknoten speichern kann. Dafür bietet die UML jedoch keine Möglichkeiten. Für den Softwareentwickler wird aber ein Kommentarknoten hinterlegt, der diese Eigenschaften beschreibt.

4.5.1 ParallelLoop und ParallelQueue

Verzweigungen, Schleifen und Sequenzen sind weit verbreitete Muster in allen Abläufen. Die Verzweigungs- (*Decision*) und Verbindungsknoten bieten hierfür eine Struktur, um diese Muster kompakt und übersichtlich in einer Sequenz auszudrücken.

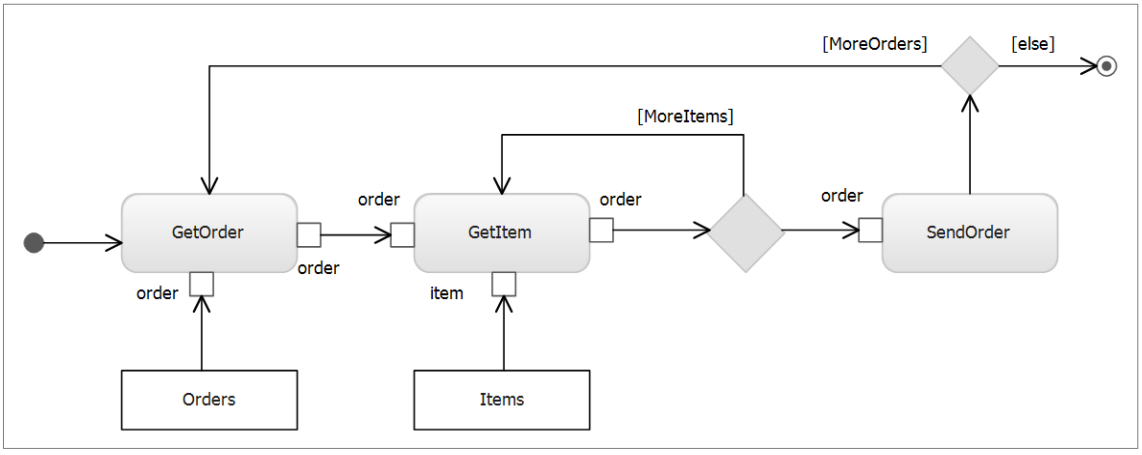


Abbildung 79. UML: Bedingte Schleifen in AD

Die Verzweigungsknoten (Wächter) sind in dem Objektmodell auch von dem Typ *InvocationAction* und enthalten in dem Attribut *Pre-* oder *Postcondition* eine Bedingung 4.3.3. In dieser Arbeit werden die Bedingungen semantisch nicht evaluiert (siehe Einschränkungen 4.2.2) und aus dem UML-Modell übernommen.

Eine Schleife wird parallelisiert, indem die einzelnen Iterationen auf unterschiedlichen Prozessorkernen ausgeführt werden. Dies ist nur möglich, wenn die Iterationen keine kausal bedingte Ausführungsreihenfolge haben (unabhängig voneinander sind).

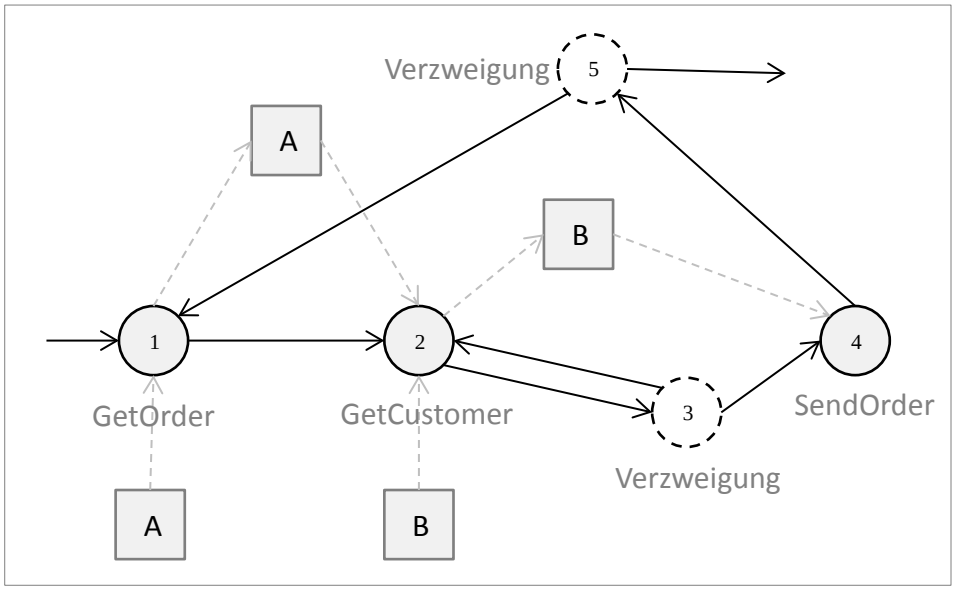


Abbildung 80. AOM: Bedingten Schleifen in dem Objektmodell

Die Anzahl der Iterationen soll auch relativ groß sein, sonst bringt die Parallelisierung kaum Leistungsvorteile. Das wichtigste Ziel beim Parallelisieren von Schleifen ist die Prozessorleistung möglichst umfassend auszunutzen, ohne dass der mit der parallelen Verarbeitung verbundene Mehraufwand die Leistungsvorteile negiert. In dem obigen Beispiel wird nur die äußere Schleife parallelisiert, da in der inneren Schleife nur wenige Aufgaben ausgeführt werden. Die Kombination aus wenigen Aufgaben und unerwünschten Auswirkungen auf den Cache können in geschachtelten parallelen Schleifen zu Leistungseinbußen führen. Daher ist eine Parallelisierung der äußeren Schleife die beste Möglichkeit, die Vorteile der Parallelität in den meisten Systemen zu maximieren.

In der UML gibt es keine expliziten Mittel für die Darstellung von parallelen Schleifen. In einem Aktivitätsdiagramm können parallele Prozesse modelliert werden und ein Sequenzdiagramm mit kombinierten Fragmenten kann die parallelen und kritischen Bereiche beschreiben, aber keine replizierbaren Fragmente definieren (Abbildung 81).

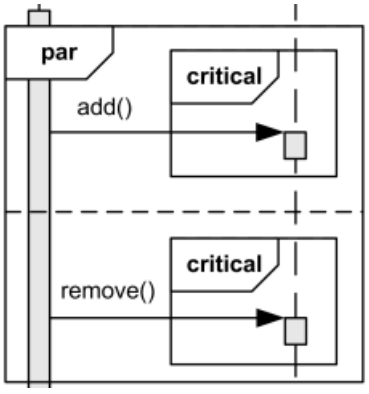


Abbildung 81. UML: Parallele und kritische Bereiche in einem SD

Die einzige Möglichkeit eine Schleife zu parallelisieren, ist eine spezielle Notation für den Entwickler, dass die Iterationen dieser Schleife replizierbar sind. Solch eine Notation kann in Form eines Kommentars zu dem Wächter-Knoten in einem Aktivitätsdiagramm hinzugefügt werden.

4.6 A4: Erzeugung der parallelen Modelle aus dem Objektmodell

Nach der Modifikation des Objektmodells sollen neue parallele UML-Modelle daraus generiert werden. Über eine Schnittstelle werden dabei ein UML-Modell und mehrere UML-Diagramme als Teilansicht dieses Modells erzeugt. Es sind zwei unterschiedliche Szenarien möglich: eine komplett neue Generierung von parallelen UML-Modellen und eine Modifikation der bestehenden Modelle.

Wenn man sich für die Generierung von komplett neuen parallelen Modellen entscheidet, dann steht man vor folgenden Herausforderungen:

- Es soll keine Information verloren gehen, die in den ursprünglichen Modellen vorhanden war. Alle UML-Attribute und Notationen sollen im Objektmodell gespeichert werden. Das Objektmodell wird somit von der UML-Implementierung abhängig sein und der Suchraum wird nicht kleiner. Das ist ein Nachteil, da dadurch die Anforderung A_1 verletzt wird

- Anzahl der Modelle ist unklar, da in dem AOM alle Informationen in einem Graph gespeichert sind. Mögliche Lösung – man erzeugt pro Graphen im AOM jeweils ein UML-Modellsatz (AD/SD/CD).

- Die UML Erweiterungen, wie zum Beispiel Profile, müssen auch unterstützt werden.

In dieser Arbeit wird der zweite Ansatz verwendet: Die ursprünglichen Modelle werden kopiert und abschließend modifiziert. Dadurch werden keine Anforderungen verletzt und der Benutzer erkennt die Modelle wieder, da die räumliche Komposition von Elementen im Modell nicht verletzt wird. Die neuen parallelen Muster werden an bestimmten Stellen eingebaut, so dass man die alten und neuen Modelle auch optisch leicht vergleichen kann.

Der Erzeugungsprozess ist dem Einlesen sehr ähnlich. Die Attribute und Beziehungen werden in der UML übernommen. Die neuen Elemente, die durch Modifikation des Objektmodells entstanden sind, werden in den entsprechenden Kontrollflussknoten umgewandelt. Eine prototypische Implementierung dieses Prozesses wird in Kapitel 5 vorgestellt. Dafür wird eine Schnittstelle mit Hilfe der UML-API von Microsoft Visual Studio implementiert. Zusammen bilden mehrere Modelle einen Packet, der als Visual Studio Modellierungsprojekt gespeichert wird. Die Implementierung kann sich in der Zukunft ändern, sodass auch weitere Diagramme oder Versionen unterstützt werden.

4.7 Zusammenfassung

In diesem Kapitel wurde beschrieben, wie eine musterbasierte Modellanalyse in einem abstrakten Objektmodell durchgeführt werden kann. Zunächst wurden in Kapitel 4.1 einige wichtige UML-Begriffe, sowie eigene Definitionen erklärt. Danach wurden das Ziel und die Anforderungen dieser Arbeit definiert 4.2 und mit anderen verwandten Arbeiten aus Kapitel 3 verglichen. Der Kernpunkt dieser Arbeit, nämlich ein abstraktes Objektmodell (AOM), das gewisse Informationen über den Kontroll- und Datenfluss, sowie Objekt- und Typknoten aus unterschiedlichen UML-Diagrammen in sich verbindet wurde im Kapitel 4.3 beschrieben. Die musterbasierte Transformationen 4.4 wurden ebenfalls ausführlich beschrieben und anhand von vielen Beispielen erklärt. Dabei lag der Fokus auf die Zusammenarbeit zwischen den Objekten und Methoden im Modell. Zum Schluss werden fünf Suchmuster vorgestellt, die ein Parallelisierungspotenzial haben und in dem AOM gesucht und ersetzt werden 4.5.

Durch den Aufbau eines abstrakten Objektmodells (AOM) aus den vorliegenden UML-Modellen wurde eine unabhängige Basis für die weiteren Analysen und Transformationen geschaffen. AOM hat sein eigenes Metamodell, das durch eine Erweiterung des Metamodells aus [Stö05] entstanden ist. Der Ansatz ist flexibel und kann durch neue Suchmuster und Ersetzungsregeln beliebig erweitert und angepasst werden.

Die in folgendem Kapitel vorgestellte Implementierung zeigt die Umsetzbarkeit des Ansatzes und die Möglichkeiten, die sich für Softwareentwickler dadurch ergeben.

5 IMPLEMENTIERUNG: AUTOMATISCHE MODELLBASIERTE PARALLELISIERUNG (MAP)

Nach der formalen Spezifikation des kombinierten Objektmodells AOM im vorherigen Kapitel wird in diesem Kapitel der Aufbau des MAP Werkzeugs und die konkrete Umsetzung mit dem GrGen.NET vorgestellt. Hierbei wird insbesondere auf die Transformationsregeln für in Kapitel 4.5 definierten Suchmuster eingegangen. Dies ist nötig, um die im Rahmen der Parallelisierung durchführenden Transformationen nachzuvollziehen.

Im Kapitel 5.1 wird zunächst der allgemeine Aufbau des MAPs-Werkzeugs erklärt und seine Oberfläche vorgestellt. Das Werkzeug hat einen modulareren Aufbau und besteht insgesamt aus vier Modulen, die eine jeweilige Bearbeitungsstufe durchführen. Im Kapitel 5.2 wird die Implementierung der ersten Stufe, dass Einlesen von UML-Modellen mit Hilfe Microsoft UML API, vorgestellt. Die in Kapitel 4.5 definierten Suchmuster und Ersetzungsregeln, sowie das gesamte Metamodell des AOMs werden durch eigene deklarative GrGen-Sprache implementiert und in .NET-Assemblies übersetzt 5.4. Mit diesen Bibliotheken kann das Objektmodell AOM modifiziert werden. Abschließend wird in Kapitel 5.5 Möglichkeiten zur Generierung von neuen parallelen UML-Modellen vorgestellt.

5.1 Grundlagen und Aufbau des MAPs

Heutzutage existieren viele verschiedene Entwicklungsumgebungen und Tools für die Entwicklung von UML-Modellen. Die Firma Microsoft bietet mit der Visual Studio Entwicklungsumgebung spezielle Projekttypen für die gängigsten UML-Modelle an. Die anderen populären Werkzeuge und Umgebungen haben ihre eigenen Implementierungen. Diese Produkte bieten Modellierungsmöglichkeiten, Transformationen und Codegenerierung in einem. Es gibt auch eine Menge von reinen Modellierungswerkzeugen, die keine automatischen Transformationen unterstützen. Als Standardformat für Speicherung und Austausch von Modellen hat sich das **XML Metadata Interchange** (XMI) von Object Management Group [OMG] etabliert. Das Format ist offen und anbieterneutral und gestattet den Datenaustausch von Objekten auf Basis von Meta-Metamodellen [XMI].

Der in dieser Arbeit entwickelte Ansatz nutzt für die Analyse und Transformation von Modellen sein eigenes abstraktes Objektmodells namens AOM. Somit ist der MAP-Ansatz von der Implementierung der UML-Modellen unabhängig. Die Elemente aus UML-Klassen-, Sequenz- und Aktivitätsdiagrammen von anderen Tools können aus XMI 2.1 oder Microsoft UML Dateien mit Hilfe des Microsoft UML API eingelesen werden. Es ist jedoch zu beachten, dass eine XMI-Datei einige Elemente enthalten kann, die über benutzerdefinierte Profile verfügen. Beim Importieren einer XMI-Datei müssen diese Profile installiert werden.

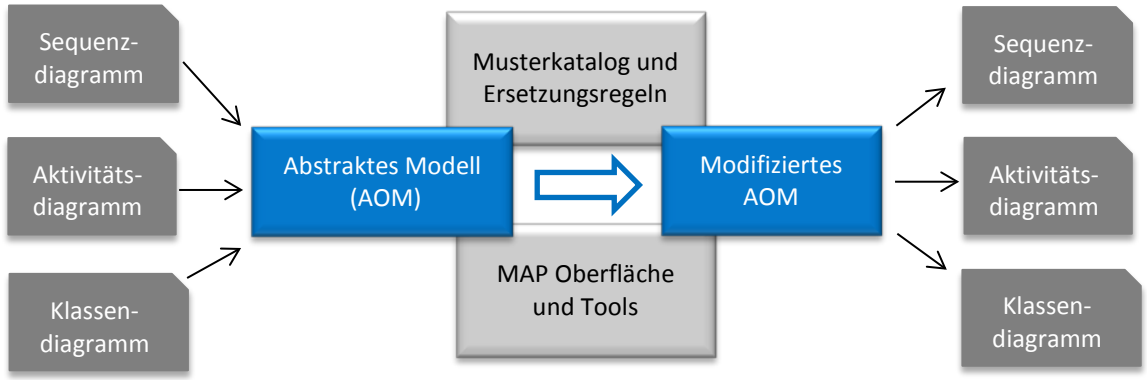


Abbildung 82: Arbeitsschema von MAP

Der MAP Ansatz besteht aus vier Stufen, die in Kapitel 4 bereits beschrieben wurden. Die erste und die letzte Stufe (Einlesen und Generieren von UML Diagrammen) sind von der Mustersuche und Transformation unabhängig und können leicht geändert werden, um z.B. mit anderen UML Implementierungen arbeiten zu können. Abbildung 83 zeigt einen schematischen Aufbau des MAPs.

Die vom Werkzeug entdeckten Suchmuster werden im AOM durch ihre parallelen Varianten mit Hilfe eines Graphersetzungssystem GrGen.NET gefunden und ersetzt. So entstehen neue parallelisierte UML-Modelle, die dem Entwickler seine weitere Arbeit erleichtern sollen. Diese Arbeit setzt voraus, dass alle Modelle zu einander konform und wiederspruchfrei sind. Eine weitere Voraussetzung ist, dass die Änderungen an Variablen, nur über Methodenaufrufe passieren und diese sollen in den vorliegenden Diagrammen vorhanden sein (siehe 4.2.2).

Die MAP-Oberfläche (Abbildung 83) bildet alle wichtigen Ansatzschritte ab. Zunächst kann ein Projekt mit UML-Modellen gewählt werden. Die vorhandenen Diagramme werden aufgelistet und es besteht die Möglichkeit zu wählen, wie modifiziert werden sollen (1). Danach kann das AOM gebaut werden. Der Aufbauprozess wird protokoliert und in (2) ausgegebenen. Auf der rechten Seite werden die gefundenen Suchmuster angezeigt (3). Es kann außerdem das Objektmodell AOM vor und nach der Modifikation anzeigen werden (4). Dafür wird es als Graphen in einer VCG-Datei gespeichert und mit Hilfe der yComp-Anwendung visualisiert.

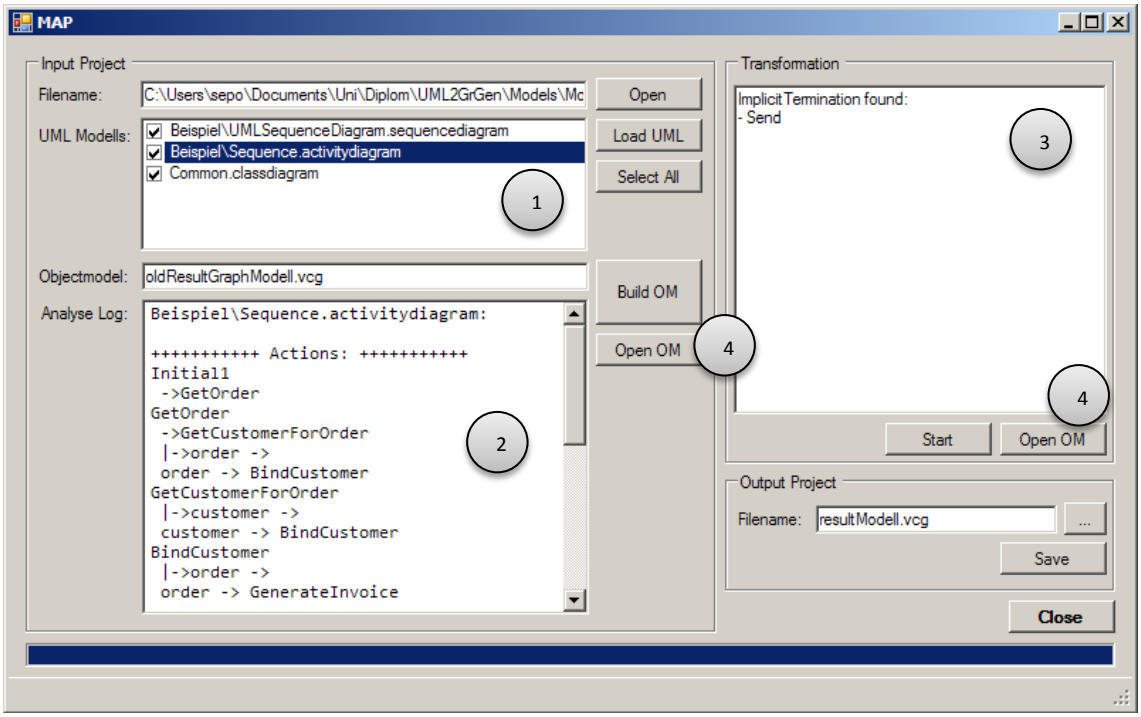


Abbildung 83: MAP Oberfläche

Für die Implementierung wurde das .NET-Framework 4.5 verwendet. Die Oberfläche wurde mit WinForms umgesetzt.

5.2 Einlesen und Parsen von UML-Modellen

Das MAP arbeitet mit seinem eigenen Modellformat und ist von dem UML-Format unabhängig. Als Grundlage für die Implementierung wird die UML API von der Firma Microsoft verwendet und entsprechend erweitert. Die Modelle können zum Beispiel mit MS Visual Studio oder auch mit Model Development Tools (MDT) in Eclipse entwickelt werden und in Form von XMI Dateien vorliegen. Durch einen Parser werden die vorliegenden Modelle eingelesen und für die Analyse benötigte Daten extrahiert. Folgende Abbildung illustriert diesen Prozess:

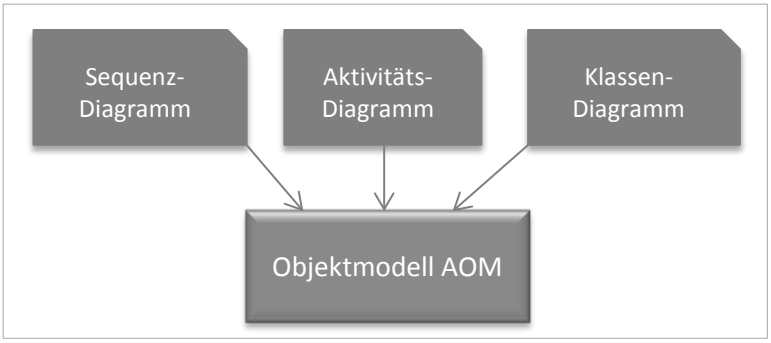


Abbildung 84: Kombiniertes Objektmodel

Die Microsoft UML API definiert die grundlegenden Modellierungskonzepte in UML. Die in der *Microsoft.VisualStudio.Uml.Interfaces.dll* Assembly definierten Typen entsprechen den Typen, die in der **UML-Spezifikation 2.1.2** definiert sind. Jeder Typ ist eine Schnittstelle, die den gleichen Namen wie der UML-Typ hat, dem jedoch ein "I" vorangestellt ist. Die obersten Typen sind *IElement* und *IRelationship*, von denen alle anderen UML-Typen abgeleitet sind.

Wie in Kapitel 2.2.1 beschrieben, besteht jedes UML Modell aus Elementen, die über unterschiedliche Beziehungen miteinander verknüpft sind. Im folgenden Abschnitte wird beschrieben, wie das eingelesene UML Modell im Programmcode dargestellt werden kann und wie aus mehreren Modellen ein kombinierter Graph aufgebaut wird. Die Klasse *MedellingProject* repräsentiert ein UML Projekt bzw. ein UML Modell. Die Methode *LoadReadOnly* kann Inhalt eines Models in einem Lese-Modus zuzugreifen.

```
1:  public static class ModelingProject
2:  {
3:      // The name of the file extension used for layer models.
4:      public string LayerModelExtension = ".layerdiagram";

5:      // The file extension used for modeling project files.
6:      public string ModelDefinitionFileExtension = ".uml";

7:      // The name of the folder where package files are stored.
8:      public string ModelDefinitionFolderName = "ModelDefinition";

9:     // Open a modeling project in read only mode
10:     public static IModelingProjectReader LoadReadOnly(string
        projectFileName, IServiceProvider serviceProvider = null);
11: }
```

Das Einlesen der UML-Modelle passiert in drei Phasen. Jede Phase benötigt die Daten aus der vorherigen Phase und fügt neue Daten in das Objektmodell hinzu. In jeder Phase können mehrere Diagramme bearbeitet und ein einem Objektmodell gespeichert werden. Falls die Daten sich überschneiden (zum Beispiel in verschiedenen Diagrammen wird eine Klasse mehrmals deklariert), so wird die erste Variante genommen und nicht überschrieben. Dieser Ansatz setzt die Konsistenz der Modelle voraus (siehe Kapitel 4.2.2).

5.2.1 Phase 1: Klassendiagramme einlesen

Zuerst werden die Klassendiagramme geladen. Sie bilden eine statische Grundlage für das Objektmodell und definieren Datentypen von Objekten, als auch die Methodenrumpfe mit Parameter. Diese Phase besteht aus folgenden Schritten:

1. Klassen laden und auf Type-Knoten im AOM abbilden (Abbildung 85, links)
2. Passive und aktive Klassen unterscheiden und entsprechendes Attribut setzen
3. Methoden und deren Parameter bestimmen und in einem Stapel speichern
4. Multiplizitäten bestimmen

5.2.2 Phase 2: Aktivitätsdiagramme einlesen

Nach dem Einlesen von statischen Modellen werden die dynamischen Modelle (Aktivitäts- und Sequenzdiagrammen) dazu geladen. Die Aktivitätsdiagramme bilden eine Komposition aus Aktionen (Methoden) und Objekten und legen die Verbindungen fest. Der Kontrollfluss wird im Objektmodell AOM, ähnlich dem Kontrollfluss in AD, abgebildet. Bei der Abbildung des Datenflusses gibt es allerdings einige Unterschiede, die bereits in Kapitel 4.3 beschrieben wurden. Die Phase 2 besteht aus folgenden Schritten:

1. Aktionen und Kontrollknoten auf entsprechenden Knoten im AOM abbilden (siehe Abbildung 86 rechts)
2. Pins und Objektknoten auf Objektknoten in Objektmodell abbilden
3. Kanten, die Kontroll- und Datenfluss (inkl. Mulziplitäten) festlegen

5.2.3 Phase 3: Sequenzdiagramme einlesen

Zum Schluss wird das Objektmodell durch zeitliche Aspekte aus den Sequenzdiagrammen ergänzt. Der Arbeitsgraph, der nach der zweiten Phase entsteht, ist eine Komposition aus mehreren Aktivitäts- und Objektknoten sowie Flüssen. Dieser beinhaltet jedoch keine Informationen über die Ausführungsreihenfolgen und Kantentypen (synchrone oder asynchrone Ausruf). Um das Objektmodell durch diese Aspekte zu vervollständigen werden folgende Schritte unternommen:

1. Ausführungsreihenfolge der Aktivitäten ablesen und im AOM speichern (siehe Zahlen in Aktivitätsknoten)
2. Synchrone/Asynchrone Aufrufe bestimmen und die Kanten entsprechend ändern
3. Timelines auf aktive Objekte abbilden und mit Aktivitäten verbinden

Folgende Abbildung zeigt Elemente des AOMs nach diesen drei Phasen. Aktive Objekte und die Typknoten (links im Bild) werden normalerweise nicht visualisiert und sind hier nur für die Verständlichkeit dargestellt:

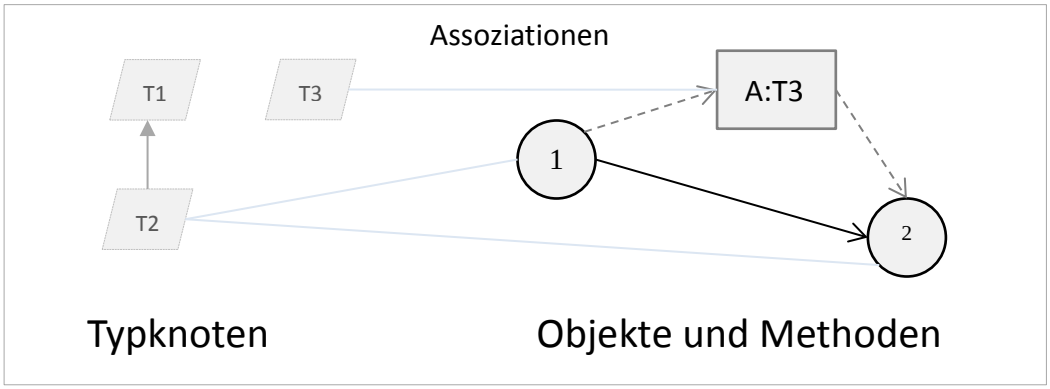


Abbildung 85: Aufbaus des AOMs

5.3 Modelltransformation durch Musterersetzung mit GrGen.NET

In diesem Kapitel werden verschiedene Ansätze zur Modelltransformation durch Musterersetzung vorgestellt und diskutiert. Sie werden mit der, in dieser Arbeit verwendeten Graphenersetzungssystem GrGen.NET, verglichen.

Für die Modelltransformation existieren heutzutage viele unterschiedliche Ansätze und Methoden. Czarnecki und Helsen [CH03] haben in ihrer Arbeit verschiedene Konzepte analysiert und verglichen. Sie kommen zum Ergebnis, dass die Modelltransformation ein neues und junges Gebiet ist und dafür noch keine befriedigende Lösung zur Transformation von komplexeren Modellen besteht. Die verschiedenen Ansätze zur Modell-zu-Modell-Transformation wie z. B. direkte Manipulation, relationaler Ansatz und der graphenbasierte Ansatz sowie hybride Formen besitzen Vor- wie auch Nachteile. Nach Czarnecki und Helsen bietet der graphenbasierte Ansatz die größtmögliche Flexibilität und Ausdrucksmächtigkeit. Damit verbunden ist jedoch eine steigende Komplexität der Transformationsdefinition und eine erforderliche Erfahrung in der Regelbildung [CH03].

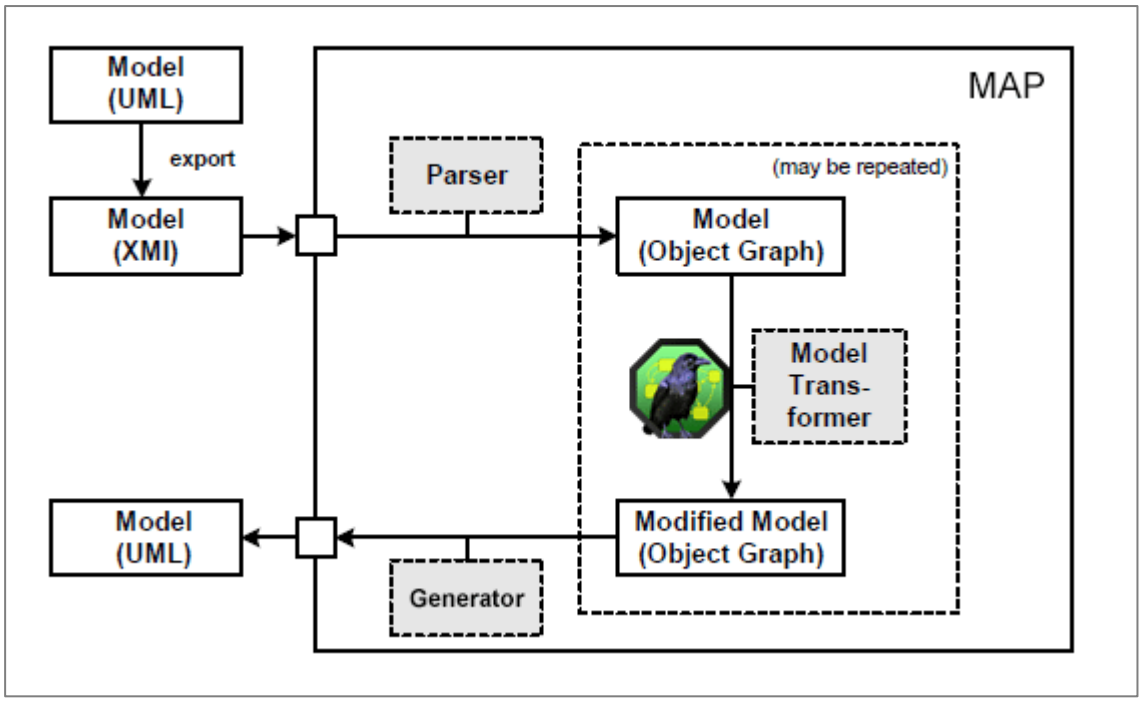


Abbildung 86: Modeltransformation mit GrGen.NET

Im Rahmen dieser Arbeit wird der graphenbasierte Ansatz verwendet, um das kombinierte Modell, das aus mehreren Graphen besteht, zu transformieren (siehe Abbildung 86). Dafür wird das **Graphersetzungssystem GrGen.NET** verwendet. GrGen.NET unterstützt benutzerdefinierte Metamodelle sowie die Mehrfachvererbung für Knoten und Kanten. Graph-Elemente werden durch Aktivitäten (Methoden) sowie aktiven und passiven Objekten aus UML-Modellen repräsentiert. Die Kanten können auch typisiert werden, um Kontroll- und Datenfluss zwischen Aktivitäten zu beschreiben. In dieser Arbeit werden alle Zugriffe nach [Woh05b] in drei Gruppen aufgeteilt. Eine ähnliche Aufteilung finden man in anderen Arbeiten auch, wie zum Beispiel in [PF+04].

Die Regeln für Mustersuche und Musterersetzung werden in einer eigenen Regelsprache formuliert. Für die Mustersuche können negative Anwendungsbedingungen sowie Typeinschränkungen verwendet werden. Somit lassen sich zum Beispiel die Datenabhängigkeiten beschreiben, die eine Parallelisierung verhindern können.

Eine mögliche Alternative zum graphenbasierten Ansatz wäre **Query View Transformation** (QVT) von der Object Management Group (OMG). Die QVT Transformationsregeln werden auf der Basis von Metamodellen definiert. Allerdings ist die QVT nur ein Standard mit vielen unterschiedlichen Implementierungen, die zuerst untersucht werden müssen. Eine Implementierung für Java ist zum Beispiel die Atlas Transformation Language (ATL) von ATLAS INRIA & LINA. ATL ist ein Modelltransformations-Toolkit auf Basis des Eclipse Modeling Framework. Die Lösung besteht aus zwei Schritten: zuerst findet die Transformation der abstrakten Syntax mit ATL, danach folgt die Transformation der konkreten Syntax mit Java.

Eine aktuelle Studie "*Graph and model transformation tools for model migration*" [RHM12] vergleicht beide Ansätze und untersucht die bekannte Werkzeuge dafür, unteranderem GrGen.NET und ATL. Laut dieser Studie bietet GrGen.NET eine voll funktionsfähige Metamodell und ausdrucksstarken Regeln, die sehr ausgeführt werden können. Eine programmierte Regel-Steuerung und grafische Debugging gehören auch zu den Vorteilen von GrGen.NET. In GrGen.NET werden Modelle in Form von gewöhnlichen Graphen definiert, die sich mit Hilfe von deklarativen Graphersetzungsregeln transformieren lassen. Dabei ist der Anzahl der Ersetzungsregeln gleich den Anzahl der gesuchten Muster und ist unabhängig von der Modellgroße. Ein weiterer Vorteil von GrGen.NET ist, dass die Ablaufsteuerung der Regelanwendung durch eine .NET-Schnittstelle steuern lässt und kann kontrolliert ablaufen.

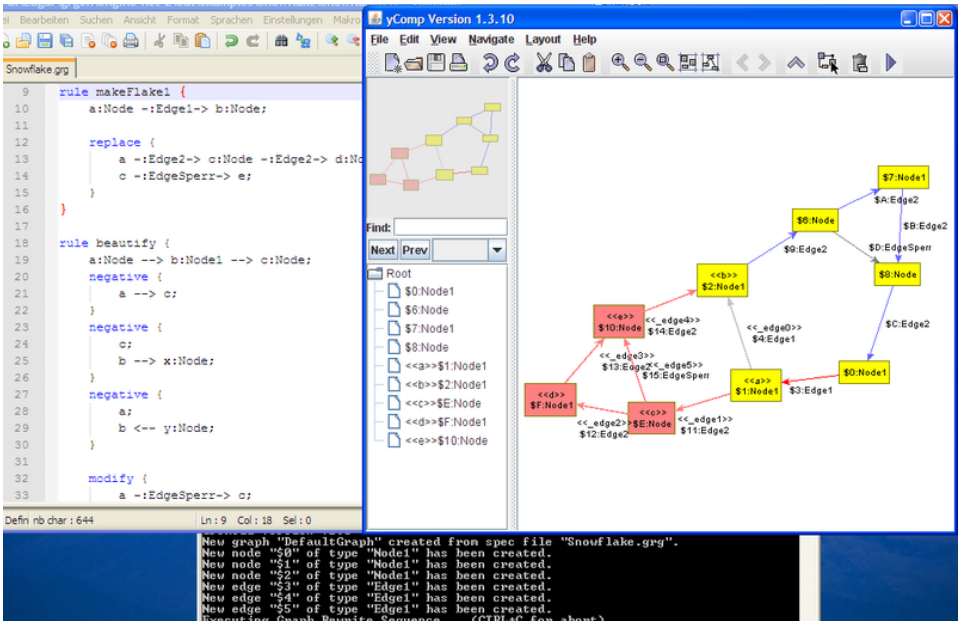


Abbildung 87: Ausführung des Ersetzungsschrittes in GrGen.NET (Beispiel)

Eine Transformation in ATL wird für jede Klasse von der Metamodell definiert und erlaubt nicht die Änderungen höherer Form (z.B. Umbenennung dieser Klasse). Weiterhin ist die Größe des Regelwerks proportional zu der Größe der Metamodelle und nicht zu der Differenz zwischen Ziel- und Quellmodellen. Die Mächtigkeit und Flexibilität den graphenbasierten Ansatz, sowie eine solide praktische Erfahrung mit GrGen.NET am Lehrstuhl IPD Tichy sind dazu geführt, dass die Entscheidung für GrGen.NET gefallen ist.

In kommenden Kapiteln werden einzelne Muster durch die Regeln in GrGen.NET beschrieben und erkläret.

5.4 Implementierung des Objektmodells mit GrGen.NET

Dieses Kapitel beschreibt den Aufbau des kombinierten abstrakten Modells, dass die Informationen aus UML Diagrammen in einem Graph vereinigen. Das in dieser Arbeit entwickeltes Objekt-Modell übernimmt viele Konzepte aus der *Diagram Interchange* (DI) Standart der Object Management Group [OMG]. Es wird davon ausgegangen, dass alle Diagramme als Graphen betrachtet werden können. Dabei können alle Elemente als Knoten und Kanten eines Graphen beschreiben werden. Alle Elemente sind typisiert, und jedes Element hat gleiches Type wie der Element des semantischen Modells der UML mit dem er verbunden ist. So ist eine Klasse oder ein Aktion der UML auch nur ein Knoten im Objektmodellgraph.

Ein weiteres grundlegendes Konzept des DI ist seine Verschachtelungsstruktur. Knoten und Kanten können beliebige Knoten und Kanten enthalten, so dass beispielsweise eine Klasse als ein Knoten dargestellt wird, der weitere Knoten enthält. Eine Verschachtelung ist im GrGen.NET-Graphen nicht möglich. Damit die Verbindungen zwischen Modellelementen nicht verloren gehen werden sie durch spezielle Kanten vom Type *Assosiate* miteinander verbunden. Es werden Objekt-Knoten mit entsprechenden Typ-Knoten verbunden und die aktiven Objekten mit ihren Methoden. Diese Kannten können auch weitere Informationen in Form von Attributen speichern.

5.4.1 Das Metamodell

Der Kern des Objektmodells AOM besteht aus einer Modelldatei und einer Datei für die grundlegenden Graphersetzungsregeln (mehr dazu später in Kapitel 5.4.2). Das Metamodell wurde bereits in Kapitel 4.3 vorgestellt. Nun wird dessen Implementierung in GrGen.NET Sprache vorgestellt.

Modelldatei

In der Datei (ObjektModell.gm) wird definiert, aus welchen Knoten und Kantentypen ein Graph besteht und welche Attribute diese Graphelemente besitzen. Außerdem werden hier die verschiedenen Kantentypen definiert. Im Kapitel 4.3 wurden die Grundlegenden Modellelemente aus dem Meta-Modell beschrieben. In diesem Abschnitt wird gezeigt wie diese Elemente mit GrGen.NET instanziiert und für Graphentransformation verwendet werden können.

Zuerst werden grundlegenden Typen für Knoten im Objektmodel AOM definiert:

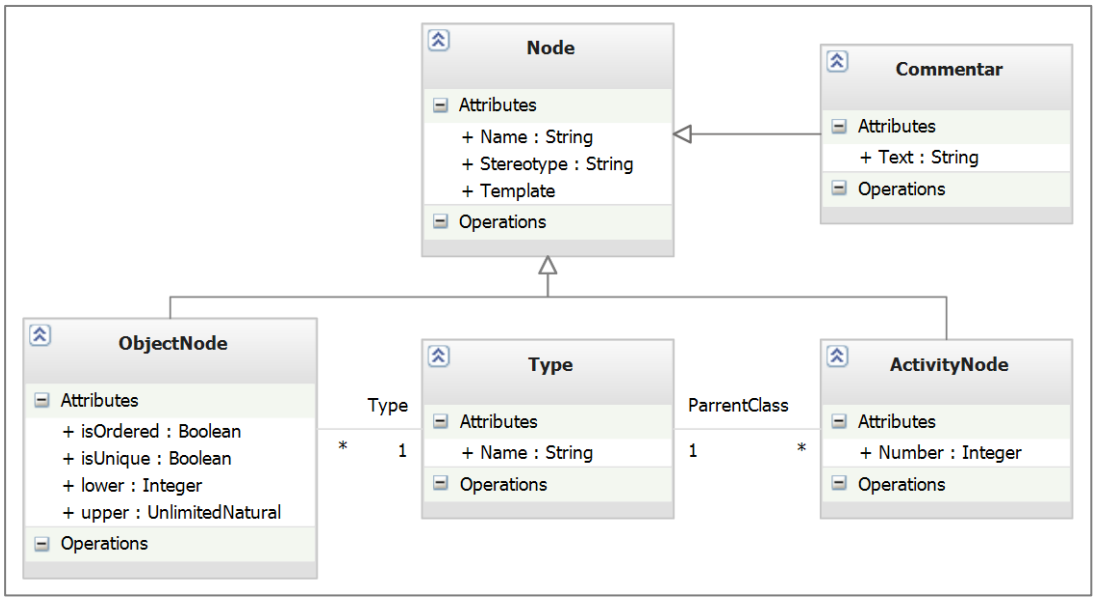


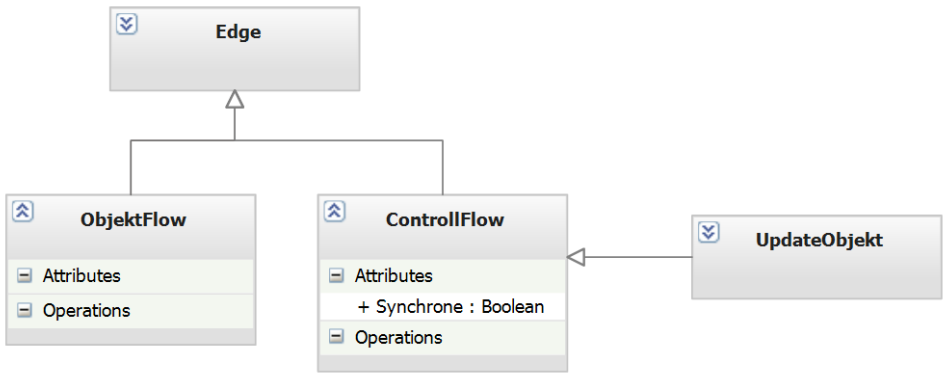
Abbildung 88: Ausschnitt aus dem Meta-Modell

Diesen Knoten werden im ObjectModell.gm wie folgt definiert:

```
1:  abstract node class uNode {
2:   Id: int;
3:   Name: string;
4:   Stereotype: string;
5:  }

6:  node class Commerntar extends uNode {
7:   Text: string;
8:  }

9:  node class ObjectNode extends uNode {
10:   isOrdered: boolean;
11:   isUnique: boolean;
12:   lower: int;
13:   upper: int;
14:  }

15:  node class ActivityNode extends uNode {
16:   Number: int;
17:  }
```

In der Modelldatei kann die Vererbung (wie in objektorientierten Sprachen) genutzt werden, um weitere Knotentypen zu definieren (Schlüsselwort *extends*). Die Knoten, die als *abstract* markiert sind, können nicht direkt instanziiert werden und dienen nur als abstrakter Elterntyp für weitere Knoten. Laut [GrGen] kann GeGen.NET auch die Mehrfachvererbung unterstützen, was aber für diese Arbeit eher irrelevant ist. Die weiteren Knotentypen werden analog definiert.

Das zweite große Teil der Modelldatei beschreibt die Kanten aus dem Meta-Modell. Alle Kanten werden von dem Type *uEdge* abgeleitet:




```
1:  edge class uEdge;

2:  edge class uAssociation extends uEdge
3:       connect uActiveObject[*] -- uActivityNode[*];

4:  edge class uObjectFlow extends uEdge;
5:   connect uObjectNode[*] --> uActivityNode[*],  // read object
6:  uObjectNode[*] <-- uActivityNode[*];           // write into object

7:  edge class uControllFlow extends uEdge;
8:  edge class uUpdateObject extends uControllFlow;
```

Jedes Knoten-Objekt besitzt in GrGen.NET zwei Referenzen auf Kanten - eine für ausgehende Kanten und eine für eingehende Kanten. Wie aus dem Meta-Modell ersichtlich 4.3.1, verbindet eine *ObjectFlow*-Kante einen passiven Objekt mit einem Aktion oder umgekehrt. Außerdem können auch zwei Objektknoten damit verbunden werden, wenn ein Pufferknoten mit unterschiedlichen In- und Output-Multiplizitäten modelliert wird. Diese Eigenschaften (Assoziationen) werden durch das Schlüsselwort *connect* modelliert. Damit können die Knotenpaare und deren Multiplizitäten definiert werden, die diese Kante verbinden darf. Im Objektmodel AOM haben alle Aktivitäten den Rang 1, d.h. sie repräsentieren nur eine Kopie (siehe 4.2.2 Einschränkungen). Deswegen haben die *ControllFlow*-Kanten den Multiplizität gleich eins. Um mehrere gleiche Aktivitäten zum Beispiel parallel ausführen zu können müssen sie im Objektmodell explizit dargestellt werden. Dafür werden selbstverständlich mehrere aktive Objekte nötig. Eine Aktion hat immer ein aktives Objekt, zu dem er auch gehört.

Validierung und Konsistenzprüfung

GrGen.NET bietet zwei verschiedene Mechanismen zur Validierung des Arbeitsgraphs. Es können die im Objektmodel AOM mit dem Schlüsselwort *connect* deklarierten Verbindungen und Multiplizitäten überprüft werden. GrGen.NET erlaubt zunächst die Kanten anzulegen, die diese Bedingungen nicht erfühlen Bietet aber die Möglichkeit mit dem Befehl *validate* die Korrektheit zu überprüfen. Falls der Arbeitsgraph nicht valide ist, liefert GrGen die Kanten und damit verbundenen Knoten zurück, die dem Meta-Modell nicht entsprechen. Das Objektmodel wird so zwei Mal überprüft: gleich nach dem Aufbau und nochmal nach der Modifikation.

Die zweite Möglichkeit ist die Verwendung von Tests und Regeln, die in der Regeldatei deklariert sind. Damit können Attribute, Anzahl der Kanten und weitere komplexere Parameter überprüft werden. So wird im *MergeObjekt*-Suchmuster zunächst anhand der Anzahl eingehender Datenflüssen überprüft und die Transformation wird nur dann durchgeführt, wenn diese Anzahl größer als zwei ist.

GrShell Werkzeug

Alle Modellelemente sind nun definiert und man kann mit Hilfe einer GrShell-Konsolanwendung einen Graph bilden. Der GrShell bietet eine deklarative Skriptsprache, um neue Graphen zu erzeugen und zu testen. Im Folgenden Beispiel wird eine einfache Sequenz aus zwei Aktivitäten mit einem Objekt-Fluss dazwischen definiert (siehe [4.4.1] und Abbildung 44).

```
1: new graph Suchmuster "Seq"

2: // Knoten
3: new :ActivityNode($=a1, id=1)
4: new :ActivityNode($=a2, id=2)
5: new :ObjectNode($=o1, id=1)

6: // Kanten
7: new @(a1)-:ControllFlow-> @(a2)
8: new @(a1)-:ObjectFlow-> @(o1)
9: new @(o1)-:ObjectFlow-> @(a2)
```

Beim ersten Start der GrGen-Anwendung werden auch zwei .NET-Assemblyes generiert, die das Meta-Modell und die Suchschablonen für .NET-Anwendungen beinhalten. Die Verwendung von diesen Assebliyes wird in Kapitel [5.4] vorgestellt.

Es besteht auch die Möglichkeit den erzeugten Graphen in VCG Format zu speichern und mit Hilfe der yComp-Anwendung zu visualisieren. Die Visualisierung kann mit folgenden Befehle gesteuert werden: *dump graph*, *show graph*, und *enable debug*. Für verschiedenen Knoten

können Hintergrundfarbe (*color*), Schriftfarbe (textcolor) und die Form (shape) angegeben werden. Folgendes Schema zeigt der Syntax dieser Befehle:

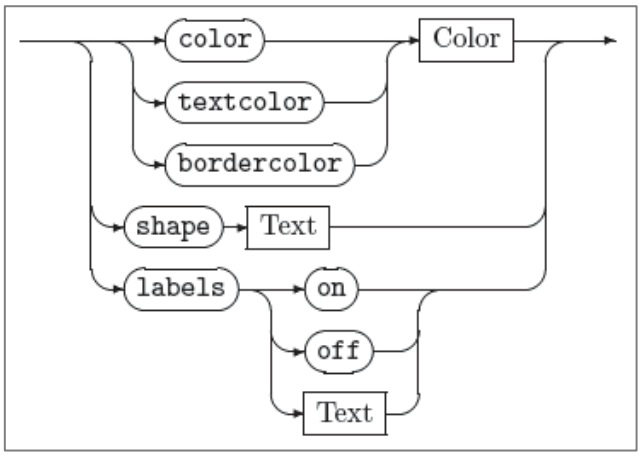


Abbildung 89: DumpNodeContinuation-Shema für GrShell aus [GrGen]

Um den Wiedererkennungswert der Suchmuster zu erhöhen, wird hier, wie in Kapitel 4 vorgeschlagen, grafische Notation bis auf wenige Unterschiede auch verwendet. Im yComp werden nur Knoten- und Kantennamen sondern auch deren Typ angezeigt. Zunächst werden Grundformen und Farben wie folgt definiert:

```
1:  dump set node uNode bordercolor "grey"
2:  dump set node uNode color "white"
3:  dump set edge uEdge color "grey"
```

Die Vererbungshierarchie gilt hier auch, so dass für die Kindertypen nur die Attribute definiert werden müssen, die sich wirklich unterscheiden:

```
4:  dump set node ActivityNode shape "circle"
5:  dump set node ActivityNode color "white"

6:  dump set node ComputationAction color "lightgrey"
7:  dump set node ReadWriteAction color "grey"
8:  dump set node InvocationAction bordercolor "lightgrey"

9:  dump set edge ObjectFlow linestyle "dashed"
10: dump set edge ObjectFlow color "lightgrey"
```

Dadurch bekommt oben definierte Sequenz folgender Gestalt:

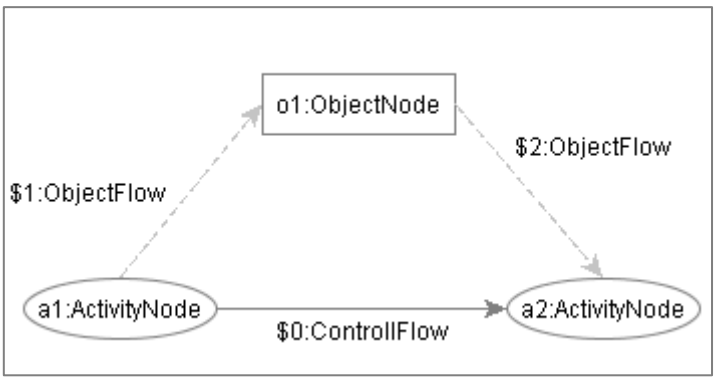


Abbildung 90: Visualisierung eines Datenflüsses mit GrShell

5.4.2 Die Suchmuster

In diesem Abschnitt wird die Implementierung von einigen Suchmustern aus dem Kapitel 4.5 vorgestellt und anhand eines Beispiels illustriert. Dafür werden graphischen Notationen aus Kapitel 4.3.3 verwendet.

Such- und Ersetzungsregeln

Wie in Kapitel 4.1 bereits beschrieben, wird in dieser Arbeit eine Menge von Such- und Ersetzungsregeln definiert. Diese Regeln beschreiben Suchmuster und ebenso wie diese transformiert werden. Ein Suchmuster besteht aus einer Menge von Knoten und Kanten im AOM, die miteinander verbunden sind (Teilgraph).

Anwendung einer Suchregel in GrGen.NET kann als eine Abbildung $p{:}L \rightarrow R$ beschrieben werden, wobei L die Instanz des Mustergraphen und R das modifizierte Graphen ist. Eine gefundene Instanz heißt Passung (engl. *match*) m von L in G. Formal wird sie durch einen Graphhomomorphismus vom Mustergraphen in den Arbeitsgraphen beschrieben [GrGen].

Wie die Abbildung 92 zeigt, besteht ein Suchmuster in GrGen.NET aus drei Blocken: Suchregeln, negative Regeln und die Ersetzungsregeln. Die Suchregeln beschreiben welche Knoten und Kanten im Suchmuster vorkommen und wie sie miteinander verbunden sind. Negativen Regeln beschreiben dagegen die Knoten und Kanten, die im Suchmuster nicht vorkommen dürfen. Der letzte Block definiert die Transformationen, die auf das gefundene Suchmuster angewendet werden und die Rückgabewerden.

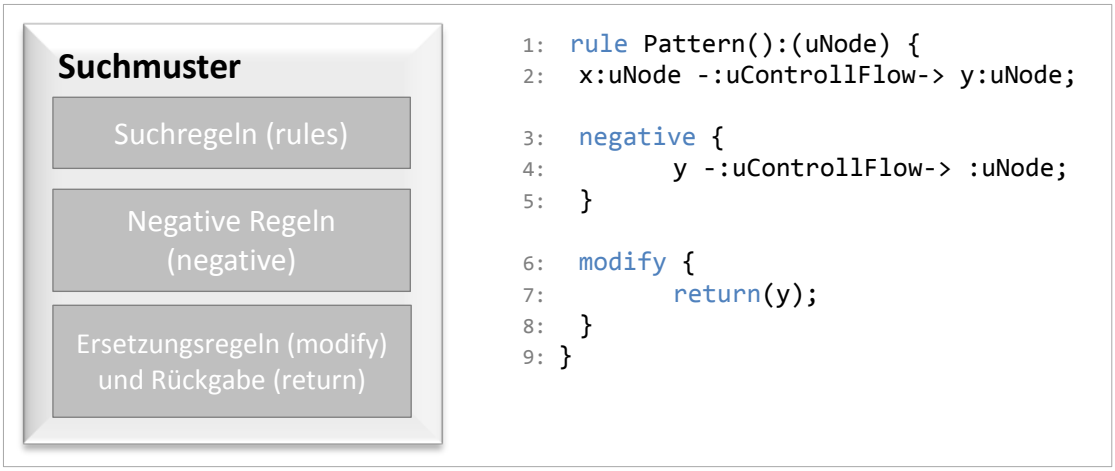


Abbildung 91: Suchmusterbeschreibung in GrGen.NET

Regeldatei

Diese Datei (*Suchschablonen.grg*) dient zur Dentition der Graphersetzungsregeln in GrGen. Diese werden deklarativ angegeben und bilden somit noch kein ausführbares GrGen-NET-Programm, sondern nur eine Laufzeitbibliothek. Das darin deklarierte Modell kann zur Definition von Such und Ersetzungsregeln in einem .NET-Programm verwendet werden.

In den folgenden Absätzen werden die einzelnen Schritte erklärt: Zunächst werden die Suchmuster durch GrGen.NET-Regeln definiert und in einer grg-Datei gespeichert. Danach wird daraus eine .NET-*Assembly* kompiliert, die später mit Hilfe des GrGen.NET API in einem Projekt aufgerufen werden kann. Die Regeldateien bestehen aus der Deklaration von null oder mehrere Graph-Meta-Modelle und einen Satz von Ersetzungsegeln.

In GrGen.NET Programmierschnittstelle wird jede Ersetzungsregel durch eine eigene Klasse dargestellt, die von der Oberklasse *LGSPAction* erbt. Alle Ersetzungsregeln implementieren

IAction-Schnittstelle und besitzen unterschiedliche Methoden, mit denen einer Mustersuche ausgeführt werden kann. In dieser Arbeit werden folgende Methoden verwendet:

- ***Match:*** Suche nach einer Musterinstanz. Bekommt als Parameter eine Referenz auf den Arbeitsgraph und die Anzahl der maximal zu liefernden Musterinstanzen.
- ***Modify***: Wende eine Musterinstanz auf den Arbeitsgraphen an. Dabei wird der Arbeitsgraph auch entsprechend modifiziert.
- ***ModifyAll***: Wende alle übergebenen Musterinstanzen auf den Arbeitsgraphen an.
- ***Apply:*** Finde eine Musterinstanz und wende Sie auf den Arbeitsgraphen an.
- ***ApplyAll:*** Finde zuerst alle Musterinstanzen, wende diese anschließend auf den Arbeitsgraphen an.

Definition und Steuerung der Regelanwendung

Die Ersetzungsregeln können als normale Programmmethoden betrachtet werden. Sie können Graphen-Elemente als Parameter einnehmen und liefern bestimmte Daten zurück. Die Änderungen in einer Graphersetzungsregel werden im *modify*-Block beschrieben. Dieser beschreibt den Ersetzungsgraph oder genauer die Unterschiede von Muster- und Ersetzungsgraph. Dadurch können Knoten bzw. Kanten hinzugefügt, gelöscht oder umtypisiert werden. Das Umtypisieren kann aber auch quer zur Typhierarchie erfolgen [GrGen]. Hier wird auch der Rückgabeelement bzw. Nachricht deklariert. Knoten und Kanten sind im Objektmodell AOM allgemein als Graphelemente zu betrachten. Um deren Attribute vergleichen oder ändern zu können, erlaubt GrGen.NET die Verwendung von Bedingungen innerhalb der Ersetzungsregeln zu verwenden. Diese Bedingungen werden in einem *if*-Block deklariert. Darin befinden sich Boolesche Ausdrücke, wobei Attribute von Graphelementen als Variablen verwendet werden dürfen.

Das einfachste Suchmuster *FindImlicitTermination* findet alle Aktivitätsknoten, die keine ausgehende Kontrollflüsse haben und auf keine Objekte schreibend zugreifen können (siehe *negative*-Block). Dieses Suchmuster wird zu einem asynchronen Aufruf, wie etwa AutoFuter-Muster [AutoF] transformiert, indem einen Kontrollknoten hinzugefügt wird mit dem Attribut *asynchrone = true* (siehe *modify*-Block). Diese Transformationsregel liefert abschießend den Aktivitätsknoten zurück, der asynchron ausgeführt wird:

```
1:  rule FindImplicitTermination():(uActivityNode) {
2:     xNode: uActivityNode;
3:     sourceNode:uNode -cFlow:uControllFlow-> xNode;

4:     negative {
5:            xNode -:uControllFlow-> :uNode;
6:            xNode -:uObjectFlow-> :uObjectNode;
7:     }

8:     modify {
9:            delete(cFlow);
10:           asyncNode:uInvocationAction;
11:           asyncNode -start:uControllFlow-> xNode;
12:           sourceNode -cNewFlow:uControllFlow-> asyncNode;
13:           emit("ImplicitTermination found\n");
14:           return(xNode);
15:    }
16: }
```

Die Suchmuster können auch andere Suchmuster verwenden. So wird der *ObjectMerge*-Muster 4.4.1 nur dann ausgeführt, wenn es mehr als zwei Objektflüsse gibt, die in einen Aktivitätsknoten rein gehen. Um dies zu überprüfen wird folgende Hilfsmethode verwendet. Sie bekommt als Parameter einen Knoten vom Typ *uActivityNode* und liefert der Anzahl der eingehenden Objekte zurück:

```
1:  rule inputsCount(xNode:uActivityNode):(int)
2:  {
3:     iterated {
4:      :uObjectNode --> xNode;
5:      modify {
6:            eval {yield count = count+1; }
7:            }
8:     }

9:     modify {
10:           def var count:int = 0;
11:           return (count);
12:    }
13:  }
```

Alle Suchregeln sind über GrGen API zugänglich und können in Form von .NET-Methoden ausgeführt werden. Anwendung in .NET:

```
1:  IuActivityNode resultNode = new uActivityNode();
2:  Actions.FindImplicitTermination.Apply(Graph, ref resultNode);
```

Aus Platzspargründen werden hier nicht alle Suchregeln angezeigt. Das vollständige Regeldatei Suchschablonen.grg mit allen implementierten Suchmustern findet man auf der beigefügten CD.

5.5 Parallele Modelle

Ziel dieser Arbeit ist die bestehenden UML-Modelle zu parallelisieren. Dafür werden die vorliegenden Modelle eingelesen und ein Objektmodell gebaut sowie musterbasiert transformiert. Abschließend sollen neue UML-Modelle erzeugt werden. In diesen Modellen wird der Parallelität explizit dargestellt. Dafür haben verschiedene Diagrammtypen spezielle Konstrukte, wie zum Beispiel *Fork-Join*-Knoten in Aktivitätsdiagrammen oder kombinierte Fragmente *Par* und *Critical* in Sequenzdiagrammen.

Durch Musterersetzung mit GrGen.NET kommen oft neue Knoten dazu, die den Kontrollfluss in mehrere parallele Fäden splitten oder bestimmte Aktivitäten asynchron aufrufen. Diese neuen Elemente werden von den Ersetzungsregeln zurückgeliefert und im Programm gespeichert. Nach dem alle Muster gefunden und ersetzt werden kann man diese Daten abschießend nutzen um UML-Modelle zu modifizieren.

Das Werkzeug bietet zwei Möglichkeiten bestehende Modelle zu parallelisieren. Entweder werden sie direkt modifiziert oder es wird an einer Kopie gearbeitet. Im zweiten Fall können auch die Zwischenschritte gespeichert werden.

5.5.1 Änderungen in Klassendiagrammen

Parallele Prozesse oder Fäden werden in der UML als aktive Objekte (Instanzen der aktiven Klassen) modelliert 2.2.2. Wird der Kontrollfluss in mehrere parallele Fäden aufgesplittert, so wird pro Faden ein aktives Objekt nötig, das dieses Faden repräsentiert. Das kann zu der Entstehung von Wettlaufsituationen führen, wenn mehrere Objekte auf der selber Ressourcen zugreifen. Diese Überprüfung findet bereits in dem Objektmodell statt und falls globale Ressourcen gemeinsam genutzt werden, wird die Parallelisierung nicht möglich. An diese Stelle ist auch eine Diskussion möglich, ob man diese Wettlaufsituationen durch einsetzen von Synchronisierungsmechanismen vermeiden kann? Die UML bietet einige Elemente dafür, wie zum Beispiel das *Critical*-Sektion in Sequenzdiagrammen oder die Pufferknoten in Aktivitätsdiagrammen, die mehrere Objekte in sich speichern können. Die Änderungen in Klassendiagrammen sind von der UML Version und Stereotypen abhängig. Die Methoden von aktiven Objekten können in Microsoft Visual Studio durch das Eigenschaft *Concurency* ihr Verhalten bei der Nebenläufigkeit bestimmen. Es stehen folgenden Möglichkeiten zur Verfügung:

- **Sequential** – Der Methode ist nicht threadsicher entworfen. Das gleichzeitige Aufrufen kann zu Fehlern führen.

- **Guarded** – Die Methode wird solange automatisch blockiert, bis andere Instanzen des Objekts abgeschlossen wurden.

- **Concurrent** – Der Datentyp wurde so entworfen, dass mehrere Aufrufe der Methode gleichzeitig ausgeführt werden können.

Diese Eigenschaften können zusätzlich gesetzt werden, um die Methoden entsprechen zu markieren. Ansonsten werden die Klassendiagramme kaum verändert.

5.5.2 Parallele Aktivitätsdiagramme

Die Änderungen in Aktivitätsdiagrammen werden hauptsächlich durch einfügen von neuen Parallelisierungs- (*Fork*) und Zusammenführungsknoten (*Join*), sowie *Call-Recive*-Knoten für asynchronen Vorgengen. Die Zusammenführungsknoten sind gleichzeitig die Synchronisationspunkte. Dies bedeutet, dass sie die Ausführung erst dann über sich durchlassen, wenn alle Eingabeflüsse eingetroffen sind (siehe 4.5.1). Alle eingehenden Datenflüsse werden an entsprechenden Input-Pins weitergeleitet. Die Änderungen an

Aktivitätsdiagrammen werden bei allen Suchmustern nötig. Folgendes Beispiel zeigt ein sequenzielles und paralleles Aktivitätsdiagramm mit Parallelisierungs- (*Fork*) und Zusammenführungsknoten (*Join*). Der Anzahl von parallel ablaufenden Aktionen kann sich in verschiedenen Suchmustern unterscheiden. So ist er in dem *ObjectMerge*-Muster nur durch den Anzahl der unabhängigen Aktionen, die Objektknoten erzeugen begrenzt. Das einführende Beispiel (siehe 2.1) kann letztendlich durch den MAP-Ansatz folgendermaßen parallelisiert werden:

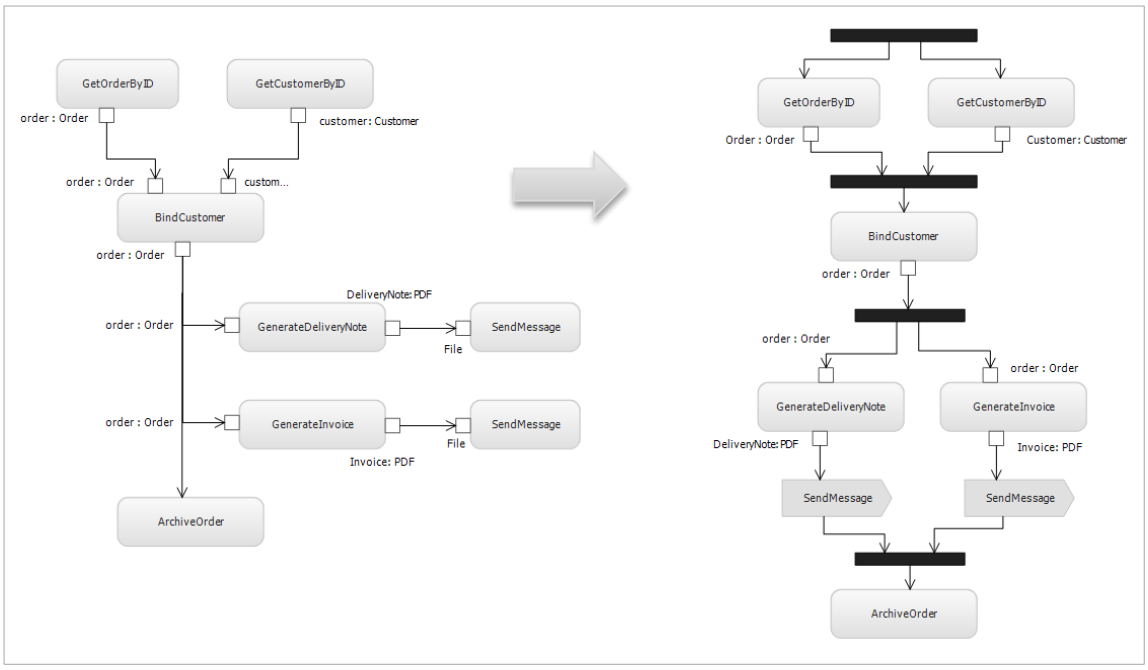


Abbildung 92: Parallelisierung von Aktivitätsdiagrammen

Hardware Parallelitätsgrad

Viele wissenschaftliche Arbeiten [Pankr11], [MS+12] belegen, dass es eine Abhängigkeit zwischen dem Anzahl der Prozessoren und dem Anzahl der parallelen Fäden im Programm gibt. Die Beschleunigung ***S*** ist nach oben begrenzt (Amdahlsches Gesetz): ab einer bestimmten Zahl der Fäden steigen die Kommunikation und Synchronisationskosten exponentiell und überwiegen die Vorteile der Parallelisierung. In dieser Arbeit wird ein spezieller Parameter eingeführt, der Parallelisierungsgrad von oben begrenzt - **Hardware Parallelitätsgrad** *(MaxParIndex)*. Wenn bei der Erzeugung von parallelen Aktivitätsdiagrammen mehr Aktionen gibt, als maximale Parallelitätsgrad erlaubt, werden sie in mehreren parallelen Sequenzen gruppiert. Dafür wird das *Number*-Attribut, der die zeitliche Ausführungsreihenfolge angibt verwendet: die Aktionen mit ähnlichen *Number*-Wert werden in unterschiedlichen Sequenzen ausgeführt, falls der Kontrollfluss dies erlaubt. Die Evaluierung hat auch gezeigt, dass eine Aktion auch durch den Mangel von anderen Betriebsmitteln (Arbeitsspeicher, Fetzchen von der Festplatte, etc) gebremst werden kann (siehe 6.1).

5.5.3 Parallele Sequenzdiagramme

In einem Sequenzdiagramm werden die aktiven Objekte optional mit Namen und Typ angegeben. Der Typ eines Objekts kann eine Klasse mit mehreren Unterklassen oder ein Interface sein [UML2.3]. Entsprechend kann das im ablaufenden System beobachtete Objekt aus einer anderen Klasse sein. In diesem Sinn berücksichtigen Sequenzdiagramme also die Vererbung aus Klassendiagrammen. Eine Meldung spiegelt entweder eine Operation-Anruf oder Senden und Empfangen eines Signals. Wenn eine Nachricht ein Signal darstellt, sind die Argumente der Nachricht die Attribute des Signals.

Für die Darstellung von parallelen Abläufen in Sequenzdiagrammen wird der **Par-Operator** verwenden (siehe Abbildung 93). Das parallele Fragment **Par** ist analog zu dem Splitting (*ForkNode*) in den Aktivitätsdiagrammen. Einer expliziten Synchronisation (*JoinNode*) gibt es hier aber nicht. Laut [UML2.3] ist die Synchronisation implizit und das par-Fragment endet, wenn alle parallelen Teile zu Ende sind. Eine Besonderheit der Sequenzdiagramme ist das **Kritisches Bereich** (engl. *critical region*). Dieses Bereich bezeichnet die kritischen Abschnitte, wo alle Aktionen ohne jegliche Unterbrechung sequenziell ausgeführt werden. Dieses Konstrukt wird in den Mustern verwendet, in denen mehrere parallel ausgeführte Aktionen auf einen Objekt zugreifen. Dadurch wird die Sperre für einen bestimmtes Objekt ermittelt, eine Aktion ausgeführt und die Sperre wieder aufgehoben wird. Kritischer Bereich ist analog zu den ***lock***-Anweisung in C# und Java.

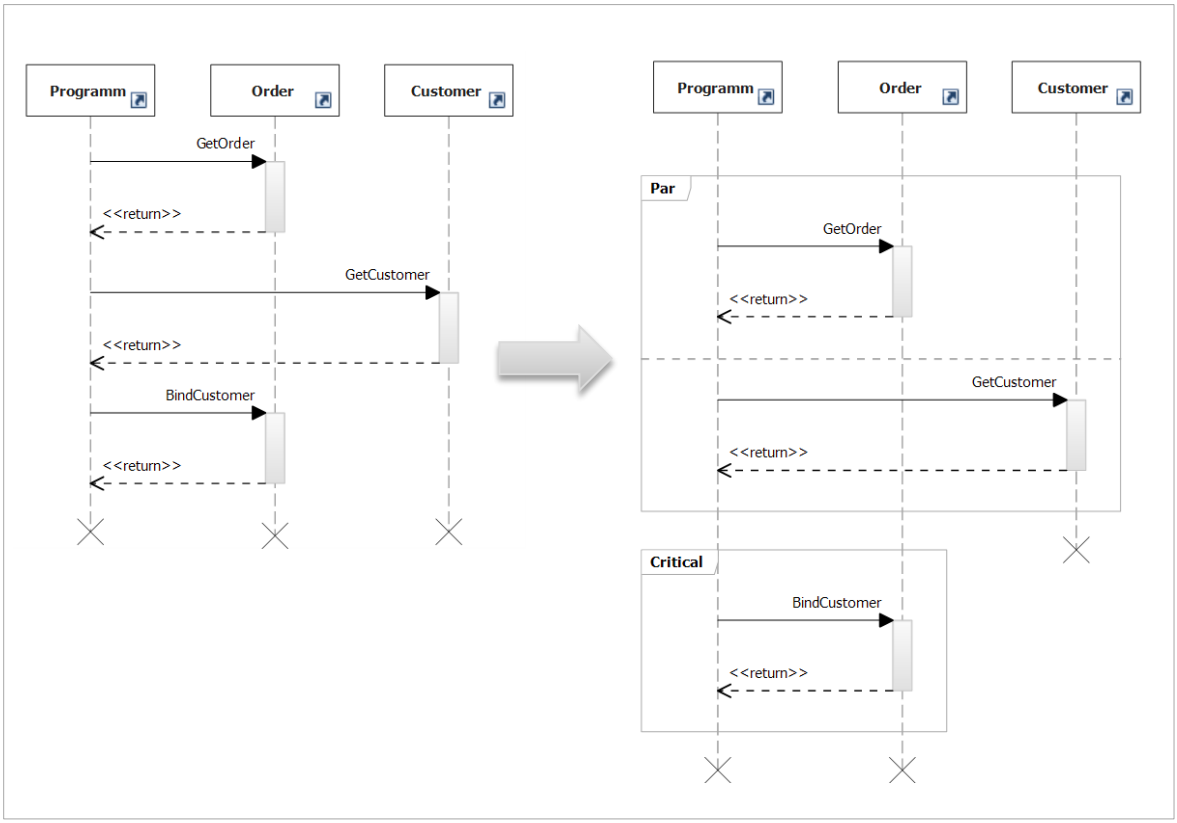


Abbildung 93: Parallelisierung von Sequenzdiagrammen

5.6 Zusammenfassung

In diesem Kapitel wurde eine prototypische Implementierung des MAP-Ansatzes beschrieben, welches sich auf dem Suchmusterkatalog aus Kapitel 4 basiert. Dazu wurden vier Phasen definiert: das Einlesen und Parsen den UML-Modellen mit Hilfe des UML API der Firma Microsoft. Aufbau des Objektmodells und die Modellmodifikation mit Hilfe des Suchmusters. Das Objektmodell, sowie das Musterkatalog wurden durch das Graphenersetzungssystem GrGen.NET implementiert. Das Meta-Modell und die Suchmuster werden in Form von .NET-Assemblies kompiliert und im Projekt verwendet. Abschließend wurden aus dem modifizierten Objektmodell neue parallele UML-Modelle erzeugt. Dafür wurden die gegebenen Modelle kopiert und an gewissen Stellen parallelisiert.

In diesem Kapitel wurden außerdem die Hilfsanwendungen GrShell und yComp vorgestellt, die für die Visualisierung des Objektmodells verwendet wurden. Mit deren Hilfe könnten einzelne Parallelisierungsschritte in Form von Graphen gespeichert und analysiert werden. Dies hat die Fehlersuche in dieser Arbeit wesentlich erleichtert.

6 EVALUIEREUNG

In diesem Kapitel werden die Ergebnisse der Implementierung beschrieben und durch Anwendung an verschiedenen Beispielen evaluiert. Für die Evaluierung wurden kleine wissenschaftliche Anwendungen, als auch zwei echten Wirtschaftsanwendungen verwendet. Zu den wissenschaftlichen Anwendungen gehörten: die Desktopsuche 6.1, die am Karlsruhe Institut für Technologie (KIT), Lehrstuhl Prof. Dr. Walter F. Tichy entwickelt wurde. Sowie Implementierung eines Sudoku-Spiels 6.2 aus Microsoft Samples Library [MSDN11a] für parallele Programmierung. Für diese Anwendungen existiert auch eine manuell parallelisierte Version. So können die Evaluierungsergebnisse zusätzlich mit manuell erzeugten Parallelisierungen verglichen werden.

Die zwei relativ großen Softwaresysteme aus der Wirtschaft werden durch ein webbasierte Lagerverwaltungssystem 6.3 und ein auf dynamischen Regeln basierendes System für die Prüfung, Berechnung und Auswertung von Finanzdaten von der Firma EXXETA AG 6.4 vertreten. In diesen Softwaresystemen wurden einige Module evaluiert. Dafür wurden aus dem Code die Sequenz- und Klassendiagrammen automatisch generiert und durch einen Aktivitätsdiagramm vervollständigt.

Dieses Kapitel ist folgendermaßen strukturiert: zuerst wird jede Anwendung kurz vorgestellt und durch eine Menge der Codemetrikwerte beschrieben. Aus jeder Anwendung werden einige Teile in Form von UML Diagrammen durch den MAP-Ansatz parallelisiert. Zur Einschätzung des Ansatzes werden die Trefferquote und Genauigkeit der einzelnen Suchmuster gemeinsam betrachtet. Die Ergebnisse werden in Form einer Konfusionsmatrix vorgestellt. Folgendes Beispiel zeigt eine solche Konfusionsmatrix:

	Relevant	**Nicht relevant**
Gefunden	2	1
Nicht gefunden	1	0

Tabelle 5: Konfusionsmatrix mit Evaluierungsergebnissen

Daraus können die Trefferquote und Genauigkeit der Suchmuster leicht abgeleitet werden: Genauigkeit: $^{8}/_{(8+4)} = {}^{8}/_{12} = {}^{2}/_{3} \approx 0{,}67$. Trefferquote: $^{8}/_{(8+12)} = {}^{8}/_{20} = {}^{2}/_{5} = 0{,}4$

Definition der Codemetrikwerte

In diesem Kapitel werden einige Codemetrikwerte verwendet, um ein Verständnis zu vermitteln, welche Module und Klassen getestet und evaluiert werden. Die folgende Werde wurden mit Hilfe des Microsoft Visual Studio berechnet:

- **LOC** (*Lines of Code*): Gibt die Anzahl von Zeilen im Code an.
- **NOC** (*Number of Classes*): Anzahl der Klassen.
- **NOM** (*Number of methods*): Anzahl der Methoden.

Außerdem wurden folgende Komplexitätsmetriken ermittelt, die eine relative Komplexität und Qualität des Moduls bestimmen:

- **CYCLO** (*Cyclomatic Complexity*): zyklomatische Komplexität – zeigt die strukturelle Komplexität des Codes an. Sie wird durch Berechnung der Anzahl unterschiedlicher Codepfade im Programmfluss erstellt.
- **MI** (*Maintainability Index*): Von Microsoft vorgeschlagener Wartbarkeitswert, der angibt, wie einfach der Code zu verwalten ist. Es wird ein Indexwert zwischen 0 und 100 berechnet. Ein hoher Wert steht für bessere Wartbarkeit.

6.1 Desktopsuche

Desktopsuche ist eine im Institut für Programmstrukturen und Datenorganisation (IPD) entwickelte Anwendung. Sie analysiert innerhalb eines eingegebenen Verzeichnisses auf der Festplatte alle im Text-, XML oder HTML-Format vorliegenden Dateien und zählt die Anzahl von unterschiedlichen Wörtern, sowie die Anzahl der Vorkommen eines Wortes auf. Die Anwendung liegt sowohl in sequentieller als auch in einer parallelen Version vor, die von David Meder [Meder10] entwickelt wurde. Folgende Abbildung zeigt die drei wichtigsten Klassen dieser Anwendung:

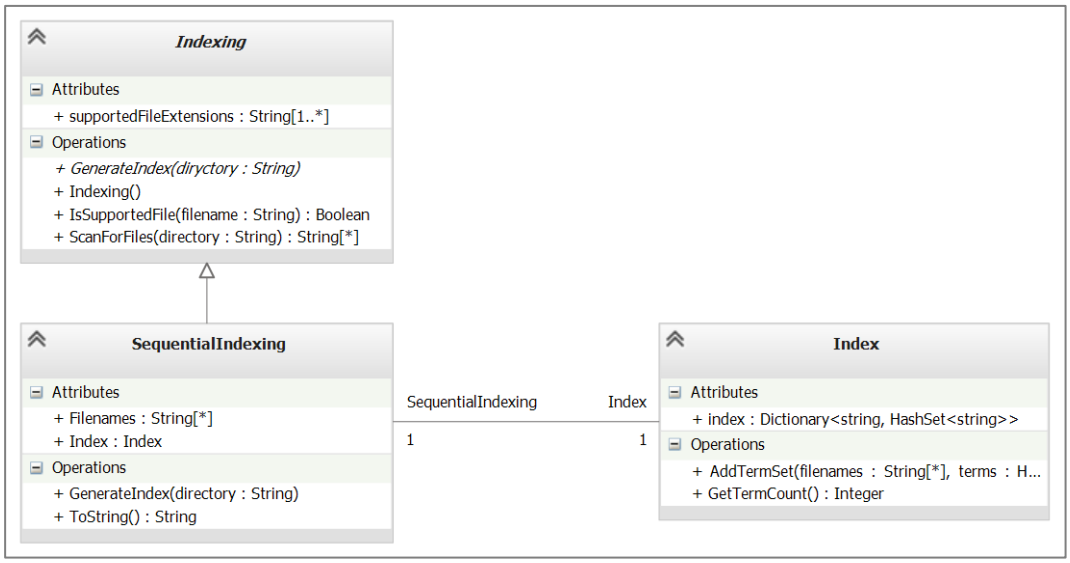


Abbildung 94: Klassendiagramm für Desktopsuche

Die abstrakte Klasse *Indexing* definiert die wichtigsten Programmmethoden: *ScanForFiles* und *GenerateIndex*, die in der Kindklasse *SequentialIndexing* implementiert wurden. Eine weitere Klasse *Index* dient einerseits zur Speicherung der Suchergebnisse (Feld *index*), andererseits ist die Klasse aktiv und führt die Methode *AddTermSet* aus. Der Suchalgorithmus wird durch folgendes Aktivitätsdiagramm abgebildet:

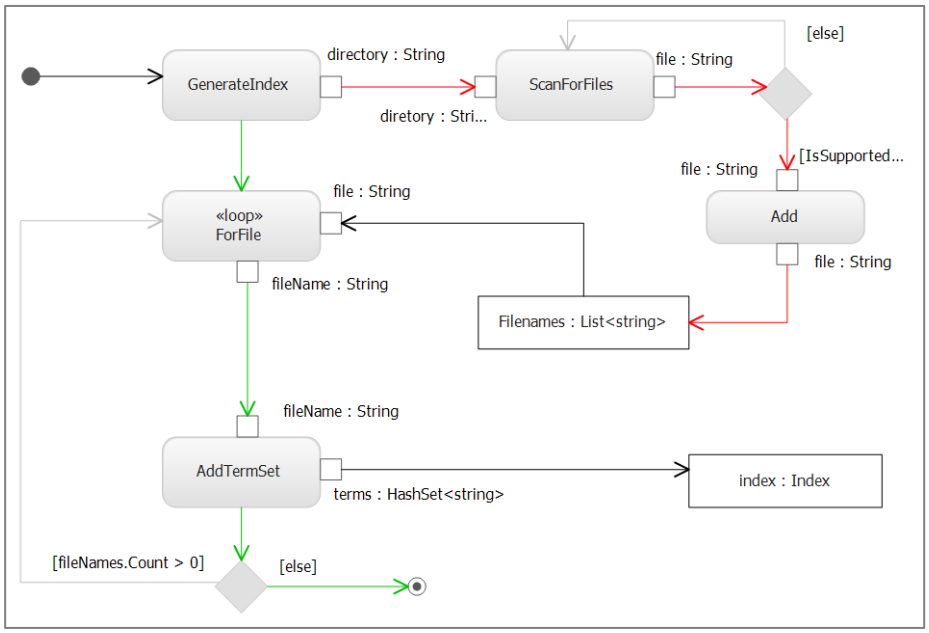


Abbildung 95: Aktivitätsdiagramm für Desktopsuche

Im obigen Aktivitätsdiagramm (Abbildung 95) wurden zwei Hauptkontrollpfade im Programm farbig markiert. Der rote Pfad beschreibt den Aufbau der Liste von Dateien, die analysiert werden müssen. Der grüne Pfad beschreibt den Aufbau eines Suchindex, der für die Suche verwendet wird (die Klasse *Index* im Klassendiagramm oben). Laut automatisch generiertem Sequenzdiagramm werden diese Pfade nacheinander ausgeführt. Das ist auch logisch, denn die Methode *GenerateIndex* (grüne Pfad) verwendet die Liste mit Dateinamen (*Filenames*). Es ist eine Read-After-Write Datenabhängigkeit [2.3.2]. Beide Pfade zu parallelisieren wäre also eine falsche Entscheidung. Beide Pfade beinhalten auch die impliziten und expliziten Schleifen. Die expliziten Schleifen werden als Aktionen vom Stereotyp «*loop*» dargestellt. Diese Schleifen können theoretisch parallelisiert werden.

Folgende Abbildung zeigt die Visualisierung des Arbeitsgraphen vor der Transformation. Beide Objektknoten *Filenames* und *Index*, sowie die Kontrollknoten sind hier dunkel grau markiert.

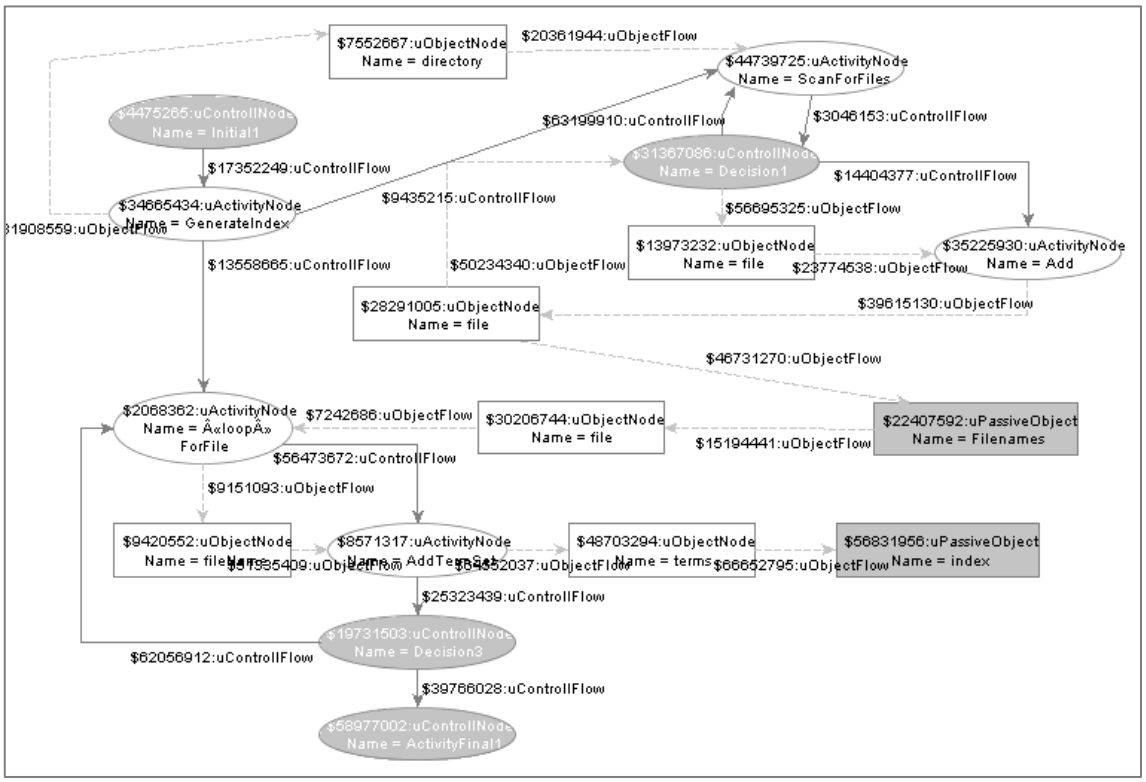


Abbildung 96: Objektmodell für Desktopsuche

Nach Angaben von MAP wurden hier **zwei Producer-Consumer-Suchmuster** mit Parallelisierungspotenzial gefunden. Das erste Muster verwendet Filenames-Objektknoten als Puffer. Die Aktion *Add* fügt neue Objekte hinzu, welche in der Schleife konsumiert werden. Das andere Muster befüllt den Index. Hier gibt es keinen Verbraucher. In diesem Fall kann die Erzeuger-Aktion theoretisch beliebig oft repliziert werden. Allerdings dient der oben beschriebene Parameter *MaxParIndex*, der den Hardware Parallelitätsgrad angibt [5.5.2], als die Obergrenze für der Anzahl der Replikationen. Implementiert wurde dieses Muster mit Hilfe einer parallelen *For*-Schleife.

Abschließend wurde diese Anwendung nach Angeben des MAPs manuell parallelisiert. Im Folgenden werden die Ergebnisse mit den vorliegenden sequenziellen und parallelen Versionen verglichen. Für die Evaluierung wurden 150 Textdateien von insgesamt 22,5 MB auf drei Rechnerkonfigurationen jeweils 20 Mal getestet. Abschließend wurde ein Mittelwert gebildet.

1. Intel DualCore 1,8GHz, 2GB RAM, SSD Festplatte
2. Intel Core2Quad Q6600, 2.4 GHz, 8GB RAM. HDD Festplatte
3. AMD 8-KernRechner, 3.2 GHz, 16GB RAM

Die Ergebnisse:
Diese Tabelle stellt die Ergebnisse der Evaluierung dar. Für die jeweilige Programm-Version wurden die Zeit in Millisekunden und die Beschleunigung im Vergleich zur sequenziellen Version gemessen.

Hardware	Version	Zeit, ms	Speedup
2-Kerne, 1,8 GHz 2 GB RAM, SSD	Seq	2616	100%
	Par	2470	106%
	MAP	1711	153%
4-Kerne, 2,4 GHz, 8 GB RAM, HDD	Seq	2215	100%
	Par	1164	190%
	MAP	1052	211%
8-Kerne, 3,2 GHz 8 GB RAM, HDD	Seq	1880	100%
	Par	685	274%
	MAP	811	232%

Tabelle 6: Desktopsuche Evaluierung

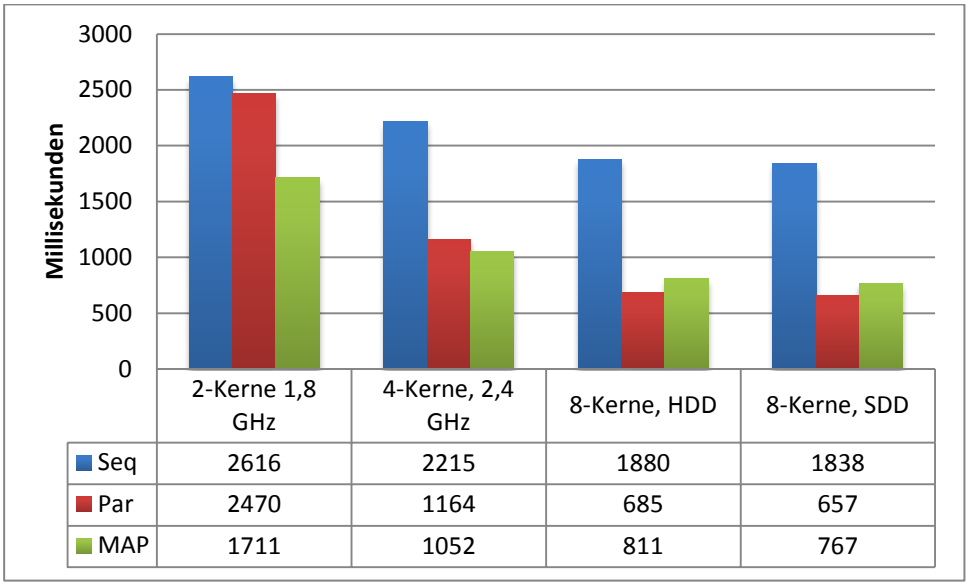


Abbildung 97: Testergebnisse in Millisekunden

Wie aus den Testergebnissen ersichtlich wird, ist die Parallelisierung mit Hilfe des MAP-Werkzeuges nicht nur einfacher, sondern auch wesentlich schneller als die vorliegende parallele Version. Ein gefundenes Muster kann man als nicht relevant einstufen, da die Parallelisierung dort kaum Beschleunigung bringt. Das zweite Muster zeigt dagegen großes Potenzial, wie die Messdaten zeigen.

	Relevant	Nicht relevant
Gefunden	1	1
Nicht gefunden	0	0

Tabelle 7: Konfusionsmatrix für Desktopsuche

6.2 Sudoku

Sudoku ist ein japanisches Logikspiel. In der üblichen Version ist es das Ziel, ein 9×9-Gitter mit den Ziffern 1 bis 9 so zu füllen, dass jede Ziffer in jeder Spalte, in jeder Zeile und in jedem Block (3×3-Unterquadrat) genau einmal vorkommt. Für die Evaluierung dieser Arbeit wurde eine Sudoku-Implementierung aus Microsoft Samples Library [MSDN11a] für parallele Programmierung mit .NET analysiert. Diese Programme liegen sowohl in der sequenziellen, als auch in der manuell parallelisierten Variante vor. Somit können die durch MAP erkannten Suchmuster und Parallelisierungsvorschläge mit den tatsächlichen manuellen Parallelisierungen verglichen werden.

Der Anfangszustand des Spiels wird durch die Methode *Generate* von der aktiven Klasse *Generator* erzeugt. Diese Methode bekommt einen Parameter vom Typ *GeneratorOptions*. Unteranderem kann dieser Parameter den Schwierigkeitsgrad und ob die Generation des Spielfeldes parallel ablaufen soll bestimmen. Abbildung 98 zeigt diesen beiden Klassen und zwei weitere Klassen, die für die Analyse verwendet werden (*Solver* und *SolverResults*):

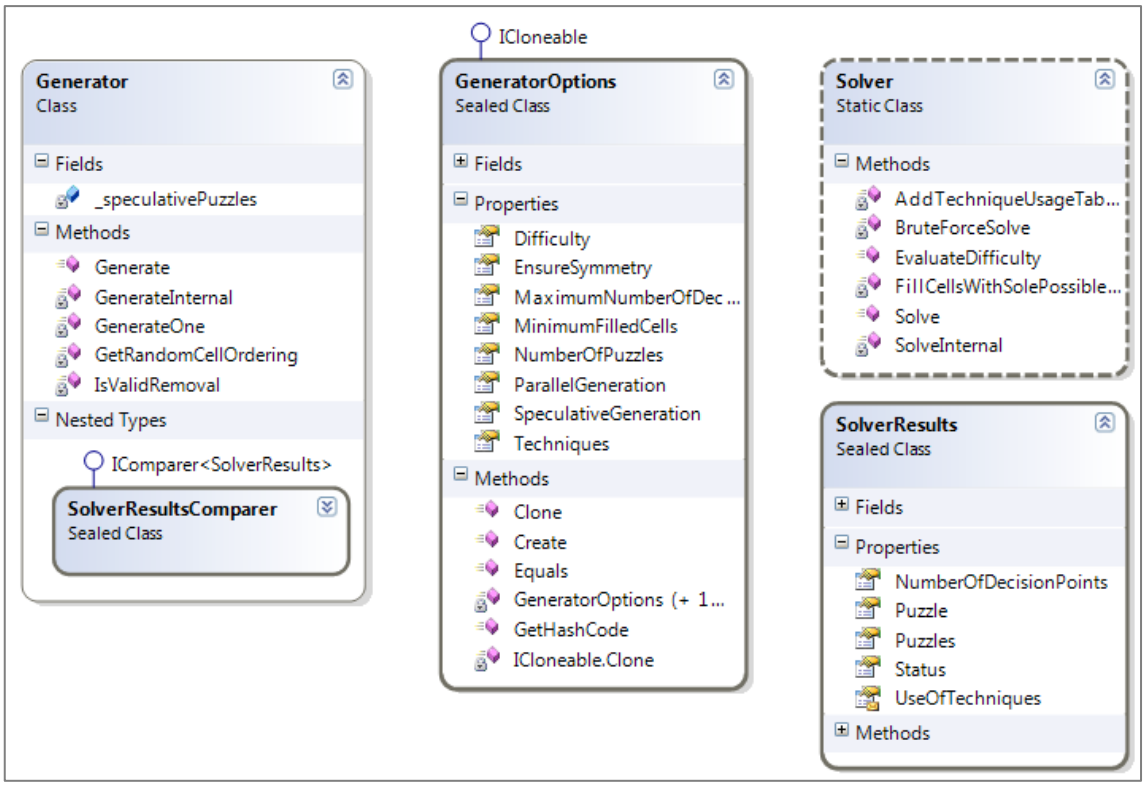


Abbildung 98: Klassendiagramm für Sudoku-Generator

Auch hier wurden Klassen und Sequenzdiagramme automatisch aus dem vorhandenen Code generiert und das Aktivitätsdiagramm manuell erstellt. Für dieses Beispiel wurden jeweils ein Sequenz- und ein Aktivitätsdiagramm gemacht, die Erstellung eines Spielfeldes beschreiben. In diesem relativ kleinen Beispiel wurden mehreren Suchmuster gefunden:

	Relevant	**Nicht relevant**
Gefunden	2	2
Nicht gefunden	0	0

Tabelle 8: Konfusionsmatrix für Sudoku

Insgesamt wurden vier Suchmuster gefunden: zwei *ObjectMerge* und zwei *ParallelLoop*. Die *ObjectMerge*-Suchmuster (siehe Abbildung 99) werden als nicht relevant eingestuft, da sie kaum Performance durch Parallelisierung bringen. Die zwei parallelen Schleifen haben dagegen sehr großes Potenzial – innerhalb dieser Schleifen wurde ein *Solver* (Methode *Solve*) für alle möglichen Kombinationen berechnet und die Laufzeit dieser Methode beträgt laut Visual Studio Performance Wizard 23% der Gesamtlaufzeit des Programms.

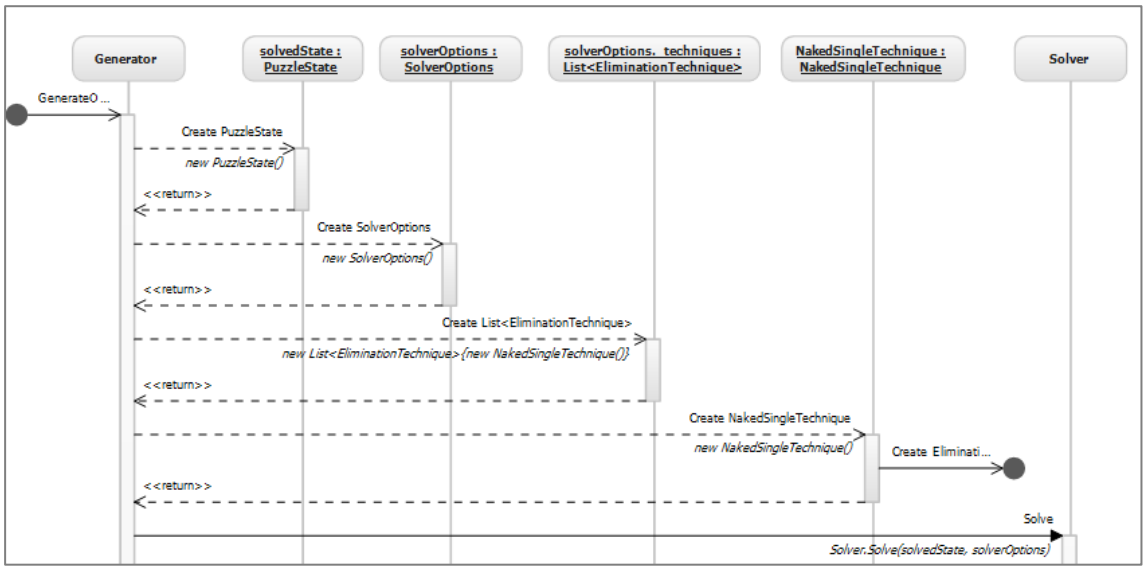


Abbildung 99: ObjectMerge-Suchmuster im Sudoku-Beispiel

Die manuelle Parallelisierung unterscheidet sich nur leicht von den von MAP vorgeschlagenen Varianten. Hier wurde anstatt einer Schleife eine parallele Version von LINQ verwendet, um die Methode *GenerateOne* mehrmals auszuführen. Das führ aber zu gleichen Performance-ergebnissen durch Parallelisierung, wie beim MAP.

6.3 Lager ERP System

Das Lager ERP System sorgt für eine bessere Transparenz und einen schnelleren Materialfluss in Lagerhallen. Das Lagerverwaltungssystem kann beliebig viele Lager und Stellplätze flexibel steuern. Es können mehrere unterschiedliche Artikel von unterschiedlichen Lieferanten auf einem Stellplatz verwaltet werden. Durch gezielte, schnelle und einfache Zugriffe kann der Anwender auf die Bestände jedes einzelnen Artikels zugreifen. Dabei können verschiedene Ansichten gewählt werden - alle Artikel auf einem Stellplatz, alle Artikel von einem Lieferant, u.s.w.

Die Anwendung ist nach der 3-Schichten-Architektur (MVC) aufgebaut und hat in der Businesslogik zehn *Controller*-Klassen und 48 *Model*-Klassen, die Datenentitäten im Programm repräsentieren. Bei der Evaluierung wurde die Darstellungsschicht (*View*) nicht betrachtet, da die dort vorhandenen Dateien keine Logik in sich hatten (hauptsächlich nur HTML-Code).

Codemetrikwerte

Die folgende Tabelle enthält Codemetriken für einzelne Projektteile:

Modul	LOC	NOC	NOM	CYCLO	MI
LagerERP	132	1	5	6	81
Controllers	783	10	120	276	76
Helper	120	2	4	5	81
Models	1925	48	158	879	83
View	*5263**	*10**		183	86
Insgesamt**	**2960**	**61**	**287**	**1168**	**82**

Tabelle 9: Codemetrikwerte für das Lager ERP System

** HTML + C# Code zusammen. Keine Klassen, sondern ASPX-Seiten für jeweiliger Controller.*

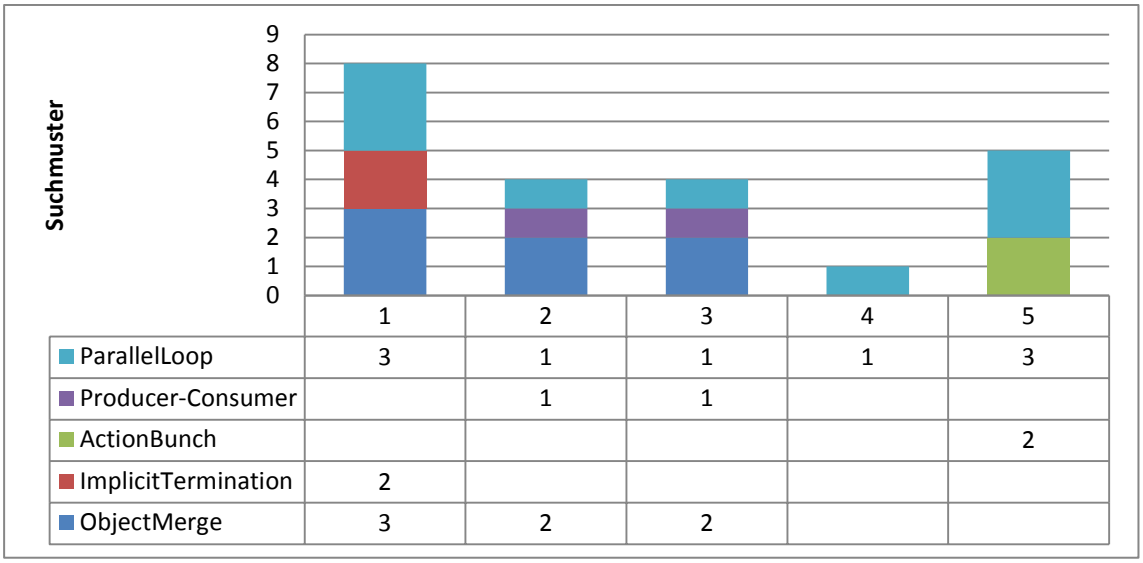
Evaluierung

Für die Evaluierung wurden insgesamt 10 Aktivitäts- und entsprechenden Sequenzdiagrammen verwendet, die die größten Abläufe im Programm beschreiben. Alle Klassen wurden in einem Klassendiagramm beschrieben. Alle Sequenzdiagramme wurden aus dem Code mit Hilfe des Visual Studio automatisch erzeugt. Die Aktivitätsdiagramme sind dagegen vor der Entwicklung in der Projektplanungsphase entstanden und für die jetzige Evaluierung leicht angepasst, damit sie Konsistent und Widerspruchsfrei zu den anderen Modellen sind. Alle diese Modelle findet man im Anhang E und auf beigefügter CD.

Modelle	Klassen	Gefundene Suchmuster
Kundensuche, Tiefe Suche	ISearch, CustomerSearch, OrderSearch, ItemsSearch, SearchResult	3 x ParallelLoop 3 x ObjectMerge 2 x ImplicitTermination
Kundenübersicht, Kundendetails	Customer, CustomerDetails, CustomerOrder	2 x ObjectMerge 1 x ParallelLoop 1 x Producer-Consumer
Lieferantenübersicht, Lieferantendetails	Supplier, SupplierDetails, SupplierItems	2 x ObjectMerge 1 x ParallelLoop 1 x Producer-Consumer
Bestellungen Übersicht	Order, Item	1 x ParallelLoop
Statistiken, Rechnungen	Report, DetailReport, Customer, Order, Item	3 x ParallelLoop 2 x ActionBunch

Tabelle 10: Evaluierung des ERP Systems

Insgesamt wurden 23 verschiedene Suchmuster in 10 Fällen (Modellen) gefunden. Die Evaluierung hat gezeigt, dass die am meisten vorkommenden Suchmuster *ParallelLoop* (9 Mal) und *ObjectMerge* (7 Mal) sind.



	1	2	3	4	5
ParallelLoop	3	1	1	1	3
Producer-Consumer		1	1		
ActionBunch					2
ImplicitTermination	2				
ObjectMerge	3	2	2		

Abbildung 100: Verteilung von Suchmuster

Die Klassen *Customer* und *Supplier*, sind vom Aufbau her sehr ähnlich und besitzen ähnliche Methoden. Die manuelle Analyse hat gezeigt, dass in diesen beiden Klassen und in der Klasse *Search* noch weitere Parallelisierungsmöglichkeiten bestehen, die durch MAP nicht erkannt wurden. So gibt es weitere Schleifen mit Parallelisierungspotenzial, die aber auf globale Variablen zugreifen. Bei der Implementierung wurden diese Zugriffe mit einer *lock*-Anweisung geschützt und die Schleifen wurden parallelisiert. Das hat eine Beschleunigung ergeben trotz der Sperranweisung, da diese Zugriffe nur selten vorkamen (innerhalb eines *if*-Blocks). Um solche Fälle zu erkennen, kann das MAP in Zukunft mit dynamischen Analysemethoden kombiniert werden.

Die zwei in dem Suchmodul gefundenen *ImplicitTermination*-Suchmuster sind als falschpositiv zu bewerten. In diesen Fällen bricht die Suche einfach ab, falls das gesuchte Objekt gefunden wird. Hier ergibt die asynchrone Ausführung keinen Sinn. Insgesamt wurde dieses Beispiel mit folgenden Ergebnissen evaluiert:

	Relevant	**Nicht relevant**
Gefunden	20	3
Nicht gefunden	4	0

Tabelle 11: Konfusionsmatrix für ERP Lager-System

6.4 System für die Berechnung von Finanzdaten

Das zweite Beispiel von der Firma EXXETA AG ist ein System für die Prüfung, Berechnung und Auswertung von Finanzdaten, das auf dynamischen Regeln basiert. Die Verarbeitung der Daten wird durch hochleistungsfähige Massendatenverarbeitungsstrategien durchgeführt. In diesem Beispiel wurden mehreren komplexe Berechnungen im Programm durchgeführt. Dadurch entstehen viele Kontroll- und Datenflüsse. Das macht das Beispiel besonders interessant für die Evaluierung des MAP-Ansatzes.

Codemetrikwerte

Evaluiert wurden drei .NET-Projekte aus der Solution, die für die regelbasierten Berechnungen verantwortlich waren. Die folgende Tabelle enthält Codemetriken für einzelne Projektteile:

Modul	LOC	NOC	NOM	CYCLO	MI
DataLogic	122	1	4	8	71
DataMapping	464	3	34	129	74
Calculations	699	12	89	334	66
Insgesamt	**1285**	**16**	**127**	**471**	**82**

Tabelle 12: Codemetrikwerte für das Fin. System

Evaluierung

Für die Evaluierung wurden 5 Fälle genommen und entsprechende Sequenz- und Aktivitätsdiagrammen erstellt. Wie im vorherigen Beispiel wurden alle Klassen in einem Klassendiagramm beschrieben. Alle Sequenzdiagramme wurden ebenfalls aus dem Code automatisch erzeugt.

Klasse	Gefundene Suchmuster
UpgradeScope	2 x ImplicitTermination 1 x ParallelLoop
CancellScope	1 x ImplicitTermination 2 x ParallelLoop
DowngradeScope	1 x ImplicitTermination 1 x ParallelLoop 1 x ActionBunch
RefundScope	2 x ObjectMerge
RenewalScope	1 x ImplicitTermination

Tabelle 13: Evaluierungsergebnisse für das Fin. System

In diesem Beispiel wurden zahlreiche *ImplicitTermination*-Muster gefunden. Das kann dadurch erklärt werden, dass die Berechnungen abschließend in die Datenbank gespeichert werden. Der Aufruf des Datenbankobjektes durch eine *void*-Methode wurde dabei als *ImplicitTermination* erkannt. Einige von diesen Aufrufen, die für einzelne Objekte gemacht wurden, könnten tatsächlich parallelisiert werden. Die anderen dagegen nicht, weil sonst Inkonsistenzen in der Datenbank entstehen könnten. Insgesamt wurden 12 Suchmuster gefunden. Vier wurden hier als nicht relevant markiert. Noch Zwei Stellen mit Parallelisierungspotenzial wurden durch das MAP nicht erkannt.

	Relevant	Nicht relevant
Gefunden	8	4
Nicht gefunden	2	0

Tabelle 14: Konfusionsmatrix für das Fin. System

7 ZUSAMMENFASSUNG UND AUSBLICK

Im Rahmen dieser Arbeit wurde ein Ansatz für die musterbasierte Modelltransformation zur Erzeugung von parallelen Softwaremodellen entwickelt und prototypisch implementiert. Das Werkzeugt liest die bestehende UML Diagramme ein und erzeugt daraus ein kombiniertes Graphen-basiertes Objektmodell AOM. Dieses Modell wird mit Hilfe eines Graphersetzungssystem GrGen.NET modifiziert. Dafür wird in Rahmen dieser Arbeit ein Musterkatalog entwickelt, in dem die Suchmuster definiert sind, die implizite Parallelität beinhalten und Transformationsregeln um dieses Parallelität explizit darzustellen. Zum Schluss werden aus dem modifizierten Objektmodell neue UML Diagramme generiert, die explizite parallele Konstrukte beinhalten.

Der vorgeschlagene Ansatz, die UML Diagramme in kombiniertes Objekt-Modell zu übersetzen und das dann zu transformieren, bietet, wie im letzten Kapitel gezeigt wurde, einige Vorteile:

- Unabhängigkeit von der Version und Implementierungsdetails der vorliegenden UML-Modellen
- Flexibilität durch Erweiterbarkeit des Musterkatalogs und zugrundeliegendes Meta-Modells
- Unterschiedliche Ausgabemöglichkeiten: es können nicht nur neue Modelle, sondern auch Kommentare für den Entwickler und durch weitere Erweiterungen sogar paralleler Programmcode generiert werden

Der Ansatz wurde mit Hilfe von mehreren Anwendungen unter anderen zwei echten Betriebsanwendungen erfolgreich evaluiert. Es wurde gezeigt, dass die vorgeschlagenen Suchmuster in den vorliegenden Beispielen wirklich vorkommen und somit eine automatische Parallelisierung auf Modellebene möglich machen. Ein Beispiel wurde mit Hilfe von vorgeschlagenen Modellen manuell Parallelisiert und es wurde eine Verbesserung der Laufzeit festgestellt. Die Untersuchungen haben auch gezeigt, dass der Mehraufwand bei der Verwendung des Werkzeugs minimal ist.

7.1 Zukünftige Arbeiten

Die in dieser Arbeit gewonnenen Erkenntnisse im Bereich Analyse der UML-Modelle, sowie bei der Extrahierung des Kontroll- und Datenflusses, können als Grundlage für weitere Forschungsarbeiten dienen. Der Musterkatalog kann durch neue Muster erweitert und spezifiziert werden. Generell spricht nichts dagegen, den vorgeschlagenen Ansatz für weitere Zwecke, wie zum Beispiel zur automatischen Verifikation von Modellen zu nutzen.

Das AOM wird aus drei Typen von UML-Modellen gebaut. In zukünftigen Arbeiten kann auch untersucht werden, ob weitere Diagrammentypen in Frage kommen. Zum Beispiel die UML-Kommunikationsdiagramme [RQZ07]. Diese stellen eine Teilmenge der Informationen eines Sequenzdiagrams dar, fokussieren aber weniger auf die zeitliche Reihenfolge, sondern mehr auf die Zusammenarbeit zwischen den Objekten.

Die Evaluierung hat gezeigt, dass die Suchmuster manchmal als falschpositiv bewertet werden. Das hat zwei Gründe:

1. **Es können nur die Daten verwendet werden, die in vorliegenden Modellen vorkommen**. So können die Zugriffe auf globalen Variablen aus Methoden nur dann erkannt werden, wenn diese in Aktivitätsdiagrammen explizit dargestellt sind. Eine mögliche Lösung dafür wäre eine Erweiterung von MAP durch eine statische Codeanalyse, die solchen Zugriffe findet kann und nach AOM abbildet. Dadurch werden im AOM neue Objektknoten und Kanten entstehen, die diese Zugriffe

repräsentieren. Die vorgeschlagenen Suchmuster müssen in diesem Fall nur minimal angepasst werden.

2. **Manche gefundenen Suchmuster haben sich als nicht relevant erwiesen, da deren Parallelisierung kaum Beschleunigung mit sich bringt.** Hier wäre eine dynamische Analyse hilfreich, die Laufzeitverteilung im Programm untersucht und in das MAP integriert. So können die Suchmuster erweitert werden, indem sie die Laufzeiten der Aktivitäten berücksichtigen und vergleichen. Die Aktivitäten mit minimalen Laufzeiten können dann als nicht relevante aussortiert werden.

Das entstehende Objektmodell AOM kann auch für Codegenerierung verwendet werden. Wenn man eine Verbindung zwischen Modellen und den sequentiellen Code hat, dann könnte man die Suchmuster mit Hilfe von Architekturbeschreibungssprache TADL beschreiben lassen. Daraus kann später einen TADL-Compiler paralleler Code generieren.

ANHÄNGE

A. Abkürzungsverzeichnis

Abkürzung	Langbezeichnung und/oder Begriffserklärung
UML	Unified Modeling Language
.NET	.NET bezeichnet eine von Microsoft entwickelte Software-Plattform zur Entwicklung und Ausführung von Anwendungsprogrammen.
CLR	Common Language Runtime - die Laufzeitumgebung des .NET-Platforms
AD	UML Aktivitätsdiagramm
SD	UML Sequenzdiagramm
DI	Diagram Interchange - DI ist ein Standard der Object Management Group (OMG), der dazu dient, das Austauschformat für das Layout von Diagrammen im Rahmen der UML 2.0 festzulegen
WAR	Write-After-Read
WAW	Write-After-Write

B. Abbildungsverzeichnis

C. Tabellenverzeichnis

D. Literaturanalysen

[PF+04] Sabri Pllana, Thomas Fahringer, Johannes Testori, Siegfried Benkner, and Ivona Brandic: *Towards an UML Based Graphical Representation of Grid Workflow Applications*, 2004

[Gud10] Stefan Gudenkauf: *A Coordination-Based Model-Driven Method for Parallel Application Development*, 2010

[Stö05] Harald Störrle: *Semantics and Verification of Data Flow in UML 2.0 Activities*, 2005

[Woh05b] Petia Wohed1, Wil M.P. van der Aalst2,3, Marlon Dumas3, Arthur H.M. ter Hofstede3 and Nick Russell3. *Pattern-based Analysis of UML Activity Diagrams*, 2005

[Tou09] Georgios Tournavitis, Zheng Wang, Björn Franke, Michael F.P. O'Boyle: *Towards a Holistic Approach to Auto-Parallelization: Integrating Profile-Driven Parallelism Detection and Machine-Learning Based Mapping*, 2009

[Tou10] Georgios Tournavitis and Björn Franke: *Semi-Automatic Extraction and Exploitation of Hierarchical Pipeline Parallelism Using Profiling Information*, 2010

E. Literaturverzeichnis

[Abe10] Prof. Sebastian Abeck, Aleksander Dikanski, Michael Gebhart, Philip Hoyer, Ingo Pansa. Lecture: *Course unit: User interaction with model-driven application management*. 2010.

[DW+05] Alan Dennis, Barbara Wixom, and David Tegarden: *Systems Analysis and Design With UML 2.0*, 2005

[BG+03] Barros, J.P. and L. Gomes: *Actions as Activities as Petri nets*, in: J. Jürjens, B. Rumpe, R. France and E.B. Fernandey, editors, Proc. Ws. Critical Systems Development with UML, 2003, pp. 129–135

[BG+05] Henriette Baumann, Patrick Grassle, Philippe Baumann: *UML 2.0 in Action: A project-based tutorial*, Buch 2005

[KBK08] Hans Kleine Büning, Uwe Kastens: *Modellierung: Grundlagen und formale Methoden*, Buch, 2008

[BK08] Christel Baier, Joost-Pieter Katoen: *Principles of Model Checking*, Buch 2008

[Boc04] Conrad Bock: *UML 2 Activity and Action Models, Part 4: Object Nodes* J. Object Technology, 3 (2004)

[CH03] Krzysztof Czarnecki; Simon Helson: *Classification of Model Transformation Approaches*. In: OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture, 2003

[GD+08] Tom Gelhausen, Bugra Derre, Rubino Geiss, *Customizing GrGen.NET for Model Transformation*, GRaMoT '08: Proceedings of the 3rd International Workshop on Graph and Model Transformation, ACM, p. 17-24, Leipzig, Germany, May 2008.

[GL+09] Tom Gelhausen, Mathias Landhäußer, Sven J. Körner: *Automatic Checklist Generation for the Assessment of UML Models*, 2009

[GH+94] Erich Gamma, Richard Helm, Ralph E. Johnson: *Design Patterns. Elements of Reusable Object-Oriented Software*, Buch 1994

[GrGen] Jakob Blomer, Rubino Geiß, Edgar Jakumeit: *The GrGen.NET User Manual*. Jan 2013

[Hot11] Jason A. Poovey, Brian Railing, Thomas M. Conte: *Parallel Pattern Detection for Architectural Improvements*, 2011

[Jak08] Edgar Jakumeit: Erweiterung der Regelsprache eines Graphersetzungswerkzeugs um rekursive Regeln mittels Sterngraphgrammatiken und Paargraphgrammatiken, Jul 2008.

[Jen92] K. Jensen: *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*. Vol. I. EATCS Monographs on Theoretical Computer ScienceSpringer Verlag (1992)

[Kle09] Stephan Kleuker: Formale Modelle der Softwareentwicklung: Model-Checking, Verifikation, Analyse und Simulation, Buch 2009

[Lar04] Craig Larman: *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*, Buch 2004

[Med10] David J. Meder, Walter F. Tichy, *Parallelizing an Index Generator for Desktop Search*, University of Karlsruhe, Technischer Bericht, Nr. 2010-9

[MS+12] Korbinian Molitorisz, Jochen Schimmel, Frank Otto: *Automatic Parallelization using AutoFutures*, International Conference on Multicore Software Engineering, Performance and Tools (MSEPT), 2012

[Pan08] Dr. Victor Pankratius, Christoph A. Schaefer, Ali Jannesari, Prof. Walter F. Tichy. *Software Engineering for Multicore Systems: An Experience Report*, 2008

[Pan11] Dr. Victor Pankratius. Vorlesung: *Softwareentwicklung für moderne, parallele Plattformen*, 2011

[PA+12] Victor Pankratius, Ali-Reza Adl-Tabatabai, Walter Tichy: Buch *Fundamentals of Multicore Software Engineering*, 2012

[Poi11] Sergej Poimzew, Studienarbeit: Kontrollflußbasierte Leistungsschätzung zur Parallelisierung, 2011

[PP09] Dan Pilone, Neil Pitman: *UML 2.0 in a Nutshell (O'Reilly)*, Buch O'Reilly Media 2009

[RH+06] N. Russell, A.H.M. ter Hofstede, W.M.P. van der Aalst, and N. Mulyar. *Workflow Control-Flow Patterns: A Revised View*. BPM Center Report BPM-06-22, BPMcenter.org, 2006.

[RH+12] Louis M. Rose, Markus Herrmannsdoerfer, Steffen Mazanek, Pieter Van Gorp: *Graph and model transformation tools for model migration. Empirical results from the transformation tool contest*. 2012

[RQ+07] C. Rupp, S. Queins, and B. Zengler. UML 2 glasklar. Carl Hanser Verlag, 3rd. edition, 2007.

[Rod00] Rodrigues, R.W., *Formalising UML Activity Diagrams using Finite State Processe.*, Technical Report No. 0006 of the Ludwig-Maximilians-Universität, München, Inst. f. Informatik, 2000

[Rus07] Silvius Rus, Maikel Pennings, Lawrence Rauchwerger: *Sensitivity Analysis for Automatic Parallelization on Multi-Cores*, 2007

[Sho04] Martin Shoemaker: *Introducing UML: Object-Oriented Analysis and Design A .Net Perspective*, by (Apress, 2004; ISBN: 1590590872)

[SV+07] Thomas Stahl, Markus Völter, Sven Efftinge: *Modellgetriebene Softwareentwicklung. Techniken, Engineering, Management.* Mai 2007

[Tic12] Walter F. Tichy: Vortrag - *Die Multicore-Transformation und ihre Herausforderung an die Softwaretechnik*, Feb 2012

[Vol05] Olga Nicole Volgin: *Analysis of Flow of Control for Reverse Engineering of Sequence Diagram, 2005*

[WB+12] M. Widl, A. Biere, P. Brosch, U. Egly, M. Heule, G. Kappel, M. Seidl, H. Tompits: *Guided Merging of Sequence Diagrams*; Dresden (2012), 163-182.

[XC+01] X. Li, M. Cui, Y. Pei, Z. Jianhua, Z. Guoliang: *Timing Analysis of UML Activity Diagrams*, Proc. 4th Intl. Conf. on the Unified Modeling Language (UML 2001), LNCS, number 2185 (2001)

F. Hyperlinks

[MSDN_SD] MSDN Artikel: UML-Sequenzdiagramme
http://msdn.microsoft.com/de-de/library/vstudio/dd409389.aspx

[MSDN_AD] MSDN Artikel: UML-Aktivitätsdiagramme
http://msdn.microsoft.com/de-de/library/vstudio/dd409360.aspx

[OMG] Object Management Group Homepage
http://www.omg.org/spec/UML/

[UML] Unified Modeling Language auf Wikipedia.de
http://de.wikipedia.org/wiki/Unified_Modeling_Language

[UML_API] API-Referenz für UML-Modellierungserweiterbarkeit
http://msdn.microsoft.com/de-de/library/ee329525.aspx

[UML2.3] OMG Unified Modeling LanguageTM (OMG UML), Infrastructure
http://www.omg.org/spec/UML/2.3/Infrastructure/PDF/

[XMI] XML Metadata Interchange. OMG Formally Released Versions of XMI®
http://www.omg.org/spec/XMI/

H. Objektmodell AOM und Suchmuster

```
/*** AOM Nodes ***/
abstract node class uNode {
 id: int;
 Guid: object;
 Name: string;
 Stereotype: string;
 umlTypeName: string;
}

node class uCommerntar extends uNode {
 Text: string;
}

node class uNodeType extends uNode;

node class uObjectNode extends uNode {
 isOrdered: boolean;
 isUnique: boolean;
 UpperBound: int;
 Multiplicity: string;
}

node class uPassiveObject extends uObjectNode;
node class uActiveObject extends uObjectNode;

node class uActivityNode extends uNode {
 Number: int;
}
node class uExecNode extends uActivityNode;

node class uReadWriteAction extends uExecNode;
node class uComputationAction extends uExecNode;

node class uControllNode extends uNode {
 NodeType: string;
}
node class uInvocationAction extends uControllNode, uActivityNode;

/***          AOM Edges                    ***/
edge class uEdge {
 UpperBound: int;
 Synchrone: boolean;
 Precondition: string;
 Postcondition: string;
}

edge class uAssociation extends uEdge
       connect uNode[*] -- uNodeType[*], // connect node with type
                uActiveObject[*] -- uActivityNode[*];

edge class uObjectFlow  extends uEdge
       connect uObjectNode[*] --> uActivityNode[*],  // read object
                uObjectNode[*] <-- uActivityNode[*];  // write into object

edge class uControllFlow extends uEdge
       connect uActivityNode[*] --> uActivityNode[*];

edge class uUpdateObject extends uControllFlow
       connect uActiveObject[1] <-- uActivityNode[*];
```

```
using ObjectModell;

// ***************************************    //
//                  SUCHMUSTER                //
// ***************************************    //

// Gibt Anzahl der eingehenden Objektflüssen
rule inputsCount(xNode:uActivityNode):(int)
{
	iterated {
	 :uObjectNode --> xNode;
	 modify {
		eval { yield count = count +1; }
		}
	}

	modify {
		def var count:int = 0;
		return (count);
	}
}

// Gibt Anzahl der ausgehenden Objektflüssen
rule outputssCount(xNode:uActivityNode):(int)
{
	iterated {
	xNode --> :uObjectNode;
	 modify {
		eval {yield count = count +1; }
		}
	}

	modify {
		def var count:int = 0;
		return (count);
	}
}

// Gibt Anzahl der eingehenden Controllflussen
rule inputControlflowsCount(xNode:uActivityNode):(int)
{
	iterated {
	 :uNode -:uControllFlow-> xNode;
	 modify {
		eval {yield count = count +1; }
		}
	}

	modify {
		def var count:int = 0;
		return (count);
	}
}


// Findet triviale Schleifen
rule findSimlpeLoop():(uActivityNode) {
	action:uActivityNode --> action;
	modify {
		return (action);
	}
}
```

```
// ObjectMerge Muster
rule ObjectMerge(xNode:uActivityNode, forkNode:uControllNode,
joinNode:uControllNode):() {

	iterated {
	 action:uActivityNode -oldCF2:uControllFlow-> xNode;
	 modify {
			// alte wege löschen
			delete(oldCF1);
			delete(oldCF2);

			// neue wege festlegen
			//start -f0:uControllFlow-> forkNode;
			forkNode -f1:uControllFlow-> action;
			action -f2:uControllFlow-> joinNode;
		}
	}

	modify {
		// forkNode:uControllNode;
		// joinNode:uControllNode;

		eval {
			forkNode.Name = "Fork";
			joinNode.Name = "Join";
		}

		joinNode -n:uControllFlow -> xNode;

		emit("ObjectMerge found\n");
		// return(xNode, forkNode, joinNode);
	}
}

// ImplicitTermination Muster
rule FindImplicitTermination():(uActivityNode) {
	xNode: uActivityNode;
	sourceNode:uActivityNode\uInvocationAction -cFlow:uControllFlow-> xNode;

	negative {
		xNode -:uControllFlow-> :uNode;
	}

	modify {
		delete(cFlow);
		asyncNode:uInvocationAction;
		asyncNode -start:uControllFlow-> xNode;
		sourceNode -cNewFlow:uControllFlow-> asyncNode;

		emit("ImplicitTermination found\n");
		return(xNode);
	}
}

// Producer-Consumer Muster
rule FindProducerConsumer():(uPassiveObject) {
	xNode:uPassiveObject;
	producerObject:uObjectNode --> xNode --> consumerObject:uObjectNode;

	modify {
		emit("ProducerConsumer found\n");
		return(xNode);
```

```
	}
}

// Gibt Producer und Datenflussobjekt zurück
Rule getProducerActionFromObject(xNode:uPassiveObject):(uActivityNode,
uObjectNode)
{
	action:uActivityNode -:uObjectFlow-> passObject:uObjectNode -:uObjectFlow-
> xNode;

	modify {
		return (action, passObject);
	}
}

// Gibt Consumer und Datenflussobjekt zurück
rule getConsumerActionFromObject(xNode:uPassiveObject):(uActivityNode,
uObjectNode)
{
	action:uActivityNode <-:uObjectFlow- passObject:uObjectNode <-
:uObjectFlow- xNode;

	modify {
		return (action, passObject);
	}
}

// ParallelLoop Muster
rule FindParallelLoop {}

rule getLoop():(uControllNode,uActivityNode,uActivityNode) {
	xNode:uControllNode;
	xNode -:uControllFlow-> action1:uActivityNode -:uControllFlow->
action2:uActivityNode -:uControllFlow-> xNode;

	// keine Objektflusse auserhalb der Schleife
	negative {
		action1 -of1:uObjectFlow-> a:uActivityNode;
		action2 -of2:uObjectFlow-> b:uActivityNode;
	}

	modify {
		return (xNode, action1, action2);
	}
}



// Action Bunch
rule FindActionBunch():(uActivityNode, uActivityNode, uActivityNode,
uActivityNode)
{
	// first pair
	a1:uActivityNode -:uObjectFlow-> o1:uObjectNode -:uObjectFlow->
a2:uActivityNode;
	a1 -cf_a:uControllFlow-> a2;
	start:uNode -cf_sa:uControllFlow-> a1;

	// second pair
	b1:uActivityNode -:uObjectFlow-> o2:uObjectNode
	 -:uObjectFlow-> b2:uActivityNode;
	b1 -cf_b:uControllFlow-> b2;
```

```
	b2 -cf_be:uControllFlow-> end:uNode;

	// control but not objectflow
	a2 -cf_ab:uControllFlow-> b1;
	negative {
		a2 -:uObjectFlow-> o3:uObjectNode -:uObjectFlow-> b1;
	}

	modify {
		// Add fork/join nodes
		delete(cf_ab);

		// do fork
		forkNode:uControllNode;
		forkNode -f_a:uControllFlow-> a1;
		forkNode -f_b:uControllFlow-> b1;
		delete(cf_sa);
		start -:uControllFlow->forkNode;

		joinNode:uControllNode;
		a2 -a_j:uControllFlow-> joinNode;
		b2 -b_j:uControllFlow->joinNode;
		delete(cf_be);
		joinNode -:uControllFlow-> end;

		eval {
			forkNode.Name = "Fork";
			joinNode.Name = "Join";
		}

		return (a1, a2, b1, b2);
	}

}

rule DeleteEmptyObjectNodes() {
	x:uObjectNode;
	negative {
		:uNode --> x --> :uNode;
	}
	modify {
		delete(x);
	}
}
```