Generische Refactoring-Spezifikation für Korrektheitsbeweise in mehrsichtigen Modellsprachen

Der Fakultät für Elektrotechnik, Informatik und Mathematik (EIM) der
Universität Paderborn
zur Erlangung des akademischen Grades eines
Dr. rer. nat.

eingereichte Dissertation

von
Herr Dipl.-Inform. Thomas Peter Ruhroth
geboren in
Euskirchen

Genehmigte Fassung

Referent: Prof. Dr. Heike Wehrheim
Koreferent: Jun.-Prof. Dr.-Ing. Steffen Becker
Tag der Prüfung: 16. Juni 2011

Abstract

In software development it is necessary that program code and models can be maintained. Refactoring is a best practice in code development which can be used to improve the internal quality of given code. Refactoring means modifying code without changing its behaviour. It is also desirable to improve the internal quality of formal models. Unlike programs most models are not executable. Thus, refactoring techniques need to be adapted. For example, a suitable definition of behaviour preservation is needed. A technique often used in the context of models is the usage of multiple views on one model, e.g. to specify several aspects using different diagrams. This also implies the need to adapt existing refactoring techniques.

The aim of this work is to describe refactorings of formal models and to ensure the behaviour preservation of these refactorings. The language family Re$\mathcal{L}$ (Refactoring Language) is used to describe refactorings of simple imperative programs (FWHILE) and specifications given in an integrated formal method (CSP-OZ). In particular, the structure of Re$\mathcal{L}$ allows proving behaviour preservation without transferring the refactoring to a different representation.

Re$\mathcal{L}$ uses templates to describe the state of the code before and after the refactoring. These templates are divided into subtemplates. These subtemplates can be used in different views and therefore provide the possibility to describe multi-view refactorings. Also the structure of Re$\mathcal{L}$ allows a refactoring to be applied to a model as well as the formal analysis.

Re$\mathcal{L}$ is derived using techniques of *Domain Specific Languages*. Thus, Re$\mathcal{L}$ depends on the syntax definition of the language to be refactored. This process allows Re$\mathcal{L}$ to derive a Re$\mathcal{L}$-Instance for all languages that be described as Backus-Naur Form (BNF). The application is therefore not restricted to formal model languages, but can also be applied to programming languages.

The preservation of the behaviour can be proved using these templates. Techniques are presented to simplify these proofs. Observation points will help to reduce the actual proof to a certain area in the semantics of the model. Also the techniques for multi view models will be derived out of observation point. Three different types of interactions are distinguishable. Refactorings affecting a single view, refactorings with condition to other views, and refactorings with effects on multiple views.

Zusammenfassung

In der Softwareentwicklung sollen Code und Modelle wartbar sein. Refactorings sind ein bewährtes Hilfsmittel der Codeentwicklung, um die innere Qualität von Code zu verbessern. Refactorings sind Änderungen am Code, wobei das sichtbare Verhalten unverändert bleibt. Diese Verbesserungen sind auch für formale Modelle erstrebenswert. Modelle sind im Gegensatz zu Programmen meist nicht ausführbar, daher ist es notwendig die verwendeten Refactoringtechniken anzupassen. Beispielsweise muss eine brauchbare Definition der Verhaltenserhaltung gefunden werden, da ein Modell nicht ausführbar ist. Häufig werden mehrere Sichten in Modellen eingesetzt, um verschiedene Aspekte eines Modells abzubilden. Um mehrere Sichten unterstützen zu können, müssen bestehende Techniken weiter angepasst werden.

Ziel dieser Arbeit ist die Darstellung von mehrsichtigen Refactorings und die Sicherstellung der Verhaltenserhaltung der Refactorings für formale Modelle. Es wird die Sprachfamilie Re$\mathcal{L}$ (Refactoring Language) genutzt, um Refactorings von einer einfachen imperativen Programmiersprache (FWHILE) und einer integrierten formalen Methode (CSP-OZ) zu beschreiben. Re$\mathcal{L}$ ermöglicht durch seine Struktur, dass die Verhaltenserhaltung direkt bewiesen werden kann, ohne auf eine andere Darstellung zurückgreifen zu müssen.

Für die Beschreibung wird die Sprachfamilie Re$\mathcal{L}$ genutzt, die Refactorings anhand des Codezustandes vor und nach dem Refactoring beschreibt. Dies wird durch Templates beschrieben, deren Subtemplates Refactorings mehrsichtiger Modelle unterstützen. Die Struktur von Re$\mathcal{L}$ erlaubt sowohl die Ausführung eines Refactorings in einer Entwicklungsumgebung als auch die formale Analyse.

Re$\mathcal{L}$ wird mit Techniken von *Domain Specific Languages* aus der Syntaxdefinition der zu refaktorisierenden Sprache abgeleitet. Durch diesen Prozess ist Re$\mathcal{L}$ für alle Sprachen ableitbar, deren Syntax als Backus-Naur Form (BNF) beschrieben werden können. Das Einsatzgebiet ist daher nicht auf formale Modellsprachen eingeschränkt, sondern kann auch auf Programmiersprachen ausgedehnt werden.

Die Verhaltenserhaltung kann auf Grundlage der Templates von Re$\mathcal{L}$ gezeigt werden. Dazu werden einige Techniken vorgestellt, um die Beweisführung zu vereinfachen. Die Grundlage bilden die Beobachtungspunkte, die für einen Ausschnitt der Codes oder der Spezifikation zeigen, dass das Refactoring für diesen Ausschnitt das Verhalten nicht ändert. Aus der Verhaltenserhaltung an einem Beobachtungspunkt kann die Gesamtverhaltenserhaltung geschlossen werden. Für mehrsichtige Modelle wird ausgehend von den Beobachtungspunkten die Wechselwirkung von Refactorings von verschiedenen Sichten betrachtet. Dabei sind drei verschiedenen Arten von Wechselwirkungen unterscheidbar: Refactorings, die auf einer Sicht separat betrachtet werden können, Refactorings mit Bedingungen an andere Sichten und echt mehrsichtige Refactorings.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

Software ist in unserer heutigen Welt allgegenwärtig, auch wenn ein großer Teil der Software nicht wahrgenommen wird. Ob in Fernsehgeräten, Autos oder Smartphones, die meisten komplexen technischen Systemen beinhalten Software. Wenn ein System nicht funktioniert, kann häufig nicht direkt festgestellt werden, ob die Ursache eines Fehlers in der Software oder in einer anderen Komponente liegt. Oft befindet sich der Fehler in der Software. Ein Grund hierfür liegt in den schnellen Entwicklungszyklen von Software. Während der Nutzungszeit eines Smartphones gibt es von einzelnen Anwendungen regelmäßig Updates. Im Gegensatz dazu bleibt die Hardware während der gesamten Nutzungszeit meist unverändert. Sie wird daher meist aus erprobten Komponenten erstellt und einer sorgfältigen Qualitätskontrolle unterzogen. Software hingegen kann schnell und kostengünstig geändert werden. Entsprechend häufig wird die Software geändert.

Damit der Nutzer nicht durch fehlerhafte Änderungen verärgert wird, ist es wichtig, eine korrekt funktionierende Software bereitzustellen. Diese Fehlerfreiheit ist eine notwendige Grundeigenschaft. Im Gegensatz hierzu, macht ein Softwareentwickler, wie Menschen allgemein, regelmäßig Fehler. Balzert führt dies im Kapitel *Zur Psychologie des Programmierens* in [Bal99] auf die Verhaltens- und Denkmodelle des Menschen zurück. Das Denken des Menschen ist unter anderem durch die Induktion von Sachverhalten und das Scheinwerfer-Prinzip gesteuert. Induktion im Sinne der Psychologie bedeutet, dass aus einem häufig auftretenden Schema eine allgemeine Behauptung ableitet wird. Damit kann die Aussage „Jedes Auto hat vier Räder“ abgeleitet werden, aber es gibt Ausnahmen (z.B. das TW4XP als dreirädiges Elektroauto [TW4]). Das Scheinwerfer-Prinzip sagt aus, dass der Mensch von den zur Verfügung stehenden Daten immer nur einen kleinen Teil bewusst verarbeitet. Dabei fallen oft Einzelheiten unter den Tisch. Diese beiden Beispiele zeigen, dass das Denken des Menschen und somit auch des Entwicklers auf Strukturannahmen aufbaut. Grams [Gra90] stellt fest, dass viele Fehler auf diese und einige andere psychologische Probleme zurück zu führen sind.

Dies führt neben Problemen beim Verständnis der Anforderungen und der Umgebung oft zu Problemen mit der Struktur von Programmen. Ein Programmierer hat implizit Strukturannahmen, die durch seine bisherige Entwicklungsarbeit geprägt sind. Aufgrund dieser Annahmen entwickelt oder verändert er ein Programm. Wenn diese Strukturannahmen nicht erfüllt sind, kommt es unweigerlich zu Fehlern in der Entwicklung. Um die Fehler zu minimieren, ist es eine gute und erprobte psychologische Methode [Mar09], innerhalb der Entwicklung eine einheitliche Struktur der Programme zu gewährleisten. Dies schafft eine Grundlage für die Induktion und das Scheinwerfer-Prinzip des Entwicklers, die der Pro-

grammwirklichkeit entspricht. Das bedeutet, dass immer wieder die gleiche Struktur für ähnliche Sachverhalte verwendet wird. Dies wird in der industriellen Softwareentwicklung auf verschiedenen Ebenen berücksichtigt. So werden z.B. die Methoden, die eine Information über den inneren Zustand einer Klasse liefern als *getter* implementiert und so benannt. Schon durch den Namen weiß der Programmierer, dass die Methoden eine bestimmte Aufgabe erfüllen. Vorschläge, wie solche Strukturen zu erstellen sind, gibt es auf verschiedenen Ebenen der Entwicklungsarbeit. Für das Programmieren gibt es Style-Regeln [Mic97], wie z.B. das Verwenden von Methodenbezeichnungen wie *get* und *set*. Auf den höheren Abstraktionsebenen gibt es Vorschläge zur Struktur von Klassensystemen [GHJV85] und der Systemebene [BMR+98, Fow02].

Ein System, welches mittels dieser Ansätze aufgebaut wurde, ist leichter zu warten und zu verändern als ein strukturlos konstruiertes System. Leider gibt es zwei Probleme: Es existiert zum Einen viel Software, bei der dies nicht beachtet wurde und zum Anderen wird bei jeder Änderung der Software, die unstrukturiert durchgeführt wird, die Struktur zerstört. Es besteht das Bedürfnis, die innere Struktur eines Programms zu verbessern und dabei so zu verändern, dass eine verständliche Programmstruktur erstellt wird. Um keine Fehler einzuführen, ist es unabdingbar Veränderungen so durchzuführen, dass sich die Funktion der Software nicht verändert. Dies ist auch sinnvoll in der Hinsicht, dass der Prozess der Veränderung von dem der Strukturierung des Programms eindeutig getrennt werden kann. Die Veränderung der Struktur eines Programms, ohne dabei das Verhalten zu verändern, wird im Allgemeinen *Refactoring* genannt.

1.1 Refactoring

Refactoring kann für verschiedene Bereiche betrachtet werden. Die ersten Betrachtungen verbessern die Qualität von Programmcode [Opd92, Rob99]. Wenn ein Refactoring durchgeführt wird, soll das Programm vor und nach dem Refactoring das gleiche Verhalten haben. Dies wirft zwei Fragen auf: Was ist das Verhalten eines Programms und was heißt es dieses zu erhalten. Bei Programmen wird das Verhalten meist als die Interaktion mit dem Benutzer verstanden. Das beinhaltet alle Ausgaben eines Programms. Wenn der Nutzer der Software nicht unterscheiden kann, ob er mit dem Original oder der refaktorisierten Version arbeitet, dann ist das Verhalten erhalten worden. Oft wird diese Eigenschaft überprüft, indem Tests für die Software benutzt werden. Diese Tests ersetzen den Benutzer der Software. Wenn die Tests nach dem Refactoring die gleichen Ergebnisse wie vor dem Refactoring liefern, wird angenommen, dass das Verhalten erhalten wurde. Die Qualität dieser Aussage hängt stark von der Qualität der Tests ab. Schon früh wurde erkannt, dass die automatisierte Durchführung von Refactorings Vorteile bringt. Zum Einen wird auf das Tool vertraut, das vermeidlich die Verhaltenserhaltung garantiert und zum Andern wird der Entwickler von der stupiden Umsetzung der genau beschriebenen Refactorings befreit.

Refactoring kann aus verschiedenen Blickwinkeln betrachtet werden. Meist trifft ein *Software-Entwickler* in Tools auf Refactorings. Ein Entwickler möchte ein Refactoring schnell durchführen können und sich auf die Korrektheit der Änderungen verlassen. Für ihn ist Refactoring ein Hilfsmittel, um seine Ziele zu erreichen. Die Refactorings, die er nutzt, sind meist von einer anderen Person erstellt worden. Dieser *Modellierer* der Refactorings kümmert sich um die Erstellung der Refactorings. Eine weitere wichtige Rolle in der Erstellung ist der *Analyst*, er untersucht ob die Refactorings wirklich die geforderten Eigenschaften besitzen.

1.2 Refactorings für formale Methoden mit mehreren Sichten

Falsche Strukturannahmen oder fehlerhafte Induktion, wie sie zur Entwicklung von Refactorings für Programme geführt haben, treten auch bei Softwaremodellen auf. Obwohl Refactoring auf Modelle übertragen werden kann, haben Modelle eigene, anders geartete Probleme bzw. Fragestellungen. Modelle sind eine Abstraktion, und im Gegensatz zu Software in manchen Aspekten unvollständig. Sie werden im Rahmen des Model-driven Development [SB88] in Schritten mit Informationen angereichert (verfeinert) und schließlich implementiert. Damit Refactorings auf Modellen angewandt werden können, muss die Theorie des Refactorings auf Softwaremodelle übertragen werden. Weiter muss festgelegt werden, was die Verhaltenserhaltung von Modellen bedeutet. Ein Modell kann in der Regel nicht wie ein Programm ausgeführt werden. Eine Frage ist daher, was das Verhalten eines Modells ausmacht. Eine Möglichkeit ist die Verhaltenserhaltung über die Semantik eines Modells zu definieren. Zum Beispiel kann eine Relation auf Grundlage der Semantik definiert werden, die die Verhaltenserhaltung beschreibt. Eine brauchbare Relation stellt die Verfeinerung (Refinement) dar. Refinement stellt sicher, dass das verfeinerte Modell nichts durchführen kann, was nicht auch das ursprüngliche Modell hätte durchführen können.

Modelle haben neben der fehlenden Ausführbarkeit noch eine weitere Eigenschaft, welche Refactorings beeinflussen. Modelle besitzen häufig mehrere Sichten (*mehrsichtige Modelle*), die mit jeder Sicht einen speziellen Sachverhalt beleuchten. Eine Sicht innerhalb der UML [UML] ist das Klassendiagramm, welches die Struktur der Klassen und deren Beziehungen untereinander beschreibt. In dieser Klassensicht fehlen die Informationen über das innere oder äußere Verhalten der Klasse. Diese Informationen werden durch andere Diagramme dargestellt. Das Statechart zeigt, wie sich der Zustand der Klasse verändern kann. Das Sequenzdiagramm und das Kollaborationsdiagramm beschreiben die Interaktion der Objekte untereinander. Wenn mehrere Sichten zu einem Modell gehören, müssen die Sichten konsistent zueinander sein, das heißt, sie müssen Charakteristika, die sich in mehreren Sichten gleichzeitig finden, teilen. Auch an dieser Stelle werden neue Fragen im Umfeld der Verhaltenserhaltung aufgeworfen. So wird ein Verständnis benötigt, wie sich das Verhalten einer Sicht auf das Gesamtverhalten auswirkt. Dies wirkt sich auch auf die Verhaltenserhaltung und die Modellierung von Refactorings aus. Refactorings mehrsichtiger Modelle müssen mit zusammengehörigen Änderungen in den Sichten umgehen können. Dazu gehört, dass Eigenschaften aus der Notation einer Sicht in die Notation einer anderen Sicht übertragen werden können müssen. Auch die Verhaltenserhaltung muss sich aus der Verhaltenserhaltung der Sichten zusammensetzten können.

Ziel im Allgemeinen ist es, Refactorings zu definieren, die auf Modellen durchgeführt werden können. Dazu müssen die Refactorings an die geänderten Gegebenheiten angepasst werden. Formale Modelle bieten sich für die Untersuchung besonders an. Formale Modelle haben eine formale Semantik und mehrsichtige formale Modelle besitzen meist nur wenige Sichten. Die Ergebnisse, die bei den mehrsichtigen formalen Modellen gewonnen werden, lassen sich häufig auch auf nicht-formale mehrsichtige Modelle, wie die UML, übertragen.

1.3 Ziel: Beweisbarkeit der Verhaltenserhaltung von Refactorings für mehrsichtige Modelle

Ein Refactoring soll das Verhalten eines Modells oder einer Software nicht verändern. Eine Frage dabei ist, wie dies gewährleistet werden kann. Man kann naiv darauf vertrauen, dass

die Durchführung eines Refactoring das Verhalten des betrachteten Systems erhält. Dies setzt voraus, dass sowohl die Refactorings an sich als auch die Durchführung korrekt sind. Leider ist dieses Vertrauen sowohl bei der manuellen Nutzung als auch bei der Nutzung von Tools nicht berechtigt. Selbst in erprobten Entwicklungssystemen wie Eclipse werden regelmäßig Fehler in der Refactoring-Komponente gefunden [EESV08]. Dabei werden zwei Arten von Fehlern unterschieden: falsche Refactorings und Umsetzungfehler. Refactorings sind falsch, wenn sie nicht in allen Fällen das Verhalten erhalten. Bei einem Umsetzungfehler ist die Implementierung eines Refactorings fehlerhaft. Es wird also nicht das gewünschte Refactoring ausgeführt. Es ist somit notwendig, zu prüfen, ob ein Refactoring das Verhalten erhält.

Die erste und heute immer noch häufig benutzte Methode zur Prüfung der Verhaltenserhaltung ist das *Testen*. Da häufig Testfälle für die Software vorliegen, ist dies eine einfache und naheliegende Lösung. Wenn die Software vor und nach dem Refactoring die gleichen Testfälle erfolgreich ausführt, wird angenommen, dass sich das Verhalten nicht verändert hat. Testen kann niemals alle Fälle betrachten, und somit werden Fehler häufig übersehen. Bei Modellen kann das Testen häufig nicht angewandt werden, da Modelle in der Regel nicht ausführbar oder simulierbar sind. Vollständige Korrektheit lässt sich nur mit formalen Techniken gewährleisten. Wenn die Zielsprache eine formale Semantik hat, kann durch Beweistechniken gezeigt werden, dass das Verhalten vorher und nachher gleich ist. Diese *Postvalidierung* muss wie das Testen nach jedem Refactoring durchgeführt werden. Beide Techniken sind aufwändig und benötigen für die Validierung deutlich länger als die Durchführung des eigentlichen Refactorings. Der Zeitverbrauch kann optimiert werden, indem mehrere Refactorings vor der Validierung durchführt werden. Es ist wünschenswert, eine *Prävalidierung* durchzuführen, da dies die Prüfung des Refactorings vor dessen Anwendung verlagert und nicht bei der Anwendung betrachtet werden muss. Ziel ist es dabei, die Verhaltenserhaltung *vor* der Anwendung des Refactorings zu prüfen. Eine Prävalidierung mittels Testen wird bei der Erstellung von Entwicklungssystemen (z.B. Eclipse) angewandt. Wie oben schon erläutert, sind diese Umsetzungen dennoch unvollständig. Das zeigt, dass hier geeignetere Techniken benutzt werden müssen. Wünschenswert ist, die Verhaltenserhaltung eines Refactorings formal Beweisen zu können.

1.4 Ziel: Eine Sprache zur automatischen Durchführung von Refactorings

Wenn Refactorings validiert sind, müssen diese Refactorings angewandt werden können. Um eine fehlerhafte Übertragung des Refactoring zu verhindern, sollte die formale Spezifikation und die Beschreibung der Durchführung des Refactorings ausschließlich in einer Sprache beschrieben werden. Durch diese Anforderung werden Übertragungsfehler zwischen der Validierten und der „ausführbaren“ Fassung verhindert. Eine geeignete Umgebung für die Durchführung des Refactorings kann unabhängig von der Zielsprache entworfen werden. Das hat den Vorteil, dass sich die Übertragungslücke stark verkleinert. Damit ein Tool, welches die Refactoringsprache benutzt, mehrere Zielsprachen unterstützen kann, sollte die Refactoringbeschreibung für verschiedene Zielsprachen geeignet sein.

Die Anforderungen an eine solche Sprache sind vielfältig. Die Sprache muss die Durchführung des Refactorings so beschreiben, dass sie durch ein Programm ausgeführt werden kann. Dies kann in Form eines Algorithmus oder einer Transformationsbeschreibung geschehen. Bei der Durchführung werden oft von einem Benutzer Eingaben verlangt, z.B. muss

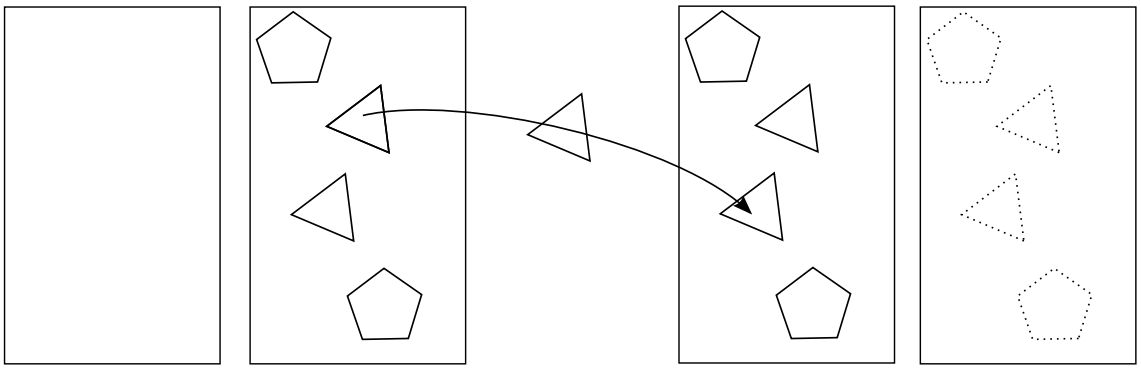


Abbildung 1.1: Vereinfachte Übersicht über das Template-Prinzip von Re$\mathcal{L}$: Ein gegebener Code (links) wird durch die Meta-Variablen zerlegt (zweiter Block). Die Inhalte der Meta-Variablen werden zur Erstellung eines neuen Codes verwendet (Pfeil). Im letzten Schritt (rechts) wird der Programmrumpf mit dem Inhalt der Meta-Variablen verschmolzen.

er beim Umbenennen einer Variable einen neuen Namen angeben. Dies kann durch die Parametrisierung von Refactorings unterstützt werden. Da eine Entwicklungsumgebung wie Eclipse [Fou] mehrere Programmiersprachen unterstützt, sollte eine Refactoringbeschreibung auch für verschiedene Sprachen verwendbar sein. Der Vorteil ist, dass nicht für jede unterstützte Sprache die Refactoringunterstützung neu implementiert werden muss. Eine zentrale Umsetzung steigert somit die Vielseitigkeit der Refactoringumsetzung. Von einer Fehlerbeseitigung im Tool profitieren alle Zielsprachen.

Aus Sicht des Refactoring-Modellierers soll die Beschreibung eines Refactorings leicht nachzuvollziehen sein. Wenn neue Refactorings erstellt werden, möchte sich der Modellierer schnell, ohne großem Aufwand, mit den Kernbereichen eines Refactorings auseinandersetzen können. Dazu ist es hilfreich, wenn die Refactoringsprache die Durchführung direkt beschreibt. Eine direkte Manipulation von inneren, abstrakten Datenstrukturen ist hierfür wenig hilfreich.

Der Analyst stellt wiederum andere Anforderungen. Die Beschreibung muss für ihn in einer Form vorliegen, die es ihm erlaubt die Eigenschaften (Verhaltenserhaltung, Informationen über die Codeverbesserung) des Refactorings zu bestimmen. Er muss prüfen können, ob das Refactoring zu syntaktisch richtigem Code führt. Ein weiterer Wunsch des Analysten ist es, die Verhaltenserhaltung eines Refactorings vor der Ausführung feststellen zu können.

1.5 Die Idee von Re$\mathcal{L}$

In dieser Arbeit wird als Verwirklichung der Ziele die Refactoringbeschreibungssprache Re$\mathcal{L}$ entwickelt, welche die gestellten Anforderungen erfüllen soll. Die Idee hinter der Sprache ist, mittels Templates (Schablonen) den Zustand vor und nach dem Refactoring zu beschreiben. Eine zentrale Eigenschaft der Templates ist, dass sie Lücken enthalten, in denen Code eingefügt werden kann. Abbildung 1.1 zeigt dies. Auf der linken Seite ist ein Block zu sehen, welcher ein Stück Programmcode repräsentiert. Aus diesem werden Code-Stücke mittels eines Templates heraus getrennt (Fünfecke und Dreiecke), wie es in dem zweiten Block dargestellt ist. Der Rest des Codes wird übernommen und die herausgelösten

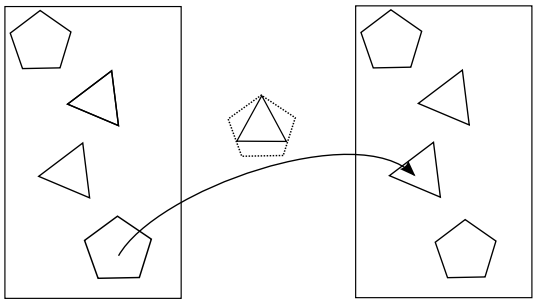


Abbildung 1.2: Modifikation von Codestücken in Re$\mathcal{L}$

Stücke werden an anderen Stellen wieder eingesetzt (dritter Block) und anschließend zu dem neuen Programm verschmolzen (rechter Block). Diese Lücken werden mit Hilfe von Meta-Variablen definiert. Der Zusatz „Meta“ grenzt die Variablen von einem evtl. vorhandenen Variablenbegriff der Zielsprache ab. Der Inhalt einer Meta-Variablen entspricht einem Stück des Codes und lässt sich durch eine Produktion aus der Grammatik der Zielsprache beschreiben. Grob gesagt, können mit den Meta-Variablen Codestücke an andere Stellen verschoben werden, um den Code zu verändern. Da dies nicht ausreicht um beliebige Modifikationen durchzuführen, beinhaltet Re$\mathcal{L}$ die Möglichkeit, neue Code-Stücke zu erstellen. In Abbildung 1.2 ist dies durch die Umwandlung eines 5-Eckes in ein Dreieck illustriert. Dabei wird ein neuer Wert einer Meta-Variablen berechnet, der dann im rechten Template zur Erstellung des Codes verwendet wird. Dabei kann, wie in der Grafik angedeutet, ein Teil eines Code-Stückes entfernt, aber auch hinzugefügt oder ganz neu berechnet werden.

1.6 Beweisbare Verhaltenserhaltung von Refactorings

Die Verhaltenserhaltung ist die wichtigste Eigenschaft von Refactorings. Bei der Überprüfung der Verhaltenserhaltung kann zwischen Postvalidierung und Prävalidierung unterschieden werden. Bei der Postvalidierung wird die Verhaltenserhaltung nach der Durchführung des Refactorings anhand des Codes durchgeführt. Die Prävalidierung hat zum Ziel, schon vor der Ausführung allgemein für das Refactoring zu zeigen, dass das Verhalten erhalten bleibt. In dieser Arbeit wird auf das Ziel Prävalidierung hingearbeitet. Die Prävalidierung wird mittels Beweis der Verhaltenserhaltung für BNF-basierte Sprachen, die eine formale, kompositionelle Semantik haben, realisiert. Die Beschreibung des Refactorings verringert den Beweisaufwand stark. Die Sprache Re$\mathcal{L}$ ist durch die Meta-Variablen darauf ausgelegt, die Beweisführung zu vereinfachen. Im Prinzip wird die Semantik des Codes auf einen Ausdruck in der Semantik der Meta-Variablen reduziert. Damit ist eine allgemeine Beweisführung mittels Regeln innerhalb der semantischen Domäne möglich.

Abbildung 1.3 gibt einen Überblick über das Vorgehen. Das Refactoring eines Code Stücks zu dem refaktorisierten Code wird durch zwei Templates (BeforeTemplate und AfterTemplate) mittels Re$\mathcal{L}$ beschrieben. So wie dem Code eine Semantik gegeben wird, kann auch dem Template in derselben semantischen Domäne eine Semantik gegeben werden. Die Semantik eines Templates steht zu der Semantik des Codes in ähnlicher Relation wie das Template und der Code: Jedes Template beschreibt eine Menge von Code. Dementsprechend beschreibt die Semantikumsetzung in die semantische Domäne der Zielsprache, die Semantik aller Codes, die dem Template entsprechen.

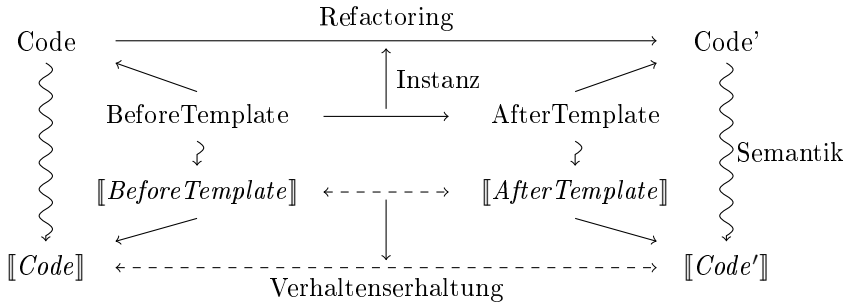


Abbildung 1.3: Zusammenhänge der Verhaltenserhaltung von Code und der Verhaltenserhaltung der Templates

Der Beweis wird dann auf den Semantiken der Templates geführt und auf die Semantiken des Codes übertragen. Diese Übertragung stellt schließlich die Verhaltenserhaltung der Durchführungen des Refactorings sicher.

1.7 Überblick über diese Arbeit

Der Inhalt der Arbeit konzentriert sich auf drei zentrale Aspekte von Refactorings. Der erste zentrale Aspekt ist Beschreibung eines Refactorings, so dass dieses Refactoring sowohl für die Ausführung als auch für die Prävalidierung mittels Beweisen geeignet ist. Die Familie der Beschreibungssprache Re$\mathcal{L}$ stellt als Domain Specific Language (DSL), für jede BNF-basierte Sprache X, in der Refactorings durchgeführt werden sollen, eine Instanz Re$\mathcal{L}_X$ zur Verfügung. Ein Vorteil ist, dass die Korrektheit des Codes nach dem Refactoring, durch ein Werkzeug anhand der Beschreibung geprüft werden kann. Damit wird im Vergleich mit anderen Sprachen (z.B. [Ver08, MB07, Kur08]) ein wichtiger Schritt in der Validierung eines Refactorings automatisiert. Re$\mathcal{L}$ ist die erste Beschreibungssprache, die in ihrer Definition die Unterstützung mehrerer Verhaltenserhaltungsbegriffe vorsieht. Für eine Menge von Refactorings für eine Sprache, hier Repository genannt, können verschiedene Verhaltenserhaltungsbegriffe definiert werden. Für jedes Refactoring wird angegeben, welche Arten des Verhaltens erhalten werden. Dies ist nötig, weil in Modellen verschieden Eigenschaften verschieden genau spezifiziert sein können. Wenn in einem genau spezifizierten Bereich ein Refactoring angewandt wird, sollte daher ein starker Verhaltenserhaltungsbegriff verwendet werden können. Im Gegensatz dazu kann an weniger spezifizierten Stellen ein schwächerer Begriff benutzt werden.

Der zweite große Aspekt dieser Arbeit ist die Prävalidierung von Refactorings. Ziel der Prävalidierung ist es, vor der Ausführung eines Refactorings zu zeigen, dass es das Verhalten erhält. Die Verhaltenserhaltung wird in dieser Arbeit als Relation aufgefasst, die als Teil eines Refactorings angesehen wird. Die Techniken zur Prävalidierung werden durch das Design von Re$\mathcal{L}$ unterstützt. Für eine kompositionelle Semantik sind die meisten Beweise der Verhaltenserhaltung einfache Folgerungen. Damit wird auch eine positive Antwort auf die „Challenge Proposal : Verification of Refactorings“ [SEdM08a] gegeben.

Die Betrachtung mehrsichtiger Modelle im Zusammenhang mit Refactorings stellen den dritten Aspekt dieser Arbeit dar. Die Mehrsichtigkeit zieht sich durch alle Bereiche von Refactorings. Für diese Arbeit muss die Mehrsichtigkeit besonders in der Definition von

Re$\mathcal{L}$ beachtet werden. Da Re$\mathcal{L}$ auch die Grundlage für die vorgestellten Beweistechniken darstellt, ist eine Übertragung auf die Verhaltenserhaltungsbeweise naheliegend.

Die ersten beiden Kapitel befassen sich mit Sprachen, deren Semantik und der Möglichkeit, Modelle mit diesen Sprachen zu beschreiben. Bei den Modellen wird außerdem die Aufteilung von Modellen auf Sichten betrachtet. Jede Sicht stellt ein Fenster auf einen Teilbereich des Modells dar. Sichten unterstützen dabei das Scheinwerferprinzip.

In Kapitel 5 wird die Sprache Re$\mathcal{L}$ zur Beschreibung von Refactorings eingeführt. Dabei spielt die Theorie der Sprachen in zweierlei Hinsicht eine Rolle. Zum Einen sind es Sprachen, die Gegenstand von Refactorings sind. Zum Anderen ist das eingeführte Re$\mathcal{L}$ selber eine Sprache. Anschließend wird in Kapitel 6 gezeigt, wie mit Re$\mathcal{L}$ Refactorings modelliert werden. Die Verhaltenserhaltung und Beweistechniken, um diese Verhaltenserhaltung zu validieren, ist Thema des Kapitels 7. Mit der Konklusion und einem Ausblick endet die Arbeit (Kapitel 8).

Kapitel 2

Sprachen

Sprachen sind die Grundlage aller Kommunikation. Für eine erfolgreiche Kommunikation müssen Regeln eingehalten werden, denn bei einem unterschiedlichen Verständnis einer Sprache können Informationen nicht übermittelt werden. Es muss klar sein, welche Zeichen verwendet werden, wie diese Zeichen zu Wörtern und Sätzen zusammengebaut werden, wann ein Satz sinnvoll ist und welche Bedeutung der Satz hat. Besonders bei Programmiersprachen, Modellen und Formalen Methoden muss dies gegeben sein.

Die Syntax einer Sprache wird in den ersten beiden Abschnitten dieses Kapitels behandelt. Dazu werden zunächst die Zeichen, das heißt die Symbole, die verwendet werden können, aufgezählt. Die Menge der Zeichen sind das Alphabet der Sprache. Aus den Zeichen des Alphabets werden die Wörter und Sätze der Sprache zusammengesetzt. Nach der Syntax werden die Wohlgeformtheit und die Semantik festgelegt. Die Wohlgeformtheit beschäftigt sich mit der Frage, was sinnvolle Sätze sind. Diesen sinnvollen Sätzen wird durch die Semantik eine Bedeutung gegeben.

Die Semantik und die Wohlgeformtheit hängen eng zusammen. In vielen Sprachen ordnet eine Semantik nicht allen syntaktisch erlaubten Wörtern bzw. Sätzen eine Bedeutung zu. So ist der Satz „Nils und Björn sind.“ in der deutschen Sprache syntaktisch erlaubt, weil ein Satz aus Subjekt und Prädikat innerhalb der deutschen Sprache erlaubt ist (z.B. ist „Björn schläft.“ ein solcher Satz). „Nils und Björn sind.“ kann aber keine Bedeutung zugemessen werden, er ist nicht vollständig. Ein Satz, dem eine Bedeutung zumessen wer-

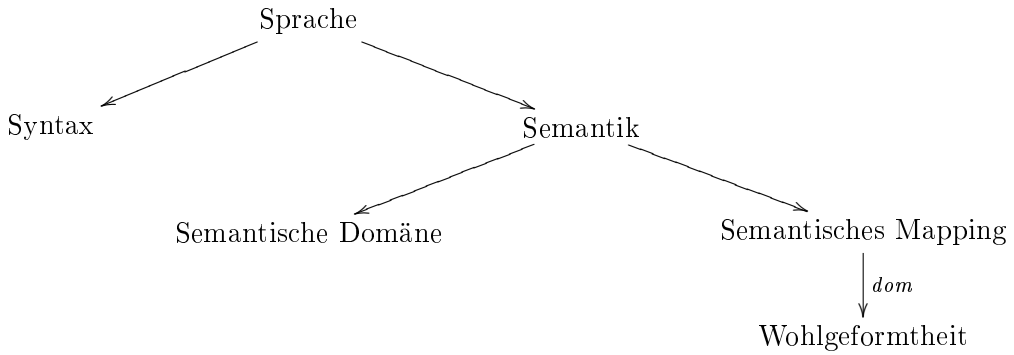


Abbildung 2.1: Zusammenhang von Sprache, Syntax und Semantik. (Adaptiert nach [AS88])

den kann, ist *wohlgeformt*. Die Wohlgeformtheit wählt aus einer Sprache die Teilmenge der Wörter bzw. Sätze aus, denen eine Bedeutung gegeben werden kann. Das wird auch im Sprachgebrauch des Compilerbaus deutlich, der die Wohlgeformtheit als *statische Semantik* bezeichnet. Im Folgenden wird kurz ein Überblick über die Struktur von Sprachen, wie sie in der Informatik genutzt werden, gegeben. Dabei werden die wichtigsten Begriffe dieser Arbeit erläutert und Notationen eingeführt. Abbildung 2.1 zeigt die Zusammenhänge der in den nächsten Abschnitten beschriebenen Begriffe und Konzepte.

Formal wird eine Sprache durch ein 4 Tupel beschreiben [AS88], wobei ein kleines sprachliches Problem auftritt. Das Wort „Sprache“ wird hier in zwei verschiedenen Bedeutungen benutzt. Der Begriff *formale Sprache*, im Sinne von [HMU02], wird für die Menge der Elemente der Sprache genutzt. Dies ist bei Java die Menge aller syntaktisch möglichen Java-Programme. Die Definition selber definiert was eine *Sprache* ist. Die formale Sprache ist ein Teil der Sprache, in der die formale Sprache durch die Semantik ergänzt ist.

Definition 1 (Sprache) *Eine Sprache P ist ein Tupel $(A, L, D, [\![.]\!])$ wobei A ein endliches Alphabet, $L \subset A^*$ eine formale Sprache, D eine Menge (semantische Domäne) und $[\![.]\!] : L \nrightarrow D$ eine Abbildung in die Menge D (semantisches Mapping) ist.*

Die Mengen A und L bilden die Syntax. Die Menge D und die Abbildung $[\![.]\!]$ stellen die Semantik dar. Auffallend ist, dass die Wohlgeformtheit sich nicht in einem Teil des Tupels ausdrückt. Die Abbildung, die später semantisches Mapping genannt wird, muss nicht jedem Element aus L eine Bedeutung D zuordnen. Es ist vielmehr so, dass die Domäne des semantischen Mapping nur eine Teilmenge der Sprache selber ist:

$$\mathrm{dom}[\![.]\!] \subset L \subset A^*$$

Somit ist eine wohlgeformte Sprache L' als Domäne der semantischen Zuordnung definiert:

$$L' = \mathrm{dom}[\![.]\!]$$

Die Wohlgeformtheit vereinfacht die Darstellung der Semantik, da das semantische Mapping als totale Funktion definiert werden kann. In dem Sonderfall, dass die Syntax schon alle wohlgeformten Programme erzeugt ($L = L'$), wird die Wohlgeformtheit stillschweigend weggelassen. Im Folgenden sollen die einzelnen Bestandteile genauer betrachtet werden. Dazu wird erst die Syntax betrachtet. Die Wohlgeformtheit wird als Menge zusätzlicher Regeln, die implizit den Definitionsbereich des semantischen Mapping ergeben, definiert.

2.1 Syntax

Eine Sprache ist mehr als eine lose Anordnung von Zeichen. Sprache hat Struktur. Dies gilt sowohl für Sprachen im üblichen Sinne, die als Ketten von Zeichen geschrieben werden, wie Programmiersprachen (Java [javb], C [KR78]) oder mathematische Sprachen (Prädikatenlogik) als auch für graphische Sprachen wie UML [UML] oder ER-Diagramme [Che76]. Die Syntax beschreibt diese Struktur, indem sie definiert, welche Zeichen möglich sind und wie diese angeordnet werden können.

Entsprechend der Struktur verschiedener Sprachen (graphisch oder nicht graphisch) gibt es verschiedene Ansätze, die Syntax zu beschreiben. Für graphische Sprachen, wie die UML, werden entweder natürlichsprachliche Beschreibungen verwendet oder es wird ein

Bootstrapping Ansatz wie bei *Meta Object Facility* [MOF02] genutzt, um immer komplexere Modelle mittels eines einfacheren Modells zu beschreiben. Am Beginn dieser Kette steht ein minimales Modell, welches sich selber beschreibt. Meist werden die verschiedenen Modelle bzw. Meta-Modelle mit M1 (Modell), M2 (Meta-Modell) und M3 (Meta-Meta-Modell) bezeichnet. Bei der UML wird z.B. die Menge aller UML-Modelle (M1) wieder in UML mit einem eingeschränkten Sprachschatz (M2) beschrieben. Dies ist möglich, da für die Beschreibung der UML-Modelle nicht alle UML-Konstrukte benötigt werden. Für das Meta-Modell M2 wird auf die gleiche Weise ein Meta-Meta-Modell M3 gebildet. Im Fall der UML ist das Meta-Meta-Modell M3 so definiert, das es gerade ausreicht sich selber zu beschreiben. Somit ist das M3 sein eigenes Meta-Modell.

Eine weit verbreitete Technik zur Beschreibung von zeichenkettenorientierten Sprachen ist die Angabe einer Grammatik z.B. als Backus-Naur-Form (BNF) bzw. als erweiterte BNF (EBNF [Int96]). Eine BNF ist eine Darstellung einer kontextfreien Grammatik:

Definition 2 (kontextfreie Grammatik [HMU02])
Eine kontextfreie Grammatik *ist ein 4-Tupel* $(NTerm, Term, Prod, S_0)$ *mit*

1. $NTerm$ *ist eine Menge von Nichtterminalen,*
2. $Term$ *ist eine Menge von Terminalen für die* $Term \cap NTerm = \varnothing$ *gilt,*
3. $Prod : NTerm \to (NTerm \cup Term)^*$ *sind Produktionsregeln und*
4. $S_0 \in NTerm$ *ist das Startsymbol.*

Eine Grammatik, und somit auch eine BNF, erzeugt eine formale Sprache L. Ein Element der Sprache L, in der Sprachtheorie Wort genannt, wird durch die Anwendung von Produktionen aus dem Startsymbol abgeleitet. Ein Wort darf keine Nichtterminale enthalten und wird durch die Regeln abgeleitet.

Definition 3 (Ableitung einer formalen Sprache [HMU02])
Sei $G = (NTerm, Term, Prod, S_0)$ *eine kontextfreie Grammatik.*

1. *Die binäre Relation* $\rightharpoonup_G : (NTerm \cup Term)^* \times (NTerm \cup Term)^*$ *mit*

   $x \rightharpoonup_G y :\Leftrightarrow \exists\, u, v, p, q \in (NTerm \cup Term)^* : x = u^\frown p^\frown v \wedge p \rightharpoonup q \in Prod \wedge y = u^\frown q^\frown v$

   heißt die Ableitungsrelation von G.
2. *Die Relation* $\rightsquigarrow_G$ *ist die reflexive transitive Hülle von* $\rightharpoonup_G$.
3. *Wenn* $x \rightsquigarrow_G y$ *für eine Sequenz r von Regelanwendungen* $\rightharpoonup_G$ *gilt, wird dies* $x \overset{r}{\rightsquigarrow}_G y$ *geschrieben.*
4. *Die formale Sprache L, die von G erzeugt wird, ist definiert als* $L = \{w \in Term^* \mid S_0 \rightsquigarrow_G w\}$.

Bemerkung 1 *Die Konkatenation* $^\frown$ *fügt Zeichenfolgen hintereinander (vgl. Anhang Definition 47).*

Verwendung	Zeichen
Definition	=
Alternative	|
Option	[...] oder (...)?
Optionale Wiederholung	(...)*
Gruppierung	(...)
Anführungszeichen, 1. Variante	"... "
Anführungszeichen, 2. Variante	' ... '
Kommentar	(* ... *)
Spezielle Sequenz	? ... ?
Ausnahme	-

Abbildung 2.2: Zeichen zur Nutzung in einer EBNF nach [Int96]

Eine EBNF besteht aus Produktionsregeln (vgl. Abbildung 2.2), bei denen links der Produktionsname steht und nach einem Gleichheitszeichen „=“ wie die Produktion definiert ist. In den Produktionen können Sequenzen aus Terminalen (Zeichenketten) und Nicht-Terminalen (Produktionsnamen), optionale Bereiche ([...]) und Wiederholungen genutzt werden. Eine Liste der Möglichkeiten ist in Abbildung 2.2 zusammengefasst. Im Gegensatz zum Standard wird in dieser Arbeit eine zusätzliche Kennzeichnung von Nichtterminalen benutzt. Nichtterminale, die in spitzen Klammern eingeschlossen sind, dürfen nicht durch Leerräume (Whitespace) getrennt werden. Durch diese Konvention werden die Grammatiken lesbarer, da die Leerräume nicht separat in der BNF angegeben werden müssen. In den Beispielen in dieser Arbeit wird dies z.B. bei Variablennamen genutzt; Ein Variablenname muss eine zusammenhängende Zeichenkette sein und darf nicht durch Leerzeichen unterbrochen werden.

In Listing 2.1 ist die EBNF einer kleinen turingvollständigen Sprache namens FWHILE angegeben. FWHILE ist die aus der Berechenbarkeitstheorie bekannte Sprache WHILE [AS88], welche um einen einfachen Funktionsbegriff erweitert wurde. FWHILE wird in dieser Arbeit für die Einführung der Sprachfamilie Re$\mathcal{L}$ genutzt werden. Jede Instanz der Familie wird aus der BNF der Sprache, auf dem die Refactorings ausgeführt werden sollen, abgeleitet. Für die Einführung und die technische Beschreibung ist es nützlich, eine Sprache vorliegen zu haben, in der aussagekräftige Refactorings durchführt werden können und deren BNF klein ist.

In Listing 2.2 ist ein einfaches FWHILE Programm gegeben, welches die Formel $result = value1 * value2 - 3$ falls $value1$ positiv ist, berechnet. Anderenfalls ergibt das Programm $result = value1 - 3$. Die genaue Funktion des Programms wird im Abschnitt 2.4 beschrieben. Das Programm wird als Beispiel für die Durchführung eines Refactorings dienen. Dieses Programm lässt sich durch die Produktionsregeln aus der BNF erzeugen. Umgekehrt können für ein gegebenes Programm die Regeln bestimmt werden, die zu diesem Programm führen.

Die Darstellung der Anwendung der Regeln führt zu einen *Syntax Tree*. Darin sind die Blätter die Terminale und die inneren Knoten die Produktionsanwendungen (Nichtterminale) enthalten. In einem Syntax Tree befinden sich oft innere Knoten, die nur ein Kind haben. Da diese Knoten keine wichtigen Information beinhalten, werden diese meist eliminiert und falls nötig die Informationen in den anderen Knoten gehalten. Diese Umformung

```
FWHILE  = MAIN (FUNC)*                                                          1
MAIN    = "main" (MSTAT)* "endmain"
FUNC    = "func" FNAME "(" PARAM ")" (STAT)* "endfunc"
MSTAT   = ("SKIP"
          | VAR ":=" ITERM ";"
          | "if" BTERM "then" (STAT)* ("else" (STAT)*)? "fi"                    6
          | "while" BTERM "do" (STAT)* "od")
STAT    = (MSTAT | "return" (ITERM | BTERM)";")
FNAME   = IDENT
PARAM   = (VAR ("," VAR)*)?
BTERM   = (ITERM RELSYM ITERM)| FUSE                                            11
ITERM   = (VAR | CONST| FUSE)
                  (("+" | "-") (VAR | CONST| FUSE))*
FUSE    = FNAME "(" PARAM ")"
VAR     = IDENT
IDENT   = <LETTER> (<LETTER> | <DIGIT>)*                                        16
CONST   = ("+" | "-")? <DIGIT> (<DIGIT>)*
<LETTER>= ["a"-"z","A"-"Z"]
<DIGIT> = ["0"-"9"]
RELSYM  =  "=" | "!=" | "<" | ">"
```

Listing 2.1: BNF von FWHILE

```
main
  result := 0;
  if value2 < 0 then
      value2 := 1;
  fi                                                                            5
  while value2 != 0 do
      result := result + value1;
        value2 := value2 - 1;
    od
  result := result - 3;                                                         10
endmain
```

Listing 2.2: Ein FWHILE-Programm, welches $result = value1 * value2 - 3$ für $value1 > 0$ berechnet

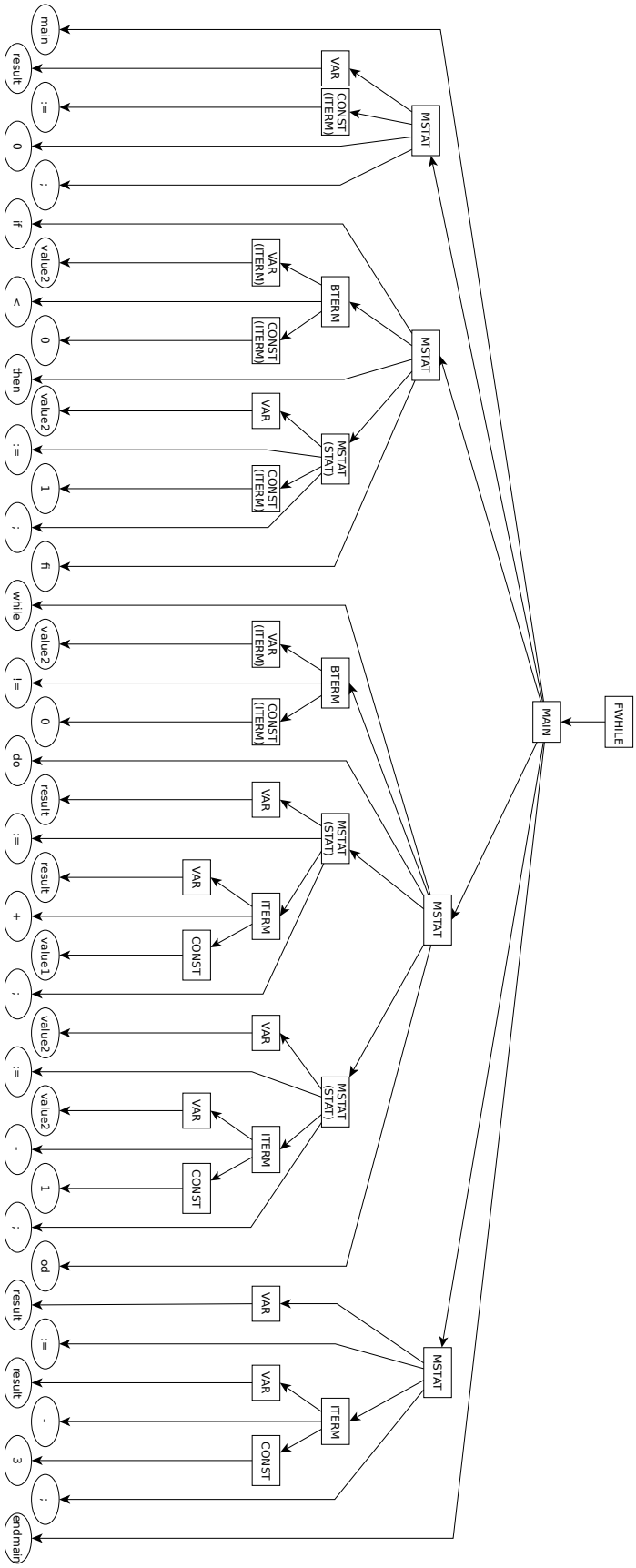


Abbildung 2.3: Abstract Syntax Tree für das Programm aus Listing 2.2

führt zu dem *Abstract Syntax Tree* (AST). Ein AST ist häufig die Ausgabe eines Parsers im Compilerbau. Der zu dem Programm in Listing 2.2 gehörende AST ist in Abbildung 2.3 gezeigt. Die Terminale sind als Ovale gezeichnet. Die Nichtterminale (Rechtecke) entsprechen den Produktionsregeln aus der BNF. Die entsprechenden Namen der Regeln sind in die Rechtecke geschrieben. Im Falle eines zusammengefassten Knotens ist auch der Name der nicht eingetragenen Produktion in Klammern angegeben.

Die Syntax-Beschreibung stellt FWHILE als eine Folge von Zeichen dar, die ohne eine Semantik keine Bedeutung hat. Im Vorgriff auf die genaue Erklärung im Zusammenhang mit der Semantik, soll hier kurz die Struktur der Sprache angedeutet werden. Die entsprechenden Produktionsnamen werden in Klammern angegeben und sind in Listing 2.1 zu finden.

Ein FWHILE Programm (FWHILE) besteht aus einem Hauptprogramm (MAIN) und einer evtl. leeren Menge von Funktionen (FUNC). Im Hauptprogramm können verschiedene Anweisungen (MSTAT) genutzt werden, die im wesentlichen den Konstrukten aus anderen Programmiersprachen entsprechen: Variablenzuweisungen, If/Then/Else und While-Schleifen. In Funktionen ist zusätzlich zu den Anweisungen des Hauptprogramms die Benutzung eines Return-Statements möglich (STAT).

2.2 Wohlgeformtheit

Die formale Sprache beschreibt die Struktur einer Sprache. Dies reicht nicht aus, da eine Sprache oft weitere Bedingungen erfüllen muss, damit dem Stück Sprache eine Bedeutung gegeben werden kann. So ist es in der deutschen Sprache notwendig, dass bestimmten Verben ein Zusatz hinzugefügt werden muss. Beispielsweise ist dem Wort „ist“ immer eine Eigenschaft zuzuordnen, während dies für andere Verben nicht notwendig bzw. verboten ist (z.B. „schläft“).
In FWHILE müssen zusätzlich zur Syntax folgende Regeln erfüllt sein:

1. Jede genutzte Funktion muss definiert sein (unter Beachtung der Parametermenge).
2. Funktionsnamen müssen eindeutig sein.
3. Jede Funktion muss immer einen Rückgabewert liefern.

Ein Problem stellt die Beschreibung solcher Eigenschaften dar. Oft sind sie natürlichsprachlich leicht zu beschreiben und werden häufig algorithmisch angegeben. Aus diesem Grund wird im Compilerbau die Wohlgeformtheit (statische Semantik) meist algorithmisch geprüft. So ist es z.B. bei FWHILE leicht, die Funktionsaufrufe mit den Funktionsdefinitionen zu vergleichen.

In dieser Arbeit wird die Wohlgeformtheit in formaler Form benötigt, um die Wohlgeformtheit eines refaktorisierten Codes zu prüfen. Die Wohlgeformtheitsbedingungen werden hier mit Hilfe der Prädikatenlogik ausgedrückt. In den Prädikaten werden neben Mengenoperatoren sogenannte *Analysefunktionen* benutzt. Diese Funktionen nehmen als Parameter ein Stück des Codes und berechnen daraus benötigte Informationen.

Ein Beispiel für eine solche Funktion ist definedFunc. Sie bestimmt die definierten Funktionen mit der Anzahl der Parameter. Die Anzahl der Parameter wird genutzt, da es in FWHILE nur einen Datentyp gibt und somit nicht der Typ der Parameter betrachtet werden muss.

Die *Analysefunktionen* werden in dieser Dissertation in weiteren Anwendungen genutzt werden. Das Thema der Definition und Auswahl solcher Funktionen wird in Kap. 6.3 behandelt.

Um die Wohlgeformtheitsbedingungen für FWHILE auszudrücken, müssen die oben genannten Bedingungen umgesetzt werden.

Die erste Wohlgeformtheitsbedingung stellt sicher, dass jede benutzte Funktion definiert ist. Das heißt, dass es zu jedem Aufruf einer Funktion eine entsprechende Definition mit einer passenden Signatur gibt. Bei FWHILE-Funktionen besteht die Signatur aus dem Namen der Funktion und der Anzahl der Parameter. Für die Formulierung wird die Signatur als Paar aus dem Namen und der Anzahl der Parameter zusammengesetzt. Eine Funktion „Sum“, welche zwei Parameter erwartet und die Summe dieser Parameter zurück gibt, hat die Signatur $(\text{Sum}, 2)$. Für die formale Beschreibung der Bedingung werden zwei Analysefunktionen benötigt. Die eine bestimmt die Menge aller aufgerufenen Funktionen mit der Anzahl der Parameter (usedFunc). Die zweite bestimmt die Multimenge (vgl. Anhang A.1) der definierten Funktionen (definedFunc), dass heißt mehrfach definierte Funktionen finden sich auch mehrfach in der Menge wieder. Die Funktion *set* wandelt eine Multimenge in eine normale Menge um, dass heißt die Funktion entfernt alle Dopplungen aus der Multimenge. Der Rückgabewert der Funktionen sind die entsprechenden Signaturmengen.

$$\text{usedFunc}(P) \subseteq \mathit{set}(\text{definedFunc}(P)) \text{ für } P \text{ ist FWHILE-Programm} \qquad \text{(FWHILE:W1)}$$

Die zweite Bedingung fordert, dass jede Funktion nur einmal definiert ist. Dies wird mit der Analysefunktion definedFunc aus der ersten Bedingung ausgedrückt. Da definedFunc eine Multimenge zurückliefert, ist eine mehrfach definierte Funktion auch mehrfach enthalten. Die Funktion *set* berechnet aus einer Multimenge eine normale Menge, das heißt jedes Element ist nur einmal vorhanden. Somit drückt die folgende Formel aus, dass, wenn die Multimenge der definierten Funktionen gleich der Menge der definierten Funktionen ist, die zweite Bedingung erfüllt ist. Die Gleichheit impliziert durch die Funktion *set*, dass jede Funktion nur einmal vorhanden ist.

$$\mathit{set}(\text{definedFunc}(P)) = \text{definedFunc}(P) \text{ für } P \text{ ist FWHILE-Programm} \qquad \text{(FWHILE:W2)}$$

Die dritte Wohlgeformtheitsbedingung für FWHILE stellt sicher, dass jede Funktion in jedem Fall einen Rückgabewert ausgibt. Das heißt, dass bei jedem Durchlauf der Funktion, der terminiert, irgendwann ein `return` ausgeführt werden muss. Dies ist notwendig, weil Funktionen in FWHILE nur in BTERM bzw. in ITERM genutzt werden. Nur wenn es einen Rückgabewert gibt, kann dieser nach der Benutzung der Funktion genutzt werden.

Dazu wird hier eine Analysefunktion calcReturn definiert, die bestimmt, ob eine Funktion einen Returnwert auf allen terminierenden Durchläufen besitzt. Diese Funktion baut direkt auf einer Typing-Funktion auf. Da Analysefunktionen so definiert sein sollen, dass sie mehrere Einsatzzwecke erfüllen kann, gibt diese den Rückgabetyp der Funktion zurück oder einen Fehlerwert *ERR*, falls entweder kein eindeutiger Typ bestimmt werden kann oder es Abläufe gibt, die ohne Return terminieren.
Als dritte Regel für die Wohlgeformtheit ergibt sich:

$$\forall f : \text{definedFunc}(P) \bullet \text{calcReturn}(f) \neq \mathit{ERR} \qquad \text{(FWHILE:W3)}$$

Damit sind die drei Wohlgeformtheitsbedingungen für FWHILE definiert.

2.3 Typing

Eine spezielle Art von Wohlgeformtheit ist die Wohlgetyptheit. Sie beschreibt, dass in einem Ausdruck bzw. einem Programm immer die passende Typen bei Aufrufen, Zuweisungen und Berechnungen genutzt werden. In FWHILE tritt das Typproblem nur bei der Nutzung von Funktionen auf, da alle Variablen den Typ Integer haben. Eine Funktion kann zusätzlich einen Wahrheitswert zurückgeben, der in `if` bzw. `while` Anweisungen genutzt wird. Die Rückgabe eines Zahlenwerts kann in Berechnung und Variablenzuweisungen genutzt werden.

Der Grund, warum die Wohlgetyptheit meist separat betrachtet wird, liegt in der Komplexität von Typsystemen und den damit verbundenen speziellen Techniken.

In FWHILE gibt es keine expliziten Typen. Die Variablen sind immer implizit vom Typ Integer. Wenn die Prädikate in den Kontrollstrukturen betrachtet werden, beschreiben diese Wahrheitswerte. Da in beiden Arten von Ausdrücken Funktionen verwendet werden können, müssen Funktionen Werte dieser beiden Typen zurückgeben können. Dabei muss der Typ einer Funktion in einem Prädikat (BTERM) ein Wahrheitswert sein und in Berechnungen (ITERM) ein Integer. Im Folgenden wird für Integer die Bezeichnung **int** sowie für Wahrheitswerte die Bezeichnung **bool** verwendet.

Die Bestimmung eines Typs eines Ausdruckes wird *Typing* genannt. Wenn dies in einer sinnvollen Art und Weise geschehen kann, ist ein Ausdruck oder Programm *wohlgetypt*. Ein Typing wird häufig durch Typableitungsregeln definiert. Ein Programm ist *wohlgetypt*, wenn es eine gültige Typableitung gibt. Das Typing hängt meist nicht nur von einzelnen Teilen des Programms ab, sondern nutzt verteilte Informationen, wie den Typ einer Variablen. Diese verteilten Informationen werden meist in einem Typmodell Σ dargestellt. Dort können neben der Typzuordnung für die Variablen unter anderem lokale Sichtbarkeiten der Variablen bzw. Typhierarchien definiert werden.

In FWHILE sind die Funktionen der entscheidende Teil für die Typprüfung. Durch die Wohlgeformtheit ist sichergestellt, dass eine benutzte Funktion definiert ist und einen Rückgabewert liefert. Hier wird gefordert, dass die Funktion einen Wert vom benötigten Typ zurück gibt. Dazu wird in Σ der Typ der Funktionen gehalten. Für FWHILE ist daher eine einfache Struktur ausreichend, wobei FNAME die Menge der Funktionsnamen ist:

$$\Sigma : \mathit{FNAME} \rightarrow \{\mathbf{int}\ , \mathbf{bool}\ \}$$

Wenn ein Ausdruck e (Expression) von einem bestimmten Typ τ sein soll, wird dies als $e \begin{smallmatrix}\circ\\ \circ\end{smallmatrix} \tau$ geschrieben. Wenn sich ein solcher Typausdruck aus einem Typmodell herleiten lässt, wird das als

$$\Sigma \vdash^{\tau} e \begin{smallmatrix}\circ\\ \circ\end{smallmatrix} \tau$$

ausgedrückt.

Die Typableitung wird durch Ableitungsregeln definiert. Eine solche Regel ist die Regel für das Return-Statement.

$$\frac{\Sigma \vdash^{\tau} e \begin{smallmatrix}\circ\\ \circ\end{smallmatrix} \tau}{\Sigma \vdash^{\tau} \texttt{return}\ e\ \texttt{;} \begin{smallmatrix}\circ\\ \circ\end{smallmatrix} \tau}$$

Oben auf der Linie steht die Vorbedingung, die erfüllt sein muss. Wenn mehrere Vorbedingungen erfüllt sein müssen, stehen diese durch Abstand getrennt hintereinander oder

übereinander. Unter der Linie ist die Schlussfolgerung zu finden. Etwaige Nebenbedingungen werden in Klammern neben die Regel geschrieben. Die Return-Statement-Regel besagt, dass der Typ des Return-Ausdruckes (τ) der Typ des Ausdruckes innerhalb des Return ist.

In den Bedingungen für die Wohlgetyptheit wird im Folgenden eine andere Darstellung des Typing verwendet. Die semantische Typfunktion $[\![]\!]^\tau$ ordnet jedem Ausdruck einen Typ zu. Diese lässt sich mithilfe der Typableitung definieren. Wenn ein Typ eines Ausdruckes durch die Ableitungsfunktion $\vdash^\tau$ unter Annahme eines Typmodelles Σ bestimmt werden kann, so definiert ergibt sich der Typ eines Ausdruckes als:

$$\Sigma[\![e]\!]^\tau = \tau :\Leftrightarrow \Sigma \vdash^\tau e \ \circ_\circ \ \tau$$

Wenn klar aus dem Kontext hervorgeht, welches Σ gemeint ist, wird es oft weggelassen. In Fällen, wie dem Return-Statement oben, kann diese Typfunktion direkt rekursiv definiert werden:

$$\Sigma[\![\texttt{return}\ \ e\ \ \texttt{;}]\!]^\tau := \Sigma[\![e]\!]^\tau$$

Die Typfunktion $[\![]\!]^\tau$ gibt für einfache Typsysteme einfache Darstellungen. Bei komplizierten Typsystemen werden viele Fallunterscheidungen benötigt. Dann ist die Angabe via Typableitung die geeignete Darstellung.

In dieser Arbeit werden beide Verfahren genutzt werden. Für FWHILE ist die direkte Definition ausreichend und somit einfacher zu verstehen. Für CSP-OZ, einer Sprache zur formalen Modellierung (vgl. Kapitel 3), ist dies nicht der Fall. Wegen der komplizierten Typstruktur, die u.a. durch Klassenhierarchien gegeben wird, sollte hier die Definition über die Ableitung genutzt werden. Die gesamte Definition der Typfunktion von FWHILE findet sich im Anhang B.2.

Definition 4 (Wohlgetyptheit) *Ein Programm ist* wohlgetypt*, wenn es für das Programm eine gültige Typableitung existiert.*

2.4 Semantik

Modelle oder Spezifikationen werden in der Informatik genutzt, um Programme zu beschreiben. Für solchen Beschreibungssysteme ist ein gemeinsames Verständnis der Bedeutung notwendig. Die Bedeutung von Formalismen (Programmen, Modellen) wird durch deren Semantik beschrieben. Die Bedeutung oder Semantik einer Sprache ist im Zusammenhang mit Refactorings in zweierlei Hinsicht wichtig. Zum Einen sollen Refactorings verhaltenserhaltend sein, aber um diesen Begriff definieren zu können, muss das Verhalten bekannt sein. Dies wird mit der Semantik beschrieben. Zum Anderen wird in dieser Dissertation auch eine Refactoringbeschreibungssprache definiert, für die es wichtig ist zu wissen, was die Sprache bedeutet und wie die Refactorings ausgeführt werden.

Dem Begriff der Semantik kann man sich aus verschiedenen Richtungen nähern, wie aus Sicht der Linguistik, der Philosophie oder der Mathematik. Hier soll nur ein formaler Semantikbegriff (vgl. [HR04]) betrachtet werden, der alle philosophischen Probleme ausklammert:

Definition 5 (Semantik) *Eine Semantik einer Sprache* $P = (A, L, D, [\![.]\!])$*, ist das Paar* $(D, [\![.]\!])_L$*, wobei* D *die* semantische Domäne *und* $[\![.]\!] : L \to D$ *das* semantische Mapping *ist.*

Die semantische Domäne ist der Formalismus, welcher genutzt wird, um die Bedeutung zu beschreiben. Für die semantische Domäne können sowohl formale oder mathematische Strukturen, aber auch natürlichsprachliche Beschreibungen, genutzt werden. Wichtig ist, dass die Bedeutung aus der semantischen Domäne unmehrdeutig ersichtlich ist. Eine semantische Domäne kann (und hat auch meistens) selbst wieder eine Semantik. Somit ist die Semantik der Semantik auch eine Semantik der Ausgangssprache L. Das hieraus entstehende philosophisch-logische Problem der Semantikkette soll hier nicht behandelt werden. Das semantische Mapping legt fest, wie die Sprache, bzw. wohlgeformten Teile der Sprache, in der semantischen Domäne abgebildet werden. Somit legt das Mapping fest, welche Bedeutung die Sprache hat. Aus diesem Grund wird das semantische Mapping selber oft einfach nur Semantik genannt. Da im semantischen Mapping die semantische Domäne implizit definiert ist, ist diese Doppelbezeichnung konsistent und wird auch in dieser Dissertation in beiden Bedeutungen genutzt.

Die Definition einer Semantik, bzw. genauer des semantischen Mapping, kann auf verschiedene Weise erfolgen: denotationelle Semantik (das Mapping ist eine Funktion), operationelle Semantik (die Domäne ist ein Transitionssystem, das Mapping ist indirekt durch die Übergangsregeln angegeben) oder axiomatische Semantik (die Domäne ist durch logische Eigenschaften charakterisiert, jedoch meist nicht vollständig; das Mapping ist nicht eindeutig). Bei den hier betrachteten Methoden ist die Art der Semantik unwesentlich, solange sie einige hier benötigte Bedingungen erfüllt. Daher soll hier auf eine detaillierte Darstellung verzichtet werden. Mehr Informationen befinden sich in der Literatur zur Sprachtheorie [AS88]. Häufig wird das semantische Mapping in mehrere Funktionen geteilt. Dies hat praktische Gründe, weil dadurch die Zuordnung besser zu verstehen ist. Das Vorgehen entspricht dennoch der Semantikdefinition, da aus den einzelnen Funktionen immer eine entsprechende Funktion gebildet werden kann. Teilfunktionen einer Semantik werden in dieser Arbeit als Semantikfunktion mit einem Superscript (Hochstellung) gekennzeichnet.

Als erstes Beispiel soll ein Teil der FWHILE Sprache betrachtet werden und dafür eine semantische Domäne gefunden werden. Die Integer-Terme (ITERM) beschreiben eine vereinfachte Integer-Arithmetik. Es bietet sich daher an, die ganzen Zahlen ($\mathbb{Z}$) als Domäne zu wählen. Auf den ersten Blick scheint dies unsinnig, weil FWHILE schon mit Zahlen arbeitet. Das ist so nicht korrekt. In FWHILE sind die Zahlen zunächst beliebige Zeichenketten. Es ist nebensächlich, dass sie zufällig wie ganze Zahlen aussehen. Wenn den Ziffernfolgen die intuitiv erwartete Bedeutung von Zahlen gegeben werden soll, muss dies erst definiert werden. Dazu wird eine Zuordnungsfunktion (Mapping) definiert, die die entsprechende Bedeutung herstellt. Dazu wird eine Funktion $[\![.]\!]^I$ benötigt, welche jedem CONST eine Zahl aus den Ganzen Zahlen zuordnet. Damit kann der Wert von $\mathsf{17}$ ermittelt werden:

$$[\![\mathsf{17}]\!]^I = 1 \cdot 10^1 + 7 \cdot 10^0 = 17$$

In dieser Gleichung wird die 17 auf zwei unterschiedliche Weisen genutzt, welches durch die unterschiedlichen Schriftarten angedeutet ist. Die $\mathsf{17}$ in Schreibmaschinenschrift innerhalb der Semantik-Klammern steht für die Zeichenfolge $\mathsf{1}$ gefolgt von $\mathsf{7}$ wie sie in der BNF von FWHILE definiert ist. Die 17 in proportionaler Schriftart steht für die ganze Zahl. Der Zeichenfolge der Zahl wird die Zahl selber zugeordnet. Für weitere Arten von Integerausdrücken wird die selbe Funktion verwendet. So kann die Addition zweier Konstanten rekursiv definiert werden:

$$[\![\texttt{<CONST1> "+" <CONST2>}]\!]^I = [\![\texttt{<CONST1>}^I]\!] + [\![\texttt{<CONST2> }]\!]^I$$

Dabei wird das Zeichen + auf die Addition innerhalb der ganzen Zahlen abgebildet.

Bevor die Semantik von FWHILE betrachtet wird, sollen einige in dieser Arbeit benötigte Eigenschaften angegeben werden, die hier von Semantiken gefordert werden.

In dieser Dissertation wird oft von einer *formalen Semantik* gesprochen. Leider hat sich noch keine einheitliche Definition durchgesetzt, die sagt, wann eine Semantik formal ist. Hier soll nicht versucht werden eine Definition anzugeben, deshalb wird hier nur das charakterisiert, was für diese Dissertation wichtig ist:

Definition 6 *Im Weiteren wird eine Semantik* formale Semantik *genannt, wenn*

- *die Domäne ein mathematischer Formalismus oder eine Formale Methode ist und*
- *sich auf der Domäne formale Beweise führen lassen.*

Die oben angedeutete Semantik für die Integer-Ausdrücke ist somit eine formale Semantik, da sie eine mathematische Methode benutzt (die ganzen Zahlen), in denen Beweise geführt werden können.

In dieser Arbeit wird eine spezielle Klasse von Semantiken noch eine wichtige Rolle spielen: Die kompositionellen Semantiken.

Definition 7 (kompositionelle Semantik) *Eine Semantik $(D, [\![.]\!])_L$ einer Sprache $P = (A, L, D, [\![.]\!])$ ist* kompositionell*, wenn sich die Semantik der Ausdrücke in der semantischen Domäne D mithilfe des semantischen Mapping $[\![.]\!]$ seiner Teilausdrücke darstellen lässt. Für einen Ausdruck $a_1 \frown a_2 \frown \ldots \frown a_i$ aus der formalen Sprache L, wobei a_j ($j \in \{1, \ldots, i\}$) Teilausdrücke sind, existiert eine Funktion $f : D^i \to D$, für die*

$$[\![a_1 \frown a_2 \frown \ldots \frown a_i]\!] = f([\![a_1]\!], [\![a_2]\!], \ldots, [\![a_i]\!])$$

gilt.

Bemerkung 2 *Die Operation $\frown$ bezeichnet die Konkatenation. Vgl. Definition 47 im Anhang.*

Ein solcher Ausdruck einer kompositionellen Semantik ist das semantische Mapping der Addition oben:

$$[\![\texttt{<CONST1> "+" <CONST2>}]\!]^I = [\![\texttt{<CONST1>}]\!]^I + [\![\texttt{<CONST2> }]\!]^I$$

Der Ausdruck links wird durch zwei Ausdrücke rechts dargestellt, die mittels einer Operation innerhalb der semantischen Domäne verknüpft wurden; hier ist das die Addition.

Die letzte Eigenschaft beschreibt, wann eine Transformation semantikerhaltend ist. Der Begriff ist eine Kurzform für semantisch äquivalent:

Definition 8 (Semantikerhaltende (In-Place-)Transformation) *Eine (In-Place-)Transformation $t : L \to L$ auf einer Sprache $P = (A, L, D, [\![.]\!])$ heißt* semantikerhaltend*, wenn*

$$\forall\, \lambda \in dom[\![.]\!] \bullet [\![t(\lambda)]\!] = [\![\lambda]\!]$$

gilt.

Bemerkung 3 *Es ist zu beachten, dass* $dom[\![.]\!] = L' \subseteq L$ *gilt.*

Nachdem der Begriff der Semantik eingeführt wurde, soll die Semantik von FWHILE angegeben werden.

2.5 FWHILE - Programme

In den vorherigen Abschnitten hat FWHILE als Beispiel für die Definition einer Sprache mittels BNF und für die Wohlgeformtheit gedient. In diesem Abschnitt soll die Sprachdefinition mit der Semantik vervollständigt werden und es sollen einige Eigenschaften von FWHILE und ein einfaches Programm betrachtet werden. Zum Verständnis der formalen Semantik ist es hilfreich, ein informales Verständnis der Sprache und deren Semantik zu besitzen.

FWHILE beschreibt einfache Berechnungen auf ganzen Zahlen. Daher sind alle Variablen implizit vom Typ Integer und müssen vor ihrer Benutzung nicht deklariert werden. Da FWHILE keine Eingabe und die Main-Funktion keine Parameter besitzt, wird das Programm mit einer initialen Belegung der Variablen gestartet, welche implizit die Eingabe ist. Gleichermaßen sieht es mit der Ausgabe aus. Die Belegung der Variablen nach der Ausführung ist das Ergebnis des FWHILE Programms. An Kontrollstrukturen besitzt FWHILE die bedingte Programmverzweigung „`if`“ und die kopfgesteuerte Schleife „`while`“. Beide stimmen mit den entsprechen Konzepten aus anderen Programmiersprachen überein. Funktionen in FWHILE haben immer einen Rückgabewert. Der Typ des Rückgabewertes wird implizit durch das Return-Statement bestimmt. Wenn in einer Funktion ein Return-Statement erreicht wird, dann wird die Ausführung der Funktion abgebrochen und in der aufrufenden Funktion oder in *main* fortgefahren. Als Parameter der Funktionen sind nur Variablen erlaubt. Es sind in den Parametern insbesondere keine Berechnungen und keine Konstanten möglich. Da es nur Integer-Variablen gibt, sind die Parameter immer vom Typ Integer.

Für die Semantik spielen drei Informationen eine zentrale Rolle: Die Variablenbelegung σ, der Rückgabewert r und die Menge der definierten Funktionen f. Die Variablenbelegung wird bei allen Nutzungen der Variablen benötigt. Darüber hinaus stellt die initiale Belegung die Eingabe des FWHILE-Programms dar. Auch die Ausgabe wird durch die Belegung bestimmt. Sie ist als Funktion des Variablennamens (VAR) auf seinen Wert definiert. Da FWHILE nur Integer-Variablen kennt, kann dies als Funktion in den ganzen Zahlen definiert werden:

$$\sigma : VAR \nrightarrow \mathbb{Z}$$

Die Startbelegung, welche die Eingabe repräsentiert, wird im Folgenden mit σ_I (I für Input) und die Ausgabe bzw. Endbelegung mit σ_O (O für Output) bezeichnet. Die semantische Domäne von FWHILE ist somit die Relation, die der Menge aller Startbelegungen eine Endbelegung zuordnet. Da die Eingabe vor der Ausführung des Programms bekannt ist, braucht das semantische Mapping nur die Endbelegungen zu berechnen. Für diese Definition werden zwei Hilfskonstrukte benötigt, die für den Aufruf von Operationen benötigt werden. So wird eine Funktion f benötigt, die einen Funktionsnamen auf die entsprechende Funktionsdefinition abbildet. Diese Definition beinhaltet den Code und die Namen der Parameter. Die Variable $r : \mathbb{Z} \cup \{\bot\}$ zeigt an, ob eine Funktion beendet wurde ($r \neq \bot$) und falls ja, welchen Rückgabewert die Funktion hat.

Um Formeln nicht mit Typinformationen zu belasten, werden bestimmte Variablennamen immer für denselben Typen verwenden. Für FWHILE sind es die folgenden:

$$\begin{aligned} bterm &: \texttt{BTERM} \\ fuse &: \texttt{FUSE} \\ main &: \texttt{MAIN} \\ fwhile &: \texttt{FWHILE} \\ stmt, stmt1, stmt2 &: \texttt{STAT} \\ stmtList, stmtList1, stmtList2 &: (\texttt{STAT})* \\ var &: \texttt{VAR} \\ iterm, iterm1, iterm2 &: \texttt{ITERM} \\ const &: \texttt{CONST} \\ func, funcList &: \texttt{FUNC} \end{aligned}$$

Die Definition der Semantik wird auf verschiedene Funktionen aufgeteilt. Sie definieren die Semantik der Teilausdrücke, wie beispielsweise $[\![.]\!]^I$ einem Integer-Term (ITERM) eine Bedeutung gibt. Die Funktion *value* stellt die Umwandlung einer Zahl in Form einer Zeichenkette in die entsprechende Zahl dar. In dieser Semantik wird die Typfunktion aus Kapitel 2.3 genutzt ($[\![.]\!]^\tau$). Da in den Integer-Termen (ITERM) Funktionen sowie Variablen genutzt werden können, werden die Funktion f und die Belegung σ benutzt. Beide sind von der Umgebung des ITERM abhängig und werden deshalb als Parameter den Funktionen übergeben.

Die Semantik von Integer-Termen ist eine Funktion aus den Codestücken, die der Produktion ITERM genügen, in die ganzen Zahlen:

$$[\![.]\!]^I(\sigma,f) : ITERM \times (VAR \to \mathbb{N}) \times ((NAME \times \mathbb{N}) \to (FUNC, PARAM)) \to \mathbb{Z}$$

Dabei ist $VAR \to \mathbb{N}$ der Typ der Belegung σ und $(NAME \times \mathbb{N}) \to (FUNC, PARAM)$ der Typ von f. Der Wert von $[\![.]\!]^I$ ist, wenn es sich um eine Konstante bzw. um eine Variable handelt, deren Wert. Diese Werte werden mit den übergebenen Funktionen in f berechnet. Wenn es eine Addition bzw. ein Subtraktion ist, ist es die entsprechende Operation auf den Werten der Teilausdrücke. Wenn es sich um eine Funktion handelt, deren Rückgabetyp ein Integer ist, ist es der Wert dieser Funktion. Da der Rückgabewert der Funktionssemantik (siehe unten $[\![.]\!]^{FU}$) ein Tripel ist, deren zweiter Wert das Ergebnis der Funktion ist, wird dieser mit der Funktion *second* aus dem Tripel bestimmt. Die Angabe, dass die Funktion einen Rückgabewert des Typ Integer hat, ist eigentlich überflüssig, da dies schon durch die Wohlgetyptheit sichergestellt ist. Zum besseren Verständnis wird es hier dennoch angegeben.

$$[\![iterm]\!]^I(\sigma,f) =$$

$$\begin{array}{lll} \sigma(var) & \text{für} & iterm = var \\ value(const) & \text{für} & iterm = const \\ [\![iterm1]\!]^I(\sigma,f) + [\![iterm2]\!]^I(\sigma,f) & \text{für} & iterm = iterm1 \text{ "+" } iterm2 \\ [\![iterm1]\!]^I(\sigma,f) - [\![iterm2]\!]^I(\sigma,f) & \text{für} & iterm = iterm1 \text{ "-" } iterm2 \\ [\![fuse]\!]^{FU}(\sigma,\bot,f) & \text{für} & iterm = fuse \text{ und } [\![fuse]\!]^\tau = \mathbf{int} \end{array}$$

Die Berechnung des Wertes einer Funktion wird durch die Funktion $[\![.]\!]^{FU}$ realisiert. Für

jeden Funktionsaufruf wird ein neues Binding σ_f erstellt. Dieses Binding enthält die Parameter mit seinen Werten und alle anderen Variablen, die in der Funktion genutzt werden.

$$\sigma_f = \{y_1 \mapsto 0, \ldots, y_n, \mapsto 0, param[1] \mapsto \sigma(parm[1]), \ldots, param[m] \mapsto \sigma(parm[m])\}$$

Die Variablen y_i werden im Code der Funktion genutzt, sind aber nicht als Parameter deklariert. Die Parameter *param* werden mit dem Wert des Bindings der Variablen, die aufgerufen werden belegt. Als Funktionsparameter ist in FWHILE nur die Nutzung von Variablen erlaubt. Daher braucht an dieser Stelle nur die Auswertung der Variablenwerte betrachtet zu werden. Die anschließende Auswertung funktioniert wie die Auswertung des Hauptprogramms, welches weiter unten erklärt ist.

$$[\![.]\!]^{FU} : FUSE \times (VAR \to \mathbb{N}) \times (NAME \times \mathbb{N}) \to (\mathbf{bool} \cup \mathbf{int})$$

$$[\![fuse]\!]^{FU}(\sigma, f) = second([\![first(f(fname, count(parm)))]\!](\sigma_f, \bot, f)) \quad \text{für } fuse = fame \texttt{ "(" } param \texttt{ ")"}$$

Die Funktion $[\![.]\!]^B$ ist die Auswertung eines Prädikates ($BTERM$). Die Definition der Semantik der Prädikate ist analog der Definition für die Integer-Terme. Die Funktion geht statt in die ganzen Zahlen in die Wahrheitswerte:

$$[\![.]\!]^B : BTERM \times (VAR \to \mathbb{N}) \times ((NAME \times \mathbb{N}) \to (FUNC, PARAM)) \to \{true, false\}$$

Entsprechend ist der Wert der Funktion das Ergebnis, der Vergleiche bzw. der Wert eines Funktionsaufrufs.

$$[\![bterm]\!]^B(\sigma, f) =$$

$$\begin{array}{lll}
second([\![fuse]\!]^{FU}(\sigma, \bot, f)) & \text{für} & bterm = fuse \\
 & & \text{und } [\![fuse]\!]^{\tau} = \mathbf{bool} \\
[\![iterm1]\!]^I(\sigma, f) = [\![iterm2]\!]^I(\sigma, f) & \text{für} & iterm1 \texttt{ "=" } iterm2 \\
[\![iterm1]\!]^I(\sigma, f) \neq [\![iterm2]\!]^I(\sigma, f) & \text{für} & iterm1 \texttt{ "!=" } iterm2 \\
[\![iterm1]\!]^I(\sigma, f) < [\![iterm2]\!]^I(\sigma, f) & \text{für} & iterm1 \texttt{ "<" } iterm2 \\
[\![iterm1]\!]^I(\sigma, f) > [\![iterm2]\!]^I(\sigma, f) & \text{für} & iterm1 \texttt{ ">" } iterm2
\end{array}$$

Nachdem die Semantik der Terme definiert ist, wird mit diesen die Semantik der Statements definiert. Dazu wird eine Funktion $[\![.]\!]^{STM}(\sigma, r, f)$ definiert, welche für Statements und Funktionsdefinitionen die Abbildung in die Semantik darstellt.

$$[\![.]\!]^{STM} : STAT \times INTZUS \to INTZUS$$

mit

$$INTZUS := (VAR \to \mathbb{N}) \times (\mathbf{int} \cup \mathbf{bool} \cup \{\bot\}) \times ((NAME \times \mathbb{N}) \to (FUNC, PARAM))$$

Zuerst soll die Funktion betrachtet werden, wenn der Rückgabewert einer Funktion nicht gesetzt ist ($r = \bot$). Dieser Fall deckt auch das Hauptprogramm ab.

$[\![stmt]\!]^{STM}(\sigma, \bot, f) =$

$[\![main]\!]^{STM}(\sigma, r, [\![funcList]\!]^F)$	falls	$stmt = main\ funcList$
$[\![stmtList]\!]^{STM} \circ [\![stmt1]\!]^{STM}(\sigma, r, f)$	falls	$stmt = stmt1\ stmtList$
$(\sigma, \bot, f)$	falls	$stmt =$ "SKIP"
$(\sigma \oplus (var \mapsto [\![iterm]\!]^I(\sigma, f)), r, f)$	falls	$stmt = var$ ":=" $iterm$ ";"
$[\![stmtList1]\!]^{STM}(\sigma, r, f)$	falls	$[\![bterm]\!]^B(\sigma, f) = true$ und $stmt =$ "if" $bterm$ "then" $stmtList1$ ("else" $stmtList2$)? "fi"
$(\sigma, \bot, f)$	falls	$[\![bterm]\!]^B(\sigma, f) = false$ und $stmt =$ "if" $bterm$ "then" $stmtList$ "fi"
$[\![stmtList2]\!]^{STM}(\sigma, r, f)$	falls	$[\![bterm]\!]^B(\sigma, f) = false$ und $stmt =$ "if" $bterm$ "then" $stmtList1$ "else" $stmtList2$ "fi"
$[\![stmt]\!]^{STM} \circ [\![stmtList]\!]^{STM}(\sigma, r, f)$	falls	$[\![bterm]\!]^B(\sigma, f) = true$ und $stmt =$ "while" $bterm$ "do" $(stmtList)*$ "od"
$(\sigma, \bot, f)$	falls	$[\![bterm]\!]^B(\sigma, f) = false$ und $stmt =$ "while" $btrem$ "do" $(stmtList)*$ "od"
$(\sigma, [\![bterm]\!]^B(\sigma, f), f)$	falls	$stmt =$ "return" $bterm$ ";"
$(\sigma, [\![iterm]\!]^I(\sigma, f), f)$	falls	$stmt =$ "return" $iterm$ ";"

Wenn der Rückgabewert einer Funktion r einen Integer-Wert enthält, sollen keine weiteren Statements der Funktion ausgeführt werden.

$$r \in \mathbb{Z} \Rightarrow [\![stmt]\!]^{STM}(\sigma, r, f) = (\sigma, r, f)$$

Einzelne interessante Fälle der Funktion sollen besonders betrachtet werden: Wenn die Funktion $[\![stmt]\!]^{STM}(\sigma, r, f)$ auf ein vollständiges FWHILE Programm angewandt wird, werden die Funktionen ($funcList$) in der Funktion f abgelegt und das FWHILE Programm ausgewertet:

$$[\![main]\!](\sigma, r, [\![funcList]\!]^F) \quad \text{falls} \quad stmt = main\ funcList$$

Für die Berechnung von f wird eine weitere Funktion $[\![.]\!]^F$ genutzt, die die Signaturen der Funktionen auf den Funktionsrumpf abbildet. Wie weiter oben erläutert, besteht die Signatur einer FWHILE-Funktion aus ihrem Namen und der Anzahl ihrer Parameter.

$$[\![funcList]\!]^F =$$

$[\![funcList1]\!]^F \oplus [\![func]\!]^F$	für	$funcList = funcList1\ \ func$
$\varnothing$	für	$funcList = \varnothing$
$((fname, count(param)) \mapsto (param, stmtList))$	für	$funcList =$ "func" $fname$ "(" $param$ ")" $stmtList$ "endfunc"

Die Fälle für die Return-Statements sind in die allgemeinen Statements integriert. Dies mag unrichtig erscheinen, da im Hauptprogramm kein Return-Statement genutzt werden darf. Aber weil dies schon durch die Syntax ausgeschlossen ist, stört es im Hauptprogramm nicht.
Die Semantik von FWHILE wird mittels der vorherigen Definitionen beschrieben als

$$[\![P]\!]_{FWHILE}(\sigma_I) = first([\![P]\!]^{STM}(\sigma_I, \bot, \varnothing))$$

Damit ist die formale Semantik von FWHILE gegeben. Bevor in den nächsten Kapiteln Modelle und Refactorings betrachtet werden, soll die Semantik noch an dem Beispiel-FWHILE-Programm aus Listing 2.2 vorgeführt werden. Im Folgenden wird der Programmtext abgekürzt, da die Darstellung sonst unübersichtlich wird. Die Punkte ... sind mit dem fehlenden Code aus Listing 2.2 auszufüllen.
Es soll die Semantik

$$[\![\texttt{main result := ... endmain}]\!]_{FWHILE}(\sigma_I)$$

berechnet werden. Für das Beispiel soll die Startbelegung

$$\sigma_I = \{value1 \mapsto 4;\ value2 \mapsto 2;\ result \mapsto 2;\ \}$$

gewählt werden. Nach den Regeln muss also

$$first([\![\texttt{main result := ... endmain}]\!]^{STM}(\sigma_I, \bot, \varnothing))$$

berechnet werden. Da keine Funktionen definiert sind, ist der innere Term gleich

$$\begin{aligned}
& [\![\texttt{main result := ... endmain}]\!]^{STM}(\sigma, \bot, \varnothing) \\
= \ & [\![\texttt{result := result - 3;}]\!]^{STM} \\
\circ \ & [\![\texttt{while value2 != 0 do result := result + value1; value2 := value2 -1;od}]\!]^{STM} \\
\circ \ & [\![\texttt{if value2 < 0 then value2 := 1; fi}]\!]^{STM} \\
\circ \ & [\![\texttt{result := 0; }]\!]^{STM}(\sigma, r, \varnothing)
\end{aligned}$$

Als erstes wird der Teilausdruck $[\![\texttt{result := 0; }]\!]^{STM}(\sigma, r, \varnothing)$ berechnet. Dabei wird in dem Binding σ der Wert der Variablen `result` auf 0 gesetzt. Das neue Binding σ_1 ist somit:

$$\sigma_1 = \sigma \oplus \{result \mapsto 0\} =$$

$$\{value1 \mapsto 4;\ value2 \mapsto 2;\ result \mapsto 0;\ \}$$

Da die anderen Werte unverändert bleiben, ergibt sich:

$$\begin{aligned}
& [\![\texttt{main result := ... endmain}]\!]^{STM}(\sigma, \bot, \varnothing) \\
= \ & [\![\texttt{result := result - 3;}]\!]^{STM} \\
\circ \ & [\![\texttt{while value2 != 0 do result := result + value1; value2 := value2 -1;od}]\!]^{STM} \\
\circ \ & [\![\texttt{if value2 < 0 then value2 := 1; fi}]\!]^{STM}(\sigma_1, \bot, \varnothing)
\end{aligned}$$

Da $value2 < 0$ nicht erfüllt ist, weil die Variablen $value2$ den Wert 2 hat ($\sigma_1(value2) = 2$) vereinfacht sich der Term wegen

(σ, r, f) falls $[\![bterm]\!]^B(\sigma, f) = false$ und $stmt =$ `"if"` $bterm$ `"then"` $stmtList$ `"fi"`

zu

$$\begin{aligned}
& [\![\texttt{main result := ... endmain}]\!]^{STM}(\sigma, \bot, \varnothing) \\
= \ & [\![\texttt{result := result - 3;}]\!]^{STM} \\
\circ \ & [\![\texttt{while value2 != 0 do result ... value2 := value2 - 1; od}]\!]^{STM}(\sigma_1, r, \varnothing)
\end{aligned}$$

Die folgenden 15 Regelanwendungen werden ausgelassen, weil sie den vorherigen gleichen. Das Ergebnis der Auswertung ergibt sich zu:

$$\begin{aligned} & [\![\texttt{main result := ... endmain}]\!]^{STM}(\sigma, \bot, \varnothing) \\ = \; & (\sigma_{13}, \bot, \varnothing) \end{aligned}$$

mit

$$\sigma_{13} = \{value1 \mapsto 4;\ value2 \mapsto 0;\ result \mapsto 5\}$$

Die Belegung σ_{13} ist das Ergebnis der Berechnung des FWHILE-Programms:

$$\begin{aligned} & [\![\texttt{main result := ... endmain}]\!]_{FWHILE}(\sigma_I) \\ = \; & \mathit{first}([\![\texttt{main result := ... endmain}]\!]^{STM}(\sigma_I, \bot, \varnothing)) \\ = \; & \sigma_{13} \end{aligned}$$

In diesem Kapitel wurde anhand des Beispiels FWHILE gezeigt, wie eine Sprache mittels Syntax, Wohlgeformtheit und Semantik gebildet wird. Diese grundlegenden Begriffe werden im Rest der Arbeit immer wieder auftreten. Die Syntax spielt insbesondere bei der Definition von Re$\mathcal{L}$, einer Sprachfamilie für die Refactoringbeschreibung, eine wichtige Rolle. Später im Kapitel 7.2 über die Verhaltenserhaltung von Refactorings wird diese Semantik die Grundlage für die dort vorgestellten Beweistechniken sein.

Kapitel 3

CSP-OZ: Eine formale Methode mit mehreren Sichten

In vielen Entwicklungsprozessen spielen Modelle eine wichtige Rolle. Ein Architekt kann anhand eines Modells sehen, ob ein Haus die Erwartungen des Auftraggebers erfüllt, bevor es errichtet wird und Änderungswünsche preiswert einplanen. Ein Flugzeugbauer entwirft verkleinerte Modelle eines Flugzeuges, um Flugeigenschaften in einem Windkanal zu testen. Beide Modelle spiegeln nicht die vollständige Wirklichkeit wider, sondern sind auf die jeweils relevanten Eigenschaften reduziert.

Ein Modell ist eine Abstraktion der Wirklichkeit, die nur die für den Zweck des Modells benötigten Informationen enthält. Nach [Sta73] wird ein Modell durch die drei Begriffe Abbildung, Vereinfachung und Pragmatismus beschrieben. Ein Modell ist immer eine Abbildung (Repräsentation) natürlicher oder künstlicher Originale, die selbst wieder Modelle sein können. Die Vereinfachung beschreibt, dass ein Modell nicht alle Attribute des Originals enthalten muss. Oft enthält es nur die Attribute, die in einem bestimmten Kontext nützlich bzw. notwendig erscheinen. Der Pragmatismus beschreibt die „Orientierung am Nützlichen“ [Sta73]. Dies beinhaltet, dass ein Modell nur unter bestimmten Voraussetzungen für das Original stehen kann. Dabei wird insbesondere die Abhängigkeit von Nutzen und Zeit hergestellt: Ein Planungsmodell dient als Anleitung für die Erstellung eines Gutes und ist nur in der Planung zu gebrauchen. Es enthält andere Informationen als ein Modell, welches zur Reparatur benötigt wird.

In dieser Arbeit ist die Abbildung zwischen der Wirklichkeit und dem Modell nebensächlich. Das Modell liegt als eine formal zu interpretierende Struktur vor, genauer gesagt als mathematisches Modell. Die Beschreibung dieser Modelle kann als eine Sprache aufgefasst werden und besteht somit aus einer Syntax und einer Semantik. Beispiele für Modellsprachen in der Informatik sind UML [UML], ER-Diagramme [Che76], Z [Spi92, WD97], Object-Z [Smi00], CSP-OZ [Fis00] und B [Abr96]. Eine spezielle Art von Modellen sind Softwaremodelle. Diese Modelle bilden eine zu erstellende oder eine existierende Software ab. Software wird mittels Programmiersprachen geschrieben. Ein Softwaremodell kann somit als Abstraktion des Programmcodes gesehen werden. Während der Programmcode vollständige Informationen über das Programm bzw. dessen Ablauf enthält, ist im Modell nur die für den jeweiligen Kontext wichtige Information gegeben. Im Verlauf des Model-driven Developments [SB88, OMG03] werden die Modelle verfeinert, d.h in neue Modelle überführt, die mehr Informationen enthalten, bis letztendlich der Code der Software erreicht ist.

Bei Modellen in der Informatik werden oft Teile eines Gesamtmodells genutzt, um bestimmte Sachverhalte genauer darzustellen, und dabei wiederum andere Teile ausgelassen. In der UML wird durch das Klassendiagramm die Klassenstruktur von Software beschrieben, das innere dynamische Verhalten einer Klasse wird dabei nicht dargestellt. Das dynamische Verhalten lässt sich mittels eines Statecharts beschreiben. Sowohl Klassendiagramme als auch Statecharts beschreiben *Sichten*, bei denen das Modell an sich unangetastet bleibt und nur ein Teil dargestellt wird. Es stellt sich die Frage, was in einem Modell, das aus mehreren Sichten besteht, das Gesamtmodell ist. In UML sind die Diagrammarten Sichten. Dabei gibt es keine Sicht, die das ganze Modell beschreibt. Es gibt zwei Möglichkeiten, das Verhältnis zwischen Gesamtmodell und Sichten zu beschreiben:

1. Es ist ein Modell gegeben und die Sichten sind vom Modell abhängig. Das heißt, wenn sich das Modell ändert, dann ändern sich die einzelnen Sichten ebenfalls (Ein-Modell-Interpretation).

2. Ein Modell wird durch die Übereinanderlegung von Sichten definiert. In diesem Fall müssen die Sichten zueinander passen, dass heißt sie dürfen keine Widersprüchlichkeiten beschreiben oder besser gesagt, sie müssen konsistent zueinander sein (Überlagerungs-Interpretation).

Beide Betrachtungsweisen haben ihre Vor- und Nachteile. Die vom Modell ausgehende Betrachtung hat einen Kern, das Modell, in welchem alles festgelegt ist. Die Sichten sind eine Art Fenster, die vom Modell abhängig sind. Probleme der Konsistenz oder der Widersprüchlichkeit der Sichten können hier nicht auftreten, da die Sichten durch das Modell spezifiziert sind. Der größte Nachteil ist, dass ein Gesamtmodell praktisch nie definiert wird, um darauf die Sichten zu betrachten. Ein Modellierer wird sich immer mit einer Sicht befassen und nie mit dem Modell als Ganzes. Dieser Nachteil ist gleich der Vorteil der zweiten Sichtweise. Sie entspricht der gängigen Denkweise eines Modellierers. Meist werden die verschiedenen Sichten getrennt betrachtet und modelliert. Das Modell entsteht erst aus den Überlagerung der einzelnen Sichten, die als spezifische Teilmodellierung betrachten werden können. Dafür muss hier sichergestellt werden, dass die Sichten zueinander konsistent sind.

Unter bestimmten Voraussetzungen gelten beide Betrachtungsweisen gleichzeitig. Wenn ein Modell aus konsistenten Sichten zusammengesetzt ist, sind beide Betrachtungsweisen gültig. In dieser Arbeit wird immer von diesem Zustand ausgegangen. Refactorings lassen sich auf einem ungültigen, d.h. inkonsistenten Modell kaum sinnvoll anwenden.

Sichten gibt es in verschiedenen Formen. In der UML kann jede Diagrammart als Sicht aufgefasst werden. Das Gesamtmodell, welches durch die Diagramme beschrieben wird, ist, wenn die Diagramme konsistent sind, durch diese gegeben. Sichten können zusammengefasst werden. In der Modellierungssprache CSP-OZ, die hier als Beispielsprache genutzt wird, werden die Ablaufsicht mittels *Communicating Sequential Prozesses* CSP (der CSP-Teil) und die Struktur-Sicht (Object-Z-Teil) in einer Klasse zusammengefasst. Dieses wird im Namen durch die Abkürzungen der beiden verwendeten Formalismen ausgedrückt. Obwohl sich hier die Sichten in einem Diagramm befinden, sind es echte Sichten.

3.1 Semantik in Modellen mit mehreren Sichten

Weil Modelle mit Sprachen beschrieben werden, gilt für die Semantik von Modellen im Grundsatz das Gleiche, wie für die Semantik von Sprachen. Bei Modellen mit mehreren Sichten gibt es ein paar Dinge, die besonderer Aufmerksamkeit bedürfen, da je nach

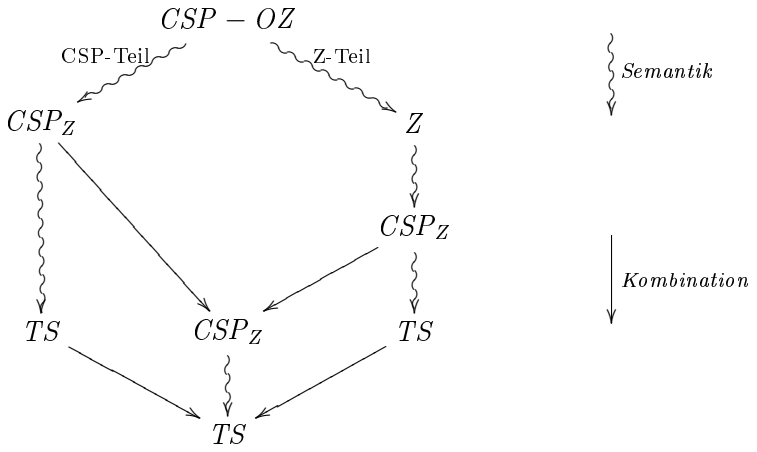


Abbildung 3.1: Struktur der Semantik von CSP-OZ: Ein Beispiel für eine Semantik mit verschiedenen Sichten: Ausgehend von CSP-OZ wird den Sichten getrennt eine lokale Semantik gegeben (CSP- bzw. Z-Teil). Damit die lokalen Semantiken zur globalen Semantik zusammengesetzt werden können, müssen sie angeglichen werden (CSP-Semantik von Z). Die lokalen CSP-Semantiken können kombiniert werden. Gleiches ist auf der Grundlage der Transitionssysteme (TS) möglich.

Betrachtungsweise (Sichten als Fenster oder Modell als Überlagerung der Sichten) die Semantik zusammengesetzt werden kann.

In der Einleitung dieses Kapitels wurden die Zusammenhänge von Sichten und Modellen beleuchtet. Diese spiegeln sich in der Interpretation von Sichten im Zusammenhang mit Semantiken wider. Einem Gesamtmodell soll nach der Ein-Modell-Interpretation genau eine Semantik zugewiesen werden. Im Gegensatz dazu ist es einfacher für jede Sicht eine Semantik anzugeben (Überlagerungs-Modell). Diese beiden Ansätze für die Semantikfindung lassen sich kombinieren, indem zur Semantik der Sichten eine gemeinsame Semantik des Modells erstellt wird. Dies ist möglich, da das semantische Mapping transitiv ist. Ein Beispiel dafür ist CSP-OZ [Fis00]. Eine CSP-OZ-Klasse hat zwei Sichten. Eine Z-Sicht, die sich an Object-Z [Smi00] anlehnt, und eine CSP-Sicht, die sich an CSP [Hoa85] anlehnt (vgl. Abbildung 3.1). Beide Sichten beleuchten einen anderen Teil des Modells. Jeder der Sichten wird im ersten Schritt eine Semantik zugeordnet. Damit hat das entsprechende Teilmodell eine eigene Semantik, die *lokale Semantik*. Die lokale Semantik des CSP-Teils hat als Domäne *CSP*, daher ist das semantische Mapping eine einfache Übertragung des Codes, ähnlich wie dies für die Zahlen mittels der Funktion $[\![.]\!]^I$ bei FWHILE durchgeführt wurde. Die lokale Semantik des Z-Teils ist etwas komplizierter: Hier wird erst die Object-Z Semantik angewandt, die als Ergebnis einen abstrakten Datentyp in Z hat. Die *globale Semantik* ist die Semantik des Gesamtmodells. Als globale Semantik wird CSP genutzt. Um die lokalen Semantiken zu der globalen Semantik zusammensetzen zu können, müssen die lokalen Semantiken zusammengefasst werden. Dazu wird im Fall von CSP-OZ der Z-Notation eine CSP-Semantik gegeben. Diese ist dann auch eine lokale Semantik des Z-Teils, da die Semantikbeziehung transitiv ist. Aus den lokalen Semantiken der Sichten wird die globale Semantik (bzw. eine globale Semantik *einer* CSP-OZ-Klasse) als Komposition der CSP-Semantik des CSP-Teils und der CSP-Semantik des Z-Teil genutzt. Die beiden Interpretationen des Modells finden sich hier wieder. Das Gesamtmodell hat eine Semantik

und jede Sicht hat eine Semantik. Diese Struktur wird sich später bei den Beweistechniken für Refactorings als nützlich erweisen.

In dem Diagramm von Abbildung 3.1 ist eine weitere wichtige Eigenschaft dargestellt: Die Zusammensetzung der lokalen Semantiken zur globalen Semantik kann direkt auf der Ebene von CSP durchgeführt werden, als auch auf der Ebene der darunterliegenden Transitionssysteme.

Umgekehrt kann aus der globalen Semantik der Modelle eine lokale Semantik für die Sichten ableitet werden. Es kann meist eine Projektion der globalen Semantik auf die für die Sicht relevanten Teile der semantischen Domäne gebildet werden.

Insgesamt wird in dieser Arbeit von einer Semantik für jede Sicht und einer Semantik für das Gesamtmodell ausgegangen.

3.2 Formalisierung von mehrsichtigen Modellen

Modelle werden in der Informatik mittels Sprachen beschrieben. Daher wird hier ein Modell mit einer Instanz einer Sprache gleichgesetzt, die zusätzlich eine Abbildung in die zu modellierende Welt besitzt. Wenn ein Modell als Sprachinstanz aufgefasst wird, kommt die Frage auf, wie sich ein mehrsichtiges Modell im Bezug auf die Sprache verhält. Aus diesem Grund soll in diesem Abschnitt ein mehrsichtiges Modell formal erfasst werden.

Definition 9 (formales Modell) *Ein* formales Modell $M = (p, Ab)$ *ist ein Paar aus einer Sprachinstanz p einer Sprache $P = (A, L, D, [\![.]\!])$ und einer Modellierungs-Abbildung Ab, die als Zielmenge die formale Sprache L hat.*

Bemerkung 4 *Der Definitionsbereich der Abbildung Ab ist nicht spezifiziert, da der Definitionsbereich nicht zwingend mathematischer Natur sein muss. Es können auch Abbildungen aus der Realität sein.*

In dieser Arbeit ist die Relation Ab, die die Relation zwischen dem Modell und der modellierten Wirklichkeit widerspiegelt, nicht interessant. Um die Schreibweisen zu vereinfachen, wird im folgenden ein Modell M mit der Sprache des Modells P gleichgesetzt.

Im Folgenden wird die Sprache eines mehrsichtigen Modells in der Überlagerungs-Interpretation definiert.

Definition 10 (mehrsichtige Modellsprache) *Eine Sprache $P = (A, L, D, [\![.]\!])$ ist eine* mehrsichtige Sprache*, wenn es Sichten gibt, die durch jeweils eine Sprache $P_i = (A_i, L_i, D_i, [\![.]\!]_i)$ mit $1 \leq i \leq i_{max}$ beschrieben werden, für welche Funktionen f_A, f_L, f_D, und $f_{[\![.]\!]}$ existieren, so dass folgendes gilt:*

1. $A = f_A(A_1, \ldots, A_{i_{max}})$,
2. $L = f_L(L_1, \ldots, L_{i_{max}})$,
3. $D = f_D(D_1, \ldots, D_{i_{max}})$,
4. $[\![.]\!] = f_{[\![.]\!]}([\![.]\!]_1, \ldots, [\![.]\!]_{i_{max}})$.

Schreibweise: $P = (P_1, \ldots, P_{i_{max}})$.

Die Funktionen f_A, f_L, f_D, und $f_{[\![.]\!]}$ spiegeln den Aufbau eines mehrsichtigen Modells nach der „Überlagerungs“-Betrachtungsweise (Kapitel 3) wider.

3.3 Die Sprache CSP-OZ

In diesem Abschnitt wird eine Sprache für mehrsichtige Modelle betrachtet. Es soll die formale Sprache CSP-OZ benutzt werden. CSP-OZ wird oft als *integrierte formale Methode* bezeichnet. Eine formale Methode beschreibt eine Sprache für formale Modelle zusammen mit einem spezifischen Vorgehen zur Entwicklung von Software. Das Adjektiv *integriert* deutet auf die Integration mehrerer Sichten hin. CSP-OZ kombiniert eine zustandsbasierte Beschreibung mittels Object-Z und eine dynamische Beschreibung mittels CSP. CSP-OZ hat im Vergleich mit anderen mehrsichtigen Modellen den Vorteil, dass CSP-OZ sowohl formal fundiert ist und eine formale Semantik für alle Sichten hat, als auch, dass alle wichtigen objektorientierten Konstrukte vorhanden sind. Auch beinhaltet CSP-OZ zwei verschiedene Arten von Integrationen der Sichten. Zum Einen gibt es zwei Sichten, die in die Klasse integriert sind, also immer zusammen als CSP-OZ-Klasse dargestellt werden. Zum Anderen werden in der Systemsicht die CSP-OZ-Klassen miteinander kombiniert. Diese Sicht ist nicht in die Klasse integriert.

Einige Sprachen wie z.B. UML haben Schwächen bei der formalen Semantik. Bei UML sind für verschiedene Diagrammarten (Sichten) verschiedene formale lokale Semantiken erarbeitet worden [EW01, FS07, KGKK02, Szl06]. Es fehlt aber eine formale globale Semantik, die für die formale Verhaltenserhaltung eines Modells benötigt wird. Es ist zu erwarten, dass diese in den nächsten Jahren aus den lokalen Semantiken zusammen gesetzt werden kann. Auch hat UML sehr viele verschiedene Diagrammarten (Sichten), so dass die Nutzung von UML sehr viel mehr Arbeit bereiten würde, als für eine grundsätzliche Betrachtung der Fragestellungen dieser Arbeit nötig ist. Im Bereich der formalen Methoden gibt es mehrere Sprachen, die mehrere Sichten beinhalten. Oft benutzen diese keine objektorientierten Techniken [Abr96, BS03, Spi92, Hoa85, pic99].

Insgesamt ist CSP-OZ für die Analyse von beweisbar korrekten Refactorings auf mehreren Sichten eine gut geeignete Sprache. Viele der in dieser Arbeit analysierten Fragestellungen und Probleme treten auch in anderen Modellierungssprachen auf.

Bevor die Einzelheiten von CSP-OZ betrachtet werden, soll ein Beispiel den Aufbau von CSP-OZ beschreiben. Im Folgenden wird eine Spezifikation (Abbildung 3.2) betrachtet, die aus der Klasse *Kasse* und einem System-CSP besteht. Eine CSP-OZ-Klasse wird als eine rechts offene Box beschrieben. Oben in der Box steht der Name der Klasse.

Das Innere einer Klasse ist in drei Bereiche aufteilbar: Im oberen, hellgrau unterlegten Bereich, beschreibt das Interface, wie eine Klasse mit anderen Klassen interagieren kann. Im dunkelgrau unterlegten Bereich (CSP-Teil) wird das dynamische Verhalten einer möglichen Instanz der Klasse beschrieben und im unteren Bereich (Z-Teil) wird die statische Beschreibung der Klasse mit dem Zustandsraum, der Initialisierung und den Methoden (hier Operationen genannt) dargestellt. Der Z-Teil einer Klasse setzt sich aus modifizierten Z-Schemata zusammen, die eine zustandsbasierte Sicht liefern.

Das Interface beschreibt, welche Operationen auf einer CSP-OZ-Klasse aufgerufen werden können und welche Parameter dort übergeben werden. Im Interface können diese Operationsdefinitionen mit `chan` oder `method` gekennzeichnet sein. Das Schlüsselwort `method` zeigt an, dass die Ablaufentscheidung sich innerhalb dieser Klasse befindet. Die Angabe eines Kanals `chan` stellt dagegen die nutzende Beziehung dar. Wenn zwei Klassen kombiniert werden, kann eine beliebige Anzahl von `chan` Aufrufen zu einer Methode benutzt werden.

Bevor die CSP-OZ Struktur weiter beschrieben wird, soll erst CSP eingeführt werden.

$System \stackrel{c}{=} Kasse$

Kasse

```
method sum
method cash : [givenAmount?, changeAmount! : ℕ]
chan print : [sum!, givenAmount!, changeAmount!, transactionID! : ℕ]
```

$main \mathrel{\widehat{=}} sum \rightarrow cash \rightarrow print \rightarrow main$

Init

$sum : \mathbb{N}$
$transactionID : \mathbb{N}$

sum

$\Delta(transactionID)$

$transactionID' = transactionID + 1$

cash

$givenAmount?, changeAmount! : \mathbb{N}$

$sum > givenAmount?$
$changeAmount! = sum - givenAmount?$

print

$sum!, givenAmount!, changeAmount!, transactionID! : \mathbb{N}$

$sum! = sum$
$givenAmount! = givenAmount$
$changeAmount! = changeAmount$
$transactionID! = transactionID$

Abbildung 3.2: Beispiel einer CSP-OZ-Spezifikation

Die hier dargestellten Sachverhalte werden im CSP-Teil und als System-CSP genutzt. Anschließend soll eine Object-Z-Klasse betrachtet werden, welche die Grundlagen für den CSP-Teil der Klasse liefert. Danach werden die Einzelteile zu CSP-OZ zusammengesetzt. Die Darstellung von CSP-OZ beschränkt sich auf die in dieser Arbeit benötigten Bestandteile. So werden in den Prädikaten hier nur Mengen-Ausdrücke und arithmetische Ausdrücke verwendet. Eine vollständige Darstellung findet sich in [Fis00].

CSP

Communicating Sequential Prozesses ist ein von Hoare [Hoa85] entwickelter Formalismus um parallel laufende Prozesse mit ihrer Kommunikation untereinander abzubilden. Dabei können die Prozesse nur über Nachrichten, den sogenannten *Events*, miteinander kommunizieren. Die Prozesse werden aus Events und primitiven Prozessen mittels Operatoren zusammengebaut. Die folgende Produktion aus der BNF zeigt, wie die Prozesse zusammengesetzt werden.

$$
\begin{array}{lcll}
Proc & = & \texttt{Stop} & \\
 & | & \texttt{Skip} & \\
 & | & \texttt{Diver} & (divergence) \\
 & | & e \rightarrow Proc & (prefixing) \\
 & | & Proc \,\Box\, Proc & (external\ choice) \\
 & | & Proc \,\sqcap\, Proc & (nondeterministic\ choice) \\
 & | & Proc \,|||\, Proc & (interleaving) \\
 & | & Proc \,{}_X\|_X\, Proc & (alphabetized\ parallel) \\
 & | & Proc \,\|_X\, Proc & (generalized\ parallel) \\
 & | & Proc \setminus X & (hiding) \\
 & | & Proc;\ Proc & (sequential\ composition) \\
 & | & \text{if } b \text{ then } Proc \text{ else } Proc & (boolean\ conditional) \\
 & | & Proc \triangleright Proc & (timeout) \\
 & | & Proc \triangle Proc & (interrupt)
\end{array}
$$

In der Produktion ist die vollständige Liste der möglichen Operationen angegeben. Die Beispiele in dieser Arbeit werden von diesen nur die Operationen Prefixing, external Choice, nondeterministic Choice und Parallel verwenden. Diese Operationen werden im Folgenden beschrieben.

Der entscheidende Punkt bei CSP ist, dass Nachrichten zwischen einzelnen Prozessen ausgetauscht werden. Jeder Prozess läuft unabhängig von anderen Prozessen, mit denen er nur mittels dieser Nachrichten kommunizieren kann. Dabei findet die Kommunikation zwischen parallel laufenden Prozessen statt. Dazu kann bei den meisten der parallelen Operatoren eine Menge von Events angegeben werden, auf die synchronisiert wird. In den Prozessen können andere Operationen, wie das hintereinander Auftreten von Events (Prefixing) oder eine Verzweigung (Choice) auftreten. Für den Aufbau eines Prozesses gibt es zwei Arten von Primitiven: Die Events, welche eine Kommunikation oder Interaktion darstellen und die primitiven Prozesse `Stop`, `Skip` und `Diver`.

Events Die Events stellen die Interaktion zwischen den Prozessen dar. Sie sind innerhalb von CSP atomar und benötigen keine Zeitdauer. Die Events bestehen aus einem Namen (z.B. *sum* oder *print*) oder aus einem Namen mit angehängten Variablen (z.B.

sum.x.y oder *print!sum?status*). Dabei werden mit einem Punkt einfache Variablen, mit einem Ausrufezeichen die Ausgabevariablen und mit einem Fragezeichen die Eingabevariablen angehängt. Events werden in dieser Arbeit in CSP-Ausdrücken mit kleinen Buchstaben bezeichnet, im Gegensatz zu den Prozessen, die mit einem Großbuchstaben beginnen.

Primitive Prozesse Der Prozess `Skip` stellt den erfolgreichen Abschluss eines Prozesses dar. Anschließend ist der Prozess terminiert, dass heißt, er führt keine weiteren Events mehr aus. Das Gegenstück ist der Prozess `Stop`, der einen Deadlock darstellt. Der Prozess `Stop` ist nicht terminiert, kann aber keine Events mehr durchführen. Der `Diver` Prozess stellt die Divergenz dar. `Diver` entspricht einem immer weiterlaufendem Programm, welches noch Events verarbeiten kann, aber ähnlich wie `Stop` nie terminiert.

Aus den Events und primitiven Prozessen werden mit Operatoren neue Prozesse zusammengebaut. Die Menge aller dieser Prozesse ist in der Produktion des Prozesses oben angegeben. Im Folgenden sollen nur die in dieser Arbeit benötigten Operationen beschrieben werden.

Prozessdefinition Die Prozessdefinition $\stackrel{c}{=}$ weißt einem Prozess einen Namen zu, der in anderen Prozessen an allen Positionen wo ein Prozess erwartet wird verwendet wird. Ein Prozess kann parametrisiert werden. Die Definition

$$P(x) \stackrel{c}{=} a \rightarrow x \rightarrow \texttt{Skip}$$

definiert einen Prozess P mit einen Parameter (x). Rechts steht der Prozess. Dieser erwartet erst das Event a und dann das welches dem Parameter übergeben wurde.

Präfix Der Präfixoperator $\rightarrow$ kombiniert ein Event mit einem Prozess. Nachdem das Event ausgeführt wurde, wird der Prozess ausgeführt. Zum Beispiel der Prozess

$$a \rightarrow P$$

möchte a ausführen, und nach dem Event a verhält er sich wie der Prozess P.

Deterministic Choice Dieser ist einer von zwei Auswahloperatoren. Der Prozess kann in zwei Richtungen verzweigen. Bei der deterministic Choice wird diese Entscheidung von aussen getrieben. Es wird mit dem Zweig fortgefahren, welcher mit einem angebotenen Event synchronisieren kann.

$$(a \rightarrow P) \,\Box\, (b \rightarrow Q)$$

möchte die Events a oder b ausführen. Wenn beide Events gleichzeitig auftreten können, wird einer der Events nichtdeterministisch ausgewählt.

Nondeterministic Choice ähnelt der deterministic Choice, jedoch ist es den synchronisierten Prozessen nicht erlaubt die Entscheidung zu treffen. Es wird hier nichtdeterministisch einer der beiden Zweige gewählt.

$$(a \rightarrow P) \sqcap (b \rightarrow Q)$$

kann sich entweder wie $(a \to P)$ oder $(b \to Q)$ verhalten. Der Prozess kann zusätzlich die beiden Events a oder b einzeln zurückweisen, das heißt, dass der Prozess nicht a bzw. nicht b erwartet. Er kann nur sicher (d.h. deadlockfrei) mit einem Prozess interagieren, der sowohl a als auch b anbietet.

Interleaving Das Interleaving stellt zwei gleichzeitig aber unabhängig voneinander laufenden Prozesse dar. Eine Synchronisation von Events ist nicht vorgesehen.

$$P \mid\mid\mid Q$$

verhält sich wie die beiden Prozesse P und Q gleichzeitig. Die Reihenfolge der Events der Prozesse ist beliebig ineinander verzahnt.

Generalized Parallel Bei diesem parallelen Operator kann eine Menge von Events angegeben werden, auf die sich die Prozesse synchronisieren müssen. Diese Events können nur auftreten, wenn beide Prozesse dieses Event anbieten. Der Prozess

$$P \underset{A}{\parallel} Q \text{ mit } A = \{a\}$$

erwartet, dass P und Q beide in der Lage sind das Event a auszuführen, ehe a ausgeführt wird. Die Menge A wird *Synchronisationsmenge* genannt. Die Prozesse in

$$(a \to P) \underset{\{a\}}{\parallel} (a \to Q)$$

können beide a ausführen und ergeben nach dessen Ausführung

$$P \underset{\{a\}}{\parallel} Q$$

Im Gegensatz dazu hat folgender Prozess einen Deadlock, weil weder a noch b ausgeführt werden kann.

$$(a \to P) \underset{\{a,b\}}{\parallel} (b \to Q)$$

Hiding Der Hiding-Operator wird genutzt um Events zu verbergen. Diese Events können nicht mehr mit einem anderen Prozess synchronisiert werden.

$$(a \to P) \setminus \{a\}$$

verhält sich, wenn a nicht in P genutzt wird, wie der Prozess P.

Ein einfaches bekanntes Beispiel für CSP-Prozesse ist die Schokoladenverkaufsmaschine [Hoa85]. Dabei wird ein Prozess für die Maschine definiert (*VendingMachine*) und ein Prozess für den Käufer (*Person*). Die Maschinen kennen zwei Events: „coin“ bedeutet, dass eine Münze eingeworfen wird und „choc“ bedeutet, dass die Schokolade ausgegeben wird. Die Maschine kann so definiert werden, dass sie erst eine Bezahlung erwartet und dann die Schokolade liefert:

$$\mathit{VendingMachine} \stackrel{c}{=} \mathit{coin} \to \mathit{choc} \to \texttt{Stop}$$

Der Käufer kann mit Münzen bezahlen oder mit einer ec-Karte („card“):

$$Person \stackrel{c}{=} (coin \rightarrow \texttt{Stop}) \Box (card \rightarrow \texttt{Stop})$$

Wenn diese beiden Prozesse mit dem Parallel-Operator verbunden werden, können diese Prozesse miteinander synchronisieren. Wie die Prozesse miteinander interagieren, hängt von dem Synchronisationssalphabet ab, welches mit angegeben wird. Wenn sowohl „coin“ als auch „card“ enthalten sind, ergibt sich:

$$VendingMachine \underset{\{coin,card\}}{\parallel} Person \equiv coin \rightarrow choc \rightarrow \texttt{Stop}$$

Dabei zeigt $\equiv$ an, dass die beiden Prozesse das gleiche Verhalten zeigen. Wenn nur auf „coin“ synchronisiert wird, ist das Ergebnis:

$$VendingMachine \underset{\{coin\}}{\parallel} Person \equiv (coin \rightarrow choc \rightarrow \texttt{Stop}) \Box (card \rightarrow \texttt{Stop})$$

Nun können im letzten Prozess die Events „coin“ und „card“ mit dem Hiding-Operator verborgen werden.

$$((coin \rightarrow choc \rightarrow \texttt{Stop}) \Box (card \rightarrow \texttt{Stop})) \setminus \{coin, card\}$$

das Ergebnis ist der nichtdeterministische Prozess

$$(choc \rightarrow \texttt{Stop}) \sqcap \texttt{Stop}$$

Der Übergang von der deterministischen Auswahl zur nichtdeterministischen Auswahl ist auf den ersten Blick überraschend. Die Events, die die Auswahl bestimmen („coin“„card“) werden versteckt, somit kann die Auswahl nicht mehr von aussen beeinflusst werden. Durch das Hiding wird hier ein Nichtdeterminismus eingeführt.

Damit ist neben der Syntax ein informelles Verständnis der Semantik von CSP erläutert. Für Beweise wird eine genaue formale Semantik benötigt. Die CSP-Semantik kann als operationelle Semantik angegeben werden. Die semantische Domäne ist ein Transitionssystem, welches durch Transitionsregeln gegeben ist.

Definition 11 (Gelabeltes Transitionssystem [Fis00])
Ein gelabeltes Transitionssystem *ist ein 4-Tupel* (S, s_0, La, T) *wobei*

- S *ist eine Menge von Zuständen,*
- $s_0 \in S$ *ist eine Startzustand,*
- La *ist eine Menge von Labeln,*
- $T \subseteq S \times La \times S$ *sind gelabelte Übergänge.*

Wenn $x, y \in S$ zwei Zustände sind und $a \in La$ ein Label ist, dann wird für $(x, a, y) \in T$ $x \xrightarrow{a} y$ geschrieben. Häufig wird die Transitionsrelation mit Hilfe von Schlussregeln angegeben:

$$\frac{V_1, \ldots, V_n}{S} \text{ mit } X$$

Aus der Erfüllung der Formeln über dem Strich $(V_1, \ldots, V_n)$ kann die Erfüllung der Formel unter dem Strich S geschlossen werden. Eine eventuell erforderliche zusätzliche Bedingung X kann angegeben werden.

In dieser semantischen Domäne werden ein paar zusätzliche Konstrukte benötigt, um das Transitionssystem aufzubauen. Zum Einen ist dies der terminierte Prozess Ω.

Definition 12 (Der terminierte Prozess) *Der terminierte Prozess Ω ist der Prozess, der keine Events mehr ausführen kann und erfolgreich terminiert ist.*

Für die Semantik von CSP-OZ sollen die Zustände S die CSP-Prozesse sein. Zusätzlich enthält S den terminierten Prozess Ω.

Für die Label werden neben den Events noch zwei zusätzliche Ereignisse benötigt:

Definition 13 (Spezielle Events [Fis00]) *In CSP werden zwei Events, die nicht im Alphabet enthalten sein dürfen, definiert.*

- *Das unsichtbare Event τ stellt einen nicht beobachtbaren Übergang dar. Ein τ-Übergang kann jederzeit erfolgen.*
- *Das Tick-Event ✓ zeigt das Terminieren eines Prozesses an. Das Tick-Event darf von jedem Prozess höchstens einmal ausgeführt werden. Nachdem das Tick-Event aufgetreten ist, dürfen keine weiteren Events mehr ausgeführt werden.*

Damit sind fast alle Bestandteile des Transitionssystem zusammen, bleibt nur noch die Übergangsregeln anzugeben. Dies geschieht getrennt nach den Operationen.

Primitive In dieser Arbeit werden die primitiven Prozesse `Skip` und `Stop` benötigt. Ein Prozess terminiert, wenn der Prozess `Skip` ausgeführt wird. Das Terminieren wird durch den ✓ Übergang angezeigt. Der terminierte Prozess ist zu keiner anderen Operation fähig.

$$\frac{}{\texttt{Skip} \xrightarrow{\checkmark} \Omega} \tag{3.1}$$

Der `Stop` Prozess stellt den Prozess dar, der nicht terminiert und keine Events ausführen kann. Daher wird keine Regel benötigt, ein `Stop` stellt ein totes Ende dar.

Präfixe Ein Prozess mit einem Prefixing verhält sich, nachdem das Event ausgeführt wurde, wie der Prozess, der nach dem Prefixing folgt.

$$\frac{}{(a \rightarrow P) \xrightarrow{a} P} \tag{3.2}$$

Generalized Parallel Der generalisierten Parallel-Operator benötigt mehrere Regeln. Die erste Regel beschreibt, wie beide Prozesse ausgeführt werden, wenn sich das Event, welches sich innerhalb der Synchronisationsmenge A befindet, ausgeführt wird.

$$\frac{P \xrightarrow{a} P', Q \xrightarrow{a} Q'}{P \underset{A}{\parallel} Q \xrightarrow{a} P' \underset{A}{\parallel} Q'} \text{ mit } a \in A \tag{3.3}$$

Ein Event, das nicht in der Synchronisationsmenge A enthalten ist, kann unabhängig von einem der Prozesse ausgeführt werden:

$$\frac{P \xrightarrow{a} P'}{P \underset{A}{\parallel} Q \xrightarrow{a} P' \underset{A}{\parallel} Q} \text{ mit } a \notin A \cup \{\checkmark\} \tag{3.4}$$

$$\frac{Q \xrightarrow{a} Q'}{P \underset{A}{\parallel} Q \xrightarrow{a} P \underset{A}{\parallel} Q'} \text{ mit } a \notin A \cup \{\checkmark\} \tag{3.5}$$

Ein paralleler Prozess terminiert, wenn beide Prozesse terminiert sind. Die Terminierung der Teilprozesse wird in ein nicht sichtbares τ-Event umgewandelt.

$$\frac{P \xrightarrow{\checkmark} \Omega}{P \underset{A}{\parallel} Q \xrightarrow{\tau} \Omega \underset{A}{\parallel} Q} \tag{3.6}$$

$$\frac{Q \xrightarrow{\checkmark} \Omega}{P \underset{A}{\parallel} Q \xrightarrow{\tau} P \underset{A}{\parallel} \Omega} \tag{3.7}$$

$$\frac{}{\Omega \underset{A}{\parallel} \Omega \xrightarrow{\checkmark} \Omega} \tag{3.8}$$

Parallel Der Parallel-Operator ähnelt dem generalisierten Parallel. Die Regeln unterscheiden sich in der Menge der Operationen, die zu den entsprechenden Regeln gehören. Wenn ein Event sich in der Schnittmenge der Mengen A und B befindet, können die Prozesse synchronisiert ausgeführt werden.

$$\frac{P \xrightarrow{a} P', Q \xrightarrow{a} Q'}{P_A \|_B Q \xrightarrow{a} P'_A \|_B Q'} \text{ mit } a \in A \cap B \tag{3.9}$$

Wenn sich das Event nicht in der Schnittmenge befindet, kann jeder Prozess das Event für sich verbrauchen.

$$\frac{P \xrightarrow{a} P'}{P_A \|_B Q \xrightarrow{a} P'_A \|_B Q} \text{ mit } a \notin (A \cap B) \cup \{\checkmark\} \tag{3.10}$$

$$\frac{Q \xrightarrow{a} Q'}{P_A \|_B Q \xrightarrow{a} P_A \|_B Q'} \text{ mit } a \notin (A \cap B) \cup \{\checkmark\} \tag{3.11}$$

Wie beim generalized Parallel wird die Terminierung der einzelnen Prozesse verborgen und der Gesamtprozess terminiert, wenn beide Prozesse terminiert sind.

$$\frac{P \xrightarrow{\checkmark} \Omega}{P_A \|_B Q \xrightarrow{\tau} \Omega_A \|_B Q} \tag{3.12}$$

$$\frac{Q \xrightarrow{\checkmark} \Omega}{P_A \|_B Q \xrightarrow{\tau} P_A \|_B \Omega} \tag{3.13}$$

$$\frac{}{\Omega_A \|_B \Omega \xrightarrow{\checkmark} \Omega} \tag{3.14}$$

Deterministic Choice Die deterministic Choice lässt einen synchronisierten Prozess entscheiden, welcher Zweig ausgeführt werden soll. Wenn beide Prozesse das Event anbieten, dann wird durch die nichtdeterministische Auswahl der Regeln ein Zweig ausgewählt.

$$\frac{P \xrightarrow{a} P'}{P \,\Box\, Q \xrightarrow{a} P'} \text{ mit } a \neq \tau \tag{3.15}$$

$$\frac{Q \xrightarrow{a} Q'}{P \,\Box\, Q \xrightarrow{a} Q'} \text{ mit } a \neq \tau \tag{3.16}$$

Ein nicht sichtbares Event τ entscheidet nicht, welcher Zweig genommen wird.

$$\frac{P \xrightarrow{\tau} P'}{P \,\Box\, Q \xrightarrow{\tau} P' \,\Box\, Q} \tag{3.17}$$

$$\frac{Q \xrightarrow{\tau} Q'}{P \,\Box\, Q \xrightarrow{\tau} P \,\Box\, Q'} \tag{3.18}$$

Nondeterministic Choice Bei der nondeterministic Choice wird der Nichtdeterminismus durch einen spontanen τ-Übergang realisiert. Nach dem Übergang ist nur noch ein Zweig vorhanden.

$$\frac{}{P \sqcap Q \xrightarrow{\tau} P} \tag{3.19}$$

$$\frac{}{P \sqcap Q \xrightarrow{\tau} Q} \tag{3.20}$$

Hiding Beim Hiding werden die versteckten Events in nicht sichtbare τ Events umgewandelt. Alle anderen Events bleiben unverändert.

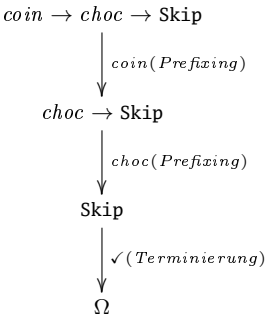
$$\frac{P \xrightarrow{a} P'}{P \setminus A \xrightarrow{\tau} P' \setminus A} \text{ für } a \in A \tag{3.21}$$

$$\frac{P \xrightarrow{a} P'}{P \setminus A \xrightarrow{a} P' \setminus A} \text{ für } a \notin A \cup \{\checkmark\} \tag{3.22}$$

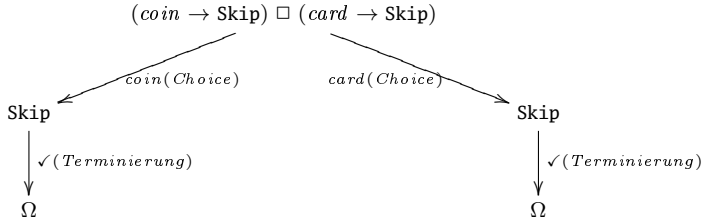
Process Invocation Wenn der Prozess auf einen Prozessbezeichner trifft, wird der entsprechende Prozess ausgeführt. Dazu wird der Prozessbezeichner durch den Prozess selber ersetzt:

$$\frac{}{X \xrightarrow{\tau} P} \text{ falls } X = P \text{ definiert ist} \tag{3.23}$$

Am Beispiel der Schokoladenverkaufsmaschine (vgl. Seite 35) soll die Anwendung der Semantik erläutert werden. Der Prozess *VendingMachine* besteht aus zwei Prefixingoperatoren und dem primitiven Prozess `Stop`.

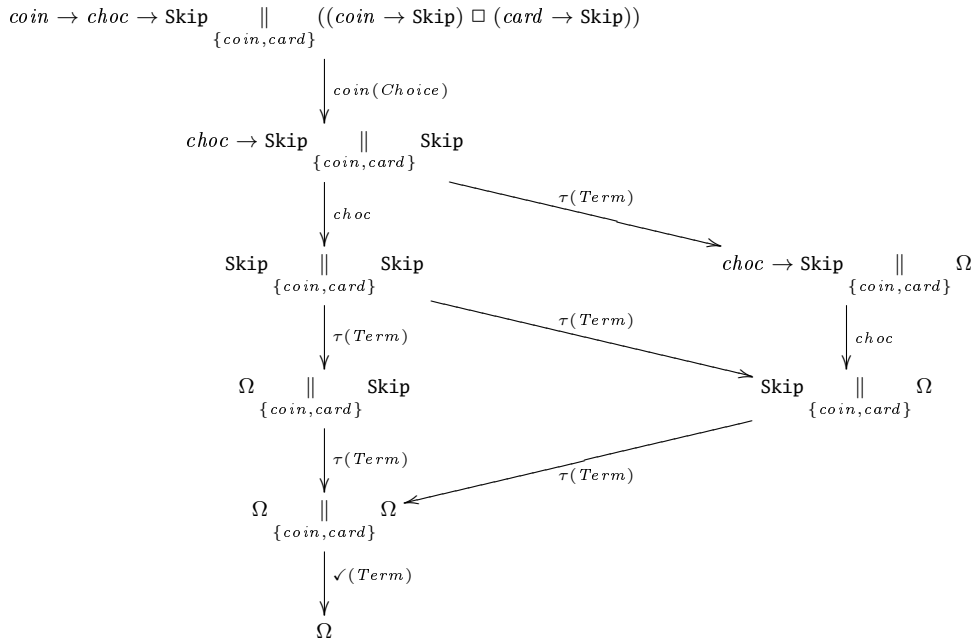


Beim Prozess *Person* wird zusätzlich die deterministic Choice verwendet. Im Transitionssystem wird dies als Verzweigung dargestellt.



In diesem Transitionssystem sind die beiden `Skip`-Knoten identisch. Die getrennte Darstellung zeigt deutlich die deterministic Choice. In weiteren Transitionssystemen werden die Knoten zusammengefasst.

Das letzte Beispiel für ein Transitionssystem zeigt, wie die Terminierung bei parallelen Prozessen funktioniert. Die Teilprozesse terminieren in einem τ-Übergang. Diese Übergänge sind hier zusätzlich mit (Term) annotiert. Der Gesamtprozess terminiert erst, wenn beide Teilprozesse terminiert sind.



Damit ist die Einführung von CSP abgeschlossen. Als zweite Voraussetzung für CSP-OZ wird Object-Z benötigt.

Object-Z

Object-Z ist eine objektorientierte Erweiterung der Z-Notation [Int02, Spi92]. Object-Z spielt in zweierlei Hinsicht eine wichtige Rolle für CSP-OZ: Zum Einen können Object-Z-Klassen in einer CSP-OZ Spezifikation verwendet werden. Zum Anderen sind die CSP-OZ-Klassen von den Object-Z-Klassen abgeleitet. Die Klasse wird, wie die CSP-OZ-Klasse, in einer rechts offenen Box beschrieben (Abbildung 3.3). Der Klassenname steht dabei oben in der Box-Linie. Eine Object-Z-Klasse beginnt mit einem einfachen Interface, in dem die nach außen sichtbaren Elemente der Klasse aufgelistet sind (⌈). Anschließend wird mit dem State-Schema der Zustandsraum aufgeführt. In dem State-Schema werden die Instanzvariablen der Klasse definiert. Das darauf folgende Init-Schema beschreibt einen initialen Zustand der Klasse. Die weiteren Teile der Klasse sind Operationen. Die Angabe $[ART]$ über der Klasse definiert einen nicht weiter bestimmten Typ. ART wird hier für die Artikel verwendet.

Visibility-List Wie in vielen objektorientierten Systemen kann von anderen Klassen nicht auf alle Bestandteile einer Klasse zugegriffen werden. In der Visibility-Liste werden alle Bestandteile der Klasse aufgeführt, auf die von aussen zugegriffen werden darf. Im Gegensatz zu anderen Systemen kennt Object-Z nur ein Alles-Oder-Nichts-Prinzip, es gibt keine feinere Zugriffssteuerung. In dieser Arbeit werden nur Operationen in dieser Liste aufgeführt, obwohl auch Variablen oder das Init-Schema aufgeführt sein dürfen. Im Beispiel Abbildung 3.3 sind die Operationen *setPreis*, *getPreis*, *getPreisMitStatus* sichtbar bzw. die Operation *reset* ist nicht sichtbar.

State Das State-Schema beschreibt den Zustandsraum der Klasse. Es wird als Schema ohne Namen angegeben. Wie die meisten Schemata hat es einen Deklarationsteil (über dem trennenden Strich) und einen Prädikatsteil (unter dem Strich). Im Deklarationsteil werden die Variablen der Klasse angegeben. Die Variablen werden als Liste von Variablennamen gefolgt von einem Typ, getrennt durch einen Doppelpunkt, angegeben. Im Beispiel sind drei Variablen (*artikelpreis*, *sonderaktion* und *rabatt*) definiert. Die Typen werden in Object-Z als Mengen angegeben. Wenn eine Variable von einem Typ ist, bedeutet das, dass die Variable einen Wert enthält, der in der Typ-Menge enthalten ist ($x : T \Rightarrow x \in T$). In der Deklaration

$$rabatt : \{5, 10, 15\}$$

wird eine Variable *rabatt* deklariert, die die Werte 5 , 10, oder 15 enthalten kann. Wenn eine Variable eine Menge enthalten soll, dann wird dies mittels der Potenzmenge der Menge angegeben (Operator: $\mathbb{P}$):

$$y : \mathbb{P}\{5, 10, 15\}$$

definiert eine Variable y, die die Werte $\varnothing$, $\{5\}$, $\{10\}$, ... , $\{5, 10\}$, ... , $\{5, 10, 15\}$ enthalten kann. Dies wird im Beispiel genutzt um die Variable *sonderaktion* zu definieren. Sie kann eine Teilmenge der Artikel ART beinhalten. Die Nutzung des Potenzmengen-Operators ist ein erstes Beispiel für die „Berechnung“ von Object-Z Typen. Da Typen Mengen sind, ist jeder Ausdruck, der eine Menge als Ergebnis hat, geeignet als Typ genutzt zu werden. Dies wird auch bei der Deklaration der zweiten State-Variablen *artikelpreis* genutzt. Die

[ART]

Preisdatenbank

$\upharpoonright (setPreis, getPreis, getPreisMitStatus)$

$$sonderaktion : \mathbb{P}\, ART$$
$$artikelpreis : ART \to \mathbb{N}$$
$$rabatt : \{5, 10, 15\}$$
$$sonderaktion \subseteq \mathrm{dom}(artikelpreis)$$

INIT

$$artikelpreis = \varnothing$$

setPreis

$$\Delta(artikelpreis)$$
$$artikel? : ART$$
$$preis? : \mathbb{N}$$
$$artikelpreis' = artikelpreis \oplus \{artikel? \mapsto preis?\}$$

getPreis

$$artikel? : ART$$
$$preis! : \mathbb{N}$$
$$artikel? \in \mathrm{dom}(artikelpreis)$$
$$preis! = artikelpreis(artikel?)$$

reset

$$\Delta(artikelpreis, sonderaktion)$$
$$artikelpreis' = \varnothing$$
$$sonderaktion' = \varnothing$$

$$getPreisMitStatus \mathrel{\widehat=} getPreis \wedge [ok! : \mathbb{B} \mid ok!]$$

Abbildung 3.3: Beispiel einer Object-Z-Klasse

Variable hat einen Funktionstyp. Eine Funktion ist eine spezielle Relation, also eine Menge von Tupeln. Die Variable *artikelpreis* kann demnach Funktionen enthalten, die einem Artikel einen Preis zuordnen. In dieser Arbeit werden zwei weitere Operatoren für Typen verwendet, die Klassentypen berechnen. Wenn in Object-Z (und CSP-OZ) als Typ ein Name einer Klasse angegeben wird, ist es, anders als in vielen Programmiersprachen, nicht erlaubt in der Variable eine Instanz einer Unterklasse zu verwenden. Wenn eine Unterklasse verwendet werden soll, muss dies explizit angegeben werden. Dazu wird vor dem Klassennamen der Subclasses-Operator $\downarrow$ geschrieben. Da bei Object-Z in den Unterklassen nicht die gleichen Operationen sichtbar sein müssen wie in der Klasse selber, sind nur Operationen verwendbar, die in der Klasse und allen ihren Unterklassen mit der selben Signatur sichtbar sind.

Ähnlich funktioniert der Class-Union-Operator $\cup$. Dieser erlaubt es, zwei oder mehrere Klassen in einem Typ zusammenzufassen. Auch hier sind nur Operationen sichtbar, die in allen Klassen sichtbar sind und die gleiche Signatur aufweisen.

In dem zweiten Teil der State-Beschreibung werden, durch den Strich getrennt, Invarianten angegeben. Sie beschreiben Bedingungen an die Variablenwerte, die immer erfüllt sein müssen. In dem Beispiel wird gefordert, dass die Menge der Artikel der Sonderaktion eine Teilmenge der Domäne der Artikel-Preis-Zuordnung ist.

Init-Schema Das Init-Schema definiert die initialen Zustände der Klasse. Es können Prädikate angegeben werden, die die Variablen des State nutzen. Die Prädikate beschreiben die Zustände. Ein gültiger initialer Zustand muss also die Prädikate aus dem Init-Schema erfüllen. Die Variable *artikelpreis* muss einen Wert enthalten, der der leeren Menge gleicht. Dies ist im Beispiel nur für die leere Menge wahr. Bei dem Gleichheitszeichen handelt es sich *nicht* um eine Zuweisung. Da neben den Bedingungen aus dem Init-Schema auch immer die Invarianten erfüllt sein müssen, wird durch das Init-Schema auch die Variable *sonderaktion* eingeschränkt. Die Invariante aus dem State-Schema fordert $sonderaktion \subseteq \mathrm{dom}(artikelpreis)$. Da die Funktion in der Variablen *artikelpreis* eine leere Funktion ist, ist auch der Definitionsbereich leer. Daher ist in einem initialen Zustand auch der Wert der Variablen *sonderaktion* die leere Menge. Die dritte Variable aus dem State (*rabatt*) ist nicht eingeschränkt, kann also jeden Wert des Typs annehmen. Im Beispiel gibt es somit drei initiale Zustände:

	sonderaktion	*artikelpreis*	*rabatt*
1. Zustand	$\varnothing$	$\varnothing$	5
2. Zustand	$\varnothing$	$\varnothing$	10
3. Zustand	$\varnothing$	$\varnothing$	15

Das Init-Schema darf nicht mit einem Konstruktor aus Programmiersprachen verwechselt werden. Da Object-Z sich deklarativen Konzepten bedient, kann das Init-Schema für eine Initialisierung genutzt werden. Die Initialisierung ist keine Pflicht in Object-Z. Es ist sogar nötig, mit nicht initialisierten Objekten arbeiten zu können, um die Veränderung einiger Datenstrukturen beschreiben zu können. Im Allgemeinen wird in dieser Arbeit, wenn es nicht explizit erwähnt wird, von initialisierten Objekten ausgegangen. Das Init-Schema muss ebenso wie das State-Schema in jeder Object-Z-Klasse genau einmal vorhanden sein.

Operations-Schema Die Operations-Schemata beschreiben die Ausführung der Operationen. Wie im Init-Schema beschreiben die Operationen mithilfe von Prädikaten, wie

die Ergebnisse aussehen. Es wird *kein* Algorithmus angegeben, wie die Ergebnisse ermittelt werden. Der Aufbau eines Operations-Schemas ist ähnlich dem des State-Schemas: Im oberen Teil befinden sich ein Dekalarationsteil und im unteren Teil Prädikate, die die Operationen beschreiben. Der Inhalt unterscheidet sich vom State; In der Deklaration werden Input- und Output-Variablen deklariert. Input-Variablen werden durch ein angehängtes Fragezeichen und Output-Variablen durch ein Ausrufezeichen gekennzeichnet. In jeder Operation sind impliziert alle State-Variablen *lesend* verfügbar. Wenn eine State-Variable durch die Operation geändert werden soll, muss dies explizit angegeben werden. Dazu werden diese in der Delta-Liste (Δ) der Operation angeben. In der ersten Operation der *Preisdatenbank* (*setPreis*), kann somit die State-Variable *artikelpreis* geändert werden. Alle anderen State-Variablen bleiben unverändert. Die Operation hat zwei Input-Variablen (*artikel*? und *preis*?). Im Prädikatsteil der Operation wird durch Prädikate beschrieben, was die Operation leistet. Da für diese Beschreibung der Zustand der State-Variablen vor und nach der Operation benötigt wird, werden die State-Variablen in einer gestrichenen Version (*artikelpreis*′) und einer normalen Version (*artikelpreis*) nutzen. Die gestrichene Version enthält den Wert der Variablen nach der Operation. In der Operation *setPreis* bedeutet das, dass die State-Variable *artikelpreis* so geändert wird, dass die Bedingung

$$artikelpreis' = artikelpreis \oplus \{artikel? \mapsto preis?\}$$

erfüllt wird. Die Funktion in *artikelpreis* entspricht nach der Operation (gestrichene Version) der alten Funktion *artikelpreis*, in dem das Mapping von dem übergebenen Artikel auf die natürliche Zahl *preis* hinzugefügt wird. Dabei stellt die Operation $\oplus$ sicher, dass ein vorhandenes Mapping für den Artikel aus der Funktion entfernt wird.

Die nächste Operation (*getpreis*) hat eine leere Delta-Liste, was bedeutet, dass sich keine State-Variable durch die Operation verändert. Dafür besitzt diese Operation eine Ausgabe-Variable (*preis*!), welche durch ein Ausrufezeichen gekennzeichnet wird. Auch hier wird durch eine Bedingung angegeben, wie die Ausgabe bestimmt wird. In diesem Fall stehen zwei Bedingungen untereinander. Diese Bedingungen müssen beide gleichzeitig erfüllt sein. Im Gegensatz zu der vorherigen Operation kann *getPreis* nicht immer genutzt werden. Wenn der Preis eines bisher nicht gespeicherten Artikels angefragt wird, sind die Bedingungen nicht erfüllbar. Dazu wird eine Vorbedingung für jede Operation pre *op* definiert.

Um die Vorbedingung definieren zu können, wird die *Relationssicht* einer Operation genutzt. Eine Operation kann als Relation zwischen dem Wert der Input-Variablen zusammen mit den Werten der State-Variablen vor der Operation und den Output-Variablen zusammen mit den Werten der State-Variablen nach der Operation angesehen werden:

$$Input, State \quad Op \quad State', Output$$

Mit Hilfe der Relationssicht, ist die Vorbedingung definiert als die Menge aller States und Eingabe-Werte, für die ein geänderter State und Ausgabe-Werte existieren:

$$\text{pre } Op := \exists\, State', Output \bullet Op \quad State', Output$$

Die Menge der Input-Werte und der States, für die dies erfüllt ist, lässt sich auch durch ein Schema beschreiben. Dieses Schema enthält keine gestrichenen Versionen der State-Variablen und auch keine Output-Variablen. Das zu pre *getPreis* gehörende Schema ist:

Operator	Bezeichnung
$\wedge$	Konjunktion
$\parallel$	Parallele Komposition
$\parallel_!$	Assoziative Parallele Komposition
$[\!]$	Nichtdeterministische Auswahl
$\overset{\circ}{9}$	Sequenzielle Komposition
$\bullet$	Scope Enrichment
$\setminus$	Hiding

Tabelle 3.1: Schema-Operatoren in Object-Z

pre *getPreis*

$$artikel? : ART$$

$$artikel? \in \mathrm{dom}(artikelpreis)$$

Definition von Operationen mittels Schemakalkül Neben der Möglichkeit eine Operation durch ein Schema zu definieren, gibt es die Möglichkeit eine Operation durch das Schemakalkül zu spezifizieren. Grob gesagt liefert dieses Kalkül eine Möglichkeit, Operationen zu kombinieren und zu verändern. Ein häufig gebrauchter Operator ist die Schema-Konjunktion $\wedge$. Mit diesem Operator werden die Bedingungen zweier Operatoren mit kompatiblen Variablen logisch konjugiert. Die Variablen sind kompatibel, wenn Variablen mit gleichen Namen aus den verschiedenen Schemata den gleichen Typ haben. Die Operation *getPreisMitStatus* ist mit Hilfe dieser Schema-Operation definiert. Die Operation *getPreis* wird mit einem anderen Schema kombiniert. Im Beispiel Abbildung 3.3 ist dieses Schema in der Inline-Schreibweise angegeben. Dabei wird das Schema innerhalb von eckigen Klammern beschrieben und der obere Teil vom unteren mittels des Querstriches getrennt. Bestandteile, die sonst untereinander stehen, werden mittels eines Semikolon getrennt. Für *getPreisMitStatus* ergibt sich als ein resultierendes Schema:

getPreisMitStatus

$$artikel? : ART$$
$$preis! : \mathbb{N}$$
$$ok! : \mathbb{B}$$

$$ok!$$
$$artikel? \in \mathrm{dom}(artikelpreis)$$
$$preis! = artikelpreis(artikel?)$$

Eine Übersicht der Operatoren befindet sich in Tabelle 3.1. In dieser Arbeit werden einige dieser Operatoren verwendet, die nun genauer erklärt werden.

Konjunktion Die Konjunktion verbindet zwei Schemata so, dass beide Schemata erfüllt werden müssen. Um die Konjunktion zu bilden, müssen alle Variablen, die in beiden Schemata auftreten, den gleichen Typ haben. Das Vorgehen setzt sich aus mehreren Schritten zusammen:

1. Die Delta-Listen werden zusammengeführt. Wenn ein Schema den Wert einer Variable verändern kann, dann kann es die Konjunktion auch.

2. Die Deklarationen werden zusammengefasst. Die neue Operation erwartet die Eingaben beider Schemata und liefert die Ausgaben beider Schemata.

3. Die Prädikate werden zusammengefasst. Es müssen die Prädikate beider Schemata erfüllt sein.

Auf diese Weise ist das resultierende Schema für *getPreisMitStatus* berechnet worden.

Scope Enrichment In einigen Fällen reicht es nicht aus, eine Konjunktion von zwei Schemata durchzuführen. Dies ist der Fall, wenn eine Variable den Typ einer anderen Variable beeinflussen oder auf ein Schema in einer Objekt-Referenz verweisen möchte. Da der letzte Fall in dieser Arbeit ausgeklammert wird, soll hier nur der erste genauer erklärt werden (Beispiel aus [Smi00]). Wenn zwei Schemata $OP_1 \mathrel{\widehat=} [y : \mathbb{P}\,\mathbb{N}]$ und $OP_2 \mathrel{\widehat=} [x : y]$ gegeben sind, würde es durch die Konjunktion ein Schema ergeben, welches nicht das gewünschte Ergebnis liefert:

$$OP_1 \wedge OP_2 = [y : \mathbb{P}\,\mathbb{N};\ x : y]$$

An dieser Stelle ist gewünscht, dass die Variablen in der Deklaration gebunden werden. Das heißt im Beispiel, dass eine Verbindung des y in beiden Schemata besteht. Um dieses darzustellen wird der Scope Enrichment Operator genutzt. Dabei wird, grob gesprochen, das zweite Schema unter der Nebenbedingung des ersten Schemas umgeformt, so dass die gemeinsamen Variablen aus der Deklaration verschwinden. Hier würde das bedeuten, dass das Schema OP_2 zu $[x : \mathbb{N} \mid x \in y]$ wird. Dabei ist zu berücksichtigen, dass aus OP_1 bekannt ist, dass y eine Teilmenge der natürlichen Zahlen ist. Anschließend können die beiden Schemata kombiniert werden. Dies führt in unserem Beispiel zu

$$OP_1 \bullet OP_2 = [y : \mathbb{P}\,\mathbb{N};\ x : \mathbb{N} \mid x \in y]$$

Sequential Composition In manchen Fällen soll eine Operation aus zwei Operationen zusammengesetzt werden, die hintereinander ausgeführt werden. Die resultierende Operation OP kann nur durchgeführt werden, wenn beide Operationen hintereinander durchgeführt werden können:

$$OP \mathrel{\widehat=} OP_1 \fatsemi OP_2$$

Falls die Operationen Eingabevariablen haben, werden diese auf die Schema verteilt. Wenn eine Eingabe in beiden Schemata verwendet wird, ist deren Wert gleich. Entsprechend werden die Ausgabevariablen vereinigt. Ein Wert einer gemeinsamen Ausgabevariablen muss beiden Schemata genügen.

Piping Eine in dieser Arbeit verwendete Schema-Operation ist keine Standard Operation. Das *Piping* erlaubt es, zwei Schemata sequentiell hintereinander zu schalten, und die Eingaben mit den Ausgaben zu verknüpfen. Das Piping ist dem ISO-Z [Int02] angelehnt. Die Idee hinter dem Piping ist, dass zwei Operationen hintereinander ausgeführt werden. Die Ausgaben der ersten Operation, die den gleichen Namen (ohne Dekoration) wie eine Eingabe der zweiten Operation haben, werden als Eingaben der zweiten genutzt.

Die übergebenen Parameter werden anschließend nach aussen verborgen. Dies entspricht der Parallel Composition von Object-Z [Smi00] mit dem Unterschied, dass es eine Richtung gibt und dass die Operationen nicht gleichzeitig sondern sequenziell betrachtet werden.

Sei beispielsweise eine Klasse mit zwei Operationen Op_1 und Op_2, die durch das Piping zu einer Operation Op_3 verbunden werden:

PipeExample

$\upharpoonright (Op_1, Op_2, Op_3)$

$a : \mathbb{N}$	INIT

Op_1	Op_2
$\Delta(a)$	$\Delta(a)$
$x?, y!, u! : \mathbb{N}$	$y?, z! : \mathbb{N}$
$y! = a$	$z! = y? + a$
$x? = a'$	$a + 1 = a'$
$u! = 2a$	

$Op_3 \mathrel{\widehat=} Op_1 >> Op_2$

Die Operation Op_1 besitzt einen Ausgabe ($y!$), die zur einer Eingabe der Operation Op_2 passt ($y?$). In diesem Beispiel wird daher die Ausgabe $y!$ zur Eingabe von $y?$. Alle anderen Aktionen werden sequentiell nacheinander ausgeführt. Die zweite Ausgabe der ersten Operation ($u!$) wird mit in die Ausgabe der Operation Op_3 übernommen. Wieder kann ein resultierendes Schema angegeben werden.

Op_3

$\Delta(a)$

$x?, u!, z! : \mathbb{N}$

$\exists\, y : \mathbb{N} \bullet (y = a \land$
$x? = a'' \land$
$u! = 2a \land z! = y + a'' \land$
$a'' + 1 = a')$

Object-Z Bindings Der Zustandsraum von Object-Z wird durch das State-Schema beschrieben. Für manche Zwecke wird das aktuelle Binding der Variablen benötigt. Dieses wird in Object-Z durch den Binding-Operator θ ausgedrückt. Ein Binding ist wie bei FWHILE eine Funktion aus Variablennamen auf Werte bzw. auf eine Objekt-Identität.

Semantik von Object-Z Die Semantik von Object-Z besteht aus einer großen Menge von Regeln, die unter anderem alle Regeln von Z [Int02] beinhaltet. Aus diesem Grund wird hier nicht die vollständige Semantik angegeben, sondern nur einige wichtige Grundzüge und Details, die später benötigt werden. Der Fokus soll auf die Semantik einer Klasse gelegt werden. Die semantische Domäne einer Object-Z-Klasse wird mit fünf Elementen beschreiben:

- Menge von sichtbaren Elementen,
- Menge der Oberklassen,
- Z-State-Schema,
- Z-Init-Schema,
- Menge von Paaren, die für jede Operation ein Z-Operations-Schema und die Delta-Liste der Operation beinhalten.

Wenn die Objektidentität nicht betrachtet wird, beschreibt ein *Abstrakter Datentyp* von Z (vgl. Anhang A.3) die Object-Z-Klasse. Die semantische Zuordnung bedient sich mehrerer Funktionen. Die wichtigste und gleichzeitig komplizierteste ist die Funktion *schema*. Sie wird genutzt, um aus einem Schema innerhalb der Object-Z-Klasse ein Z-Schema zu berechnen. Die Funktion muss deswegen einige Konstrukte aus Object-Z in entsprechende Z-Ausdrücke umformen. Eine Delta-Liste aus einem Object-Z-Operations-Schema gibt es in Z beispielsweise nicht. Deshalb wird dieses Konstrukt durch die Einbettung des State-Schema definiert. Dabei können im Gegensatz zu Object-Z alle State-Variablen verändert werden. Um dies zu verhindern, wird zusätzlich für jede State-Variable var, die nicht in der Delta-Liste vorkommt, ein Ausdruck der Art $var = var'$ hinzugefügt. Damit ist die Delta-Liste in entsprechende Z-Konstrukte umgeformt.

Auch hier werden Variablen definiert, die im Rest dieser Arbeit immer Objekte des gleichen Typs enthalten:

$superclasses$	:	Menge der Superklassen
$dlist$	:	Deltalist
$v, v1, v2$	:	Deklarationen
$p, p1, p2$	:	Prädikate
A, B, C, D	:	Klassen
Op, $Op1$, $Op2$, Opn	:	Operationsnamen
$state$	:	State-Schema
$init$	:	Init-Schema
S, T	:	Schemata
s, t	:	Ausdruck (Expressions)
OP	:	Operationen
a, b	:	Objekt-Referenzen
x, y, z	:	Variablen
$X, X_1, \ldots, X_n$	:	Generische Parameter
VL	:	Sichtbarkeitsliste (Visibility List)

Für die semantische Zuordnung werden in Object-Z Funktionen (Tabelle 3.2) verwendet. Diese werden üblicherweise ohne die Semantikklammern geschrieben. Da eine vollständige Darstellung hier den Rahmen sprengen würde, werden nur die in dieser Arbeit benötigten Details angegeben. Eine vollständige Darstellung findet sich in [Smi00].

Funktion	Beschreibung
Op	liefert für ein Operations-Schema ein Paar aus der Delta-Liste und einem Z-Schema
schema	liefert ein Z-Schema, welches die Operation repräsentiert
delta	liefert die Menge der Variablen in der Delta-Liste der Operation zurück
first	liefert das erste Element eines Tupels
second	liefert das zweite Element eines Tupels
state	liefert ein Z-Schema, welches den State repräsentiert
init	liefert ein Z-Schema, welches den initialen State repräsentiert
inherited	liefert die Menge der Oberklassennamen zurück
visible	liefert die Menge der Namen von aller sichtbaren Elementen einer Klasse zurück

Tabelle 3.2: Funktionen, die für die Semantik von Object-Z genutzt werden

Für die Beschreibung der Funktionen wird von einer Klasse ausgegangen:

$A[X_1, \ldots, X_l]$

VL
A_1
$\vdots$
A_n
$state$
$init$
$Op1$
$\vdots$
Opm

Diese verallgemeinerte Klasse nutzt die oben definierten Variablen. In dieser Klasse sind zwei Konstrukte berücksichtigt, die bisher noch nicht erklärt wurden. Zum Einen sind es die generischen Parameter. Nach dem Klassennamen können in eckigen Klammern Parameter angegeben werden, die in der Spezifikation genutzt werden. Damit ist z.B. eine Parametrisierung von Typen möglich. Das Zweite bisher nicht eingeführte Konstrukt sind die Oberklassen. Nach der Visibility-Liste können Klassennamen angegeben werden, von denen die Klasse erbt.

Visible Die Funktion **visible** liefert die Menge der Namen von sichtbaren Elementen einer Klasse zurück.

$$\textbf{visible}(A[X_1, \ldots, X_l]) = \textbf{visible}(VL)$$

wobei $\textbf{visible}(VL)$ die Elemente aus der textuellen Auflistung als Namen zurück gibt:

$$\textbf{visible}(\upharpoonright (x_1, \ldots, x_n, \textsc{Init}, Op_1, \ldots, Op_m)) = \{x_1, \ldots, x_n, \textsc{Init}, Op_1, \ldots, Op_m\}$$

Oberklassen Die Klassen, von der die Object-Z-Klasse erbt, werden im Kopf der Klasse aufgezählt. Die Funktion **inherited** gibt die Menge aller Klassen zurück, von denen geerbt wird. Die Funktion sammelt dazu rekursiv die entsprechenden Klassen:

$$\mathbf{inherited}(A[X_1, \ldots, X_l]) = \bigcup_{i \in \{A_1, \ldots, A_n\}} \mathbf{inherited}(i) \cup \{i\}$$

State Der effektive State einer Klasse setzt sich aus dem State der Klasse und den States der Oberklassen zusammen. Da die einzelnen States der Oberklassen immer die Selbstreferenz der jeweiligen Klasse (*self*) enthält, muss die *self*-Variable aus den States der Oberklassen entfernt und der jeweiligen Klasse hinzugefügt werden:

$$\begin{aligned} \mathbf{state}(A[X_1, \ldots, X_l]) \quad = \quad & [self : A[X_1, \ldots, X_l]] \\ & \wedge \bigwedge_{i \in \mathbf{inherited}(A[X_1, \ldots, X_l])} (\mathbf{state}(i) \setminus (self)) \bullet \mathbf{state}(state) \end{aligned}$$

Die Umsetzung eines State-Schemas ist das entsprechende Z-Schema:

$$\mathbf{state}([d \mid p]) = [d \mid p]$$

Init Die Funktion **init** ist ähnlich der Funktion **state** aufgebaut, mit dem Unterschied, dass nicht auf die *self*-Referenz geachtet werden muss, weil sie im Init nicht existiert:

$$\mathbf{init}(A[X_1, \ldots, X_l]) = \bigwedge_{i \in \mathbf{inherited}(A[X_1, \ldots, X_l])} \mathbf{init}(i) \bullet \mathbf{init}(init)$$

Für ein gültiges Init-Schema muss es mit dem State erweitert werden:

$$\mathbf{init}([p]) = [\mathbf{state}(A) \mid p]$$

Operation Eine Operation Op_k wird durch seine Delta-Liste und ein Z-Schema beschrieben. Dabei werden die Delta-Listen der Operation aus den Oberklassen mit der lokalen Delta-Liste vereinigt. Die Schemata der Oberklassen werden mit Hilfe des Enrichment-Operators angepasst und dann konjugiert.

$$\mathsf{Op}_k(A[X_1, \ldots, X_l]) =$$
$$\left(\bigcup_{i \in \{B_1, .., B_m\}} first(\mathsf{Op}_k(i)) \cup delta(Op_k), \bigwedge_{i \in \{B_1, .., B_m\}} second(\mathsf{Op}_k(B_i)) \bullet \mathbf{schema}(k)\right)$$
$$\text{mit } \{A_1, \ldots, A_n\} \cap \text{classdom}\mathsf{Op}_k = \{B_1, \ldots, B_m\}$$

Die letzte Zeile bedarf einer Erklärung: Sie drückt aus, dass die die Menge $\{B_1, \ldots, B_m\}$ die Menge der direkten Oberklassen ist, die die Operation auch spezifizieren. Dazu wird dies in der originalen Version von [Smi00] wie folgt beschrieben:

$$\{A_1, \ldots, A_n\} \cap \text{dom}\, \mathsf{Op}_k = \{B_1, \ldots, B_m\}$$

Der Operator dom wird in [Smi00] genutzt die Klassen zu bestimmen auf denen die Operation definiert ist. Dies ist von der Schreibweise im Kontext dieser Arbeit verwirrend. Daher

wird dies hier mit dem classdom-Operator beschrieben, der das gleiche Ergebnis liefert wie die Nutzung des dom-Operators in [Smi00].

Die zentrale Rolle innerhalb dieser Funktion spielen die Funktionen **schema** und **delta**. Diese berechnen ein Z-Schema, welches die Durchführung der Operation bzw. die Delta-Liste darstellt. Diese Funktionen berücksichtigen auch die verschiedenen Möglichkeiten eine Operation zu definieren. Im einfachsten Fall ist die Operation als Schema gegeben.

$$\begin{aligned} \mathbf{delta}([\Delta(x_1, \ldots, x_n) d \mid p]) &= \{x_1, \ldots, x_n\} \\ \mathbf{schema}([\Delta(x_1, \ldots, x_n) d \mid p]) &= [d \mid p] \end{aligned}$$

Der Fall der Schema-Konjunktion wird auf die beiden verbundenen Operationen zurückgeführt:

$$\begin{aligned} \mathbf{delta}(OP_1 \wedge OP_2) &= \mathbf{delta}(OP_1) \cup \mathbf{delta}(OP_2) \\ \mathbf{schema}(OP_1 \wedge OP_2) &= \mathbf{schema}(OP_1) \wedge \mathbf{schema}(OP_2) \end{aligned}$$

Ein bisschen komplizierter ist der Fall Scope Enrichment, da die Schemata aneinander angeglichen werden müssen:

$$\begin{aligned} \mathbf{delta}(OP_1 \bullet OP_2) &= \mathbf{delta}(OP_1) \cup \mathbf{delta}(OP_2) \\ \mathbf{schema}(OP_1 \bullet OP_2) &= [\mathbf{schema}(OP_1);\ d \mid p] \\ &\quad \text{mit } \mathbf{schema}(OP_1) \Rightarrow (\mathbf{schema}(OP_2) \Leftrightarrow [d \mid p]) \end{aligned}$$

Das Piping wird durch die sequenzielle Komposition ausgedrückt, welches eine Art Hintereinanderausführung darstellt. Die aufeinander passenden Ausgaben und Eingaben bekommen gemeinsame Variablen, die nach aussen verborgen werden.

$$\begin{aligned} \mathbf{delta}(OP_1 >> OP_2) &= \mathbf{delta}(OP_1) \cup \mathbf{delta}(OP_2) \\ \mathbf{schema}(OP_1 >> OP_2) &= (\mathbf{schema}(OP_1)[z_1/x_1!, \ldots, z_n/x_n!] \\ &\quad \fatsemi \\ &\quad (\mathbf{schema}(OP_2)[z_1/x_1?, \ldots, z_n/x_n?]) \\ &\quad \backslash (z_1, \ldots, z_n) \\ &\quad \text{mit } \mathbf{output}\ OP_1 \cap \mathbf{input}\ OP_2 = \{x_1, \ldots, x_n\} \\ &\quad \text{und } x_i \text{ sind frische Namen.} \end{aligned}$$

Nun sind die Bestandteile der Semantik von Object-Z zusammengetragen:

Definition 14 (Semantik von einer Object-Z-Klasse (vgl. [Smi00]))
Die Semantik einer Object-Z-Klasse $A[X_1, \ldots, X_l]$ ist definiert als 5-Tupel aus der Menge der sichtbaren Klassenbestandteile ***visible***$(A[X_1, \ldots, X_l])$*, der Menge aller Oberklassen* ***inherited***$(A[X_1, \ldots, X_l])$*, einem Z-State-Schema* ***state***$(A[X_1, \ldots, X_l])$*, einem Z-Init-Schema* ***init***$(A[X_1, \ldots, X_l])$ *und einer Menge von Operationen, die als Paar aus dem Schema und ihren Delta-Listen dargestellt werden,* $\{Op_k(A[X_1, \ldots, X_l])\}_{k \in \{Op1, \ldots, Opm\}}$*:*

$$
\begin{aligned}
[\![A[X_1,\ldots,X_l]]\!]_{OZ_Z} = \quad & (\textbf{\textit{visible}}(A[X_1,\ldots,X_l]),\\
& \textbf{\textit{inherited}}(A[X_1,\ldots,X_l]),\\
& \textbf{\textit{state}}(A[X_1,\ldots,X_l]),\\
& \textbf{\textit{init}}(A[X_1,\ldots,X_l]),\\
& \{\mathsf{Op}_k(A[X_1,\ldots,X_l])\}_{k\in\{Op1,\ldots,Opm\}})
\end{aligned}
$$

Nun ist ein Überblick über das Prinzip der Semantik von Object-Z gegeben. Im Verlauf der Arbeit wird neben dem informalen Semantikverständnis die formale Semantik nur beim Thema Verhaltenserhaltung benötigt. Da die Semantik mithilfe der hier angegeben Funktionen definiert ist, ist es möglich die Gleichheit der Semantik aus der Gleichheit der Funktionen abzuleiten. Dies wird später dazu genutzt, Beweise zu führen, ohne die vollständige Semantik von Object-Z vorliegen zu haben.

Mit Object-Z und CSP sind die Bausteine von CSP-OZ eingeführt.

CSP-OZ

Nachdem Object-Z-Klassen und CSP betrachtet wurden, werden die beiden Formalismen zu einer CSP-OZ-Klasse zusammengefügt. Dazu soll die Klasse *Kasse* (Abbildung 3.4) betrachtet werden.

Interface Statt wie in Object-Z nur die Elemente, die sichtbar sind, aufzuzählen, werden in CSP-OZ mehr bzw. andere Informationen benötigt. So können in CSP-OZ-Klassen keine Variablen sichtbar gemacht werden. Alle Operationsaufrufe aus dem Z-Teil (die der Object-Z-Klasse entsprechen) werden als Events von CSP aufgefasst. Die Kanäle, mit denen sich andere Klassen im Sinne von CSP synchronisieren sollen, müssen angegeben werden. Bei den Kanälen wird zwischen lokalen und nicht lokalen Kanälen unterschieden. Eine Operation, die nicht als Kanal definiert ist, kann jederzeit durchgeführt werden. Wenn dies nicht gewünscht ist und dieser Kanal nicht nach aussen sichtbar sein soll, wird dieser als lokaler Kanal `local_chan` definiert.

Im Interface können auch Parameter der Kanäle angegeben werden. Dies ist notwendig, wenn im CSP-Teil parametrisierte Events verwendet werden.

CSP-Teil Die Klasse wird durch einen CSP-Teil ergänzt. In dem angegebenen CSP-Code wird die Reihenfolge, in denen die Operationen der Klasse genutzt werden können, eingeschränkt.

Z-Teil Im unteren Teil der Klasse werden Schemata genutzt, wie sie in Object-Z auftreten. Fischer [Fis00] unterscheidet in CSP-OZ Operations-Schemata nach Schemata für Vorbedingungen, Effekte und kombinierte Schemata. Zur Vereinfachung werden in dieser Arbeit nur kombinierte Schemata betrachtet, die im wesentlichen den Operations-Schemata aus Object-Z entsprechen. Refactorings, die auch die anderen Schemata-Arten verwenden, lassen sich definieren, erweitern aber die in dieser Arbeit gezeigten Effekte nicht.

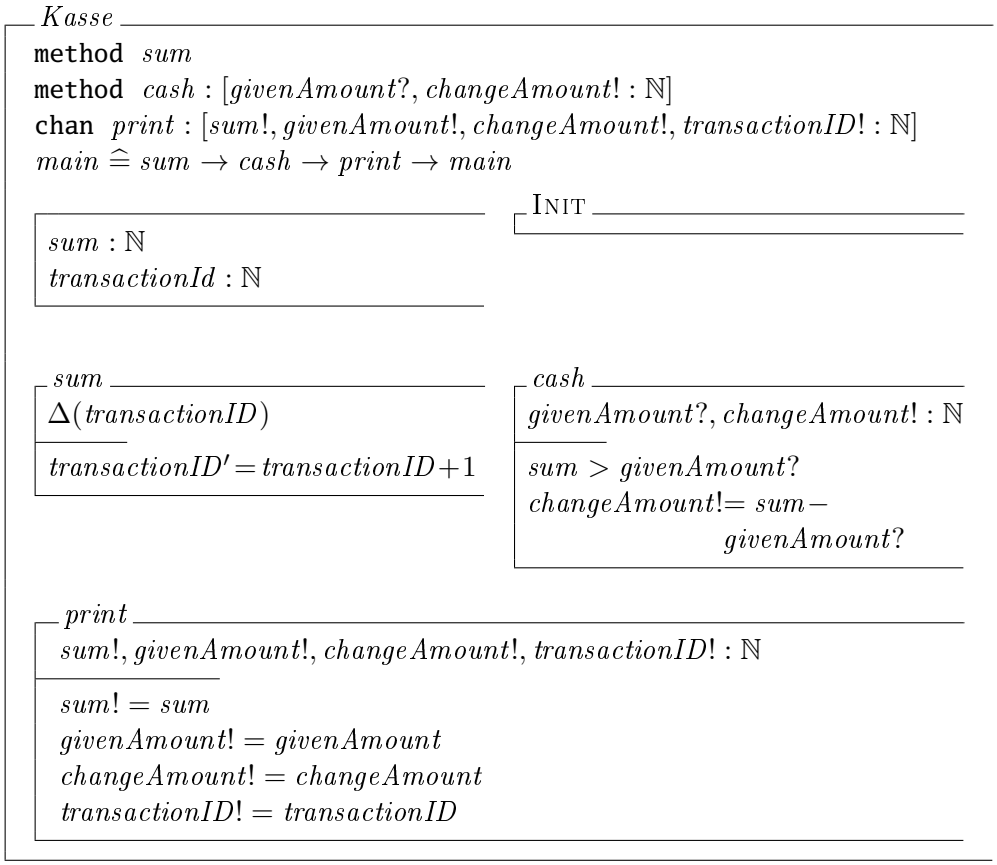


Abbildung 3.4: CSP-OZ-Klasse Kasse

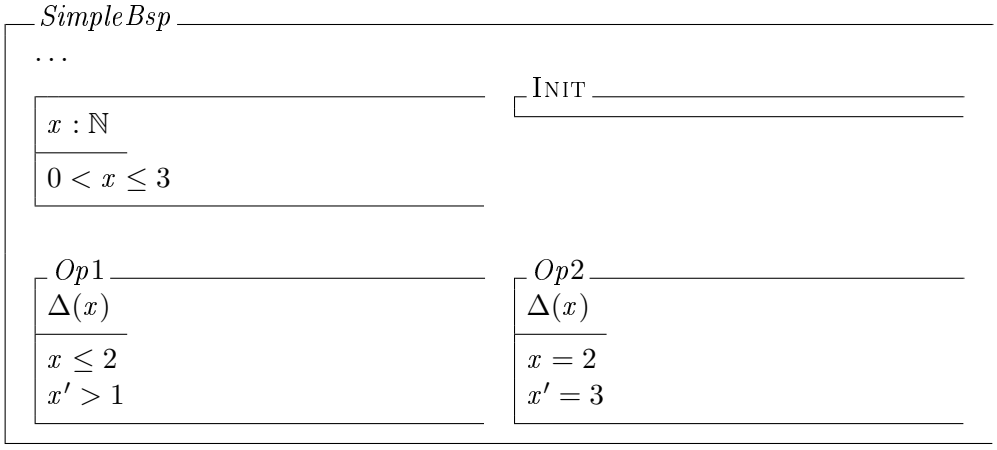


Abbildung 3.5: Eine einfache CSP-OZ-Klasse, für deren Z-Teil eine CSP-Umwandlung berechnet werden soll.

3.3.1 Semantik von CSP-OZ

Auch von CSP-OZ werden in dieser Arbeit einige Hintergründe aus der Semantik benötigt. Wieder sollen die benötigten Eigenschaften der Semantik verdeutlicht werden. Eine vollständige Darstellung der Semantik ist in [Fis00] gegeben.

Die Grundidee der Semantik von CSP-OZ ist, dass dem CSP-Teil und dem Z-Teil jeweils eine lokale Semantik mit der Domäne CSP zugeordnet wird. Da beide Teile dieselbe lokale Domäne haben, kann in dieser Domäne die Zusammensetzung der Semantik mittels CSP beschrieben werden. Im Gegensatz zu CSP, bei der die lokale Semantik direkt aus dem CSP-Teil abgeleitet wird, muss für den Z-Teil erst eine entsprechende Umwandlung in eine lokale CSP-Semantik durchgeführt werden.

3.3.2 Semantik des Z-Teils

Für die Zusammensetzung mit dem CSP-Teil wird für den Z-Teil eine Semantik mit Domäne CSP benötigt. Diese kann aus der Semantik mit Domäne Z abgeleitet werden. Im Folgenden soll diese Umwandlung exemplarisch an der Klasse *SimpleBsp* (Abbildung 3.5) vollzogen werden. Da nur der Z-Teil wichtig ist, ist das Interface und der CSP-Teil ausgelassen worden.

Die Idee der Umwandlung baut auf den Übergängen der Zustände auf. Jede Operation kann als Relation zwischen dem Zustand vor und nach der Operation aufgefasst werden. Ein- und Ausgaben werden in dieser einfachen Darstellung hier ignoriert. In der Relation würden die Input-Variablen dem Zustand vor der Operation zugeordnet, die Ausgaben dem Zustand nachher. Um diese Zustände zuordnen zu können, wird eine Erweiterung von CSP genutzt, die im Prefixing-Operator Z-Schemata erlaubt. Diese werden als Events benutzt, wobei die Variablen wie die CSP-Parameter gebunden werden. Das Schema schränkt die möglichen Werte für die Parameter wie bei Object-Z ein. Informal kann der Ausdruck

$$P \stackrel{c}{=} [x?, y! : \mathbb{N} \mid y! \leq x?] \rightarrow P$$

mit

$$OP \mathrel{\widehat=} [x?, y! : \mathbb{N} \mid y! \leq x?]$$

als

$$P \stackrel{c}{=} OP?x!y \rightarrow P$$

aufgefasst werden, wobei *Op* dabei für den Namen des Schemas steht und für die Variablen zusätzlich

$$y! \leq x?$$

gelten muss. Im Folgenden wird nicht explizit zwischen CSP und CSP_Z unterschieden.

Die Umwandlung soll anhand der vereinfachten Version der CSP-Umwandlung erläutert werden. Diese wird in [Fis00] als Grundlage genutzt und in die verschiedenen benötigten Fälle verfeinert. Ausser den bisher eingeführten Funktionen werden ein paar weitere benötigt, die ansonsten in dieser Arbeit keine Anwendung finden: die Funktion *simple* berechnet alle Simple-Variablen. Simple-Variablen sind eine spezielle Art von Variablen, die sich einige Eigenschaften von Input und Output-Variablen teilen. Grob gesagt, muss der Wert einer Simple-Variablen in Übereinstimmung der synchronisierten Methoden gewählt werden. Die beiden Funktionen *enable* und *effect* beschreiben, wann die Operation ausgeführt werden kann (*enable*) bzw. wie die Auswirkung der Operation ist (*effekt*). Da in dieser Arbeit immer mit kombinierten CSP-OZ-Schemata gearbeitet wird, entspricht *enable* hier der Vorbedingung der Operation ($enable(op) = \text{pre } Op$) und *effect* entspricht dem Operationsschema der Operation.

$$\begin{aligned} procO(op, state) \stackrel{c}{=} \quad & \Box\, input(op);\ simple(op) \mid enable(op) \bullet \\ & \sqcap\, output(op);\ state' \mid effect(op) \bullet \\ & op.(\theta input(op) \land \theta simple(op) \land \theta output(op)) \rightarrow procS(\theta state') \end{aligned}$$

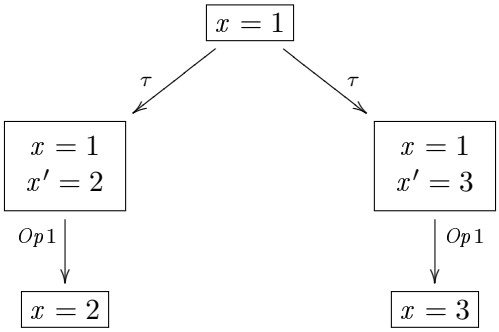
Einige vorkommende Bestandteile werden in dieser Arbeit nicht betrachtet. So werden die Simple-Variablen hier nicht benötigt. Da die nicht benötigten Teile sich zu neutralen Elementen berechnen, können diese ausgelassen werden. Damit ergibt sich die in dieser Arbeit verwendete Semantik für Operationen zu:

$$\begin{aligned} procO(op, state) \stackrel{c}{=} \quad & \Box\, input(op) \mid \text{pre } op \bullet \\ & \sqcap\, output(op);\ state' \mid schema(op) \bullet \\ & op.(\theta input(op) \land \theta output(op)) \rightarrow procS(\theta state') \end{aligned}$$

Der Prozess *procO*[1] ist ein parametrisierter Prozess, dass heißt, dass mittels Parameter Daten in den Prozess übergeben werden können. Der Prozess *procO* hat zwei Parameter: *op* ist der Name der Operation, die ausgeführt werden soll und *state* ist der aktuelle Zustand der Klasse, die State-Variablen-Belegung. Der Prozess selber ist in CSP_Z ausgedrückt. Dabei kann mittels der Choice Operation eine Auswahl von Zuständen definiert werden, die durch τ-Übergänge erreicht werden können. In dem folgenden Transitionssystem ist die Operation *Op*1 dargestellt. Im Wurzelknoten ist der State dargestellt, bei dem die Variable x den Wert 1 hat. Von der Wurzel sind zwei erweiterte States erreichbar. In den

[1] Entgegen der Konvention in CSP Prozesse mit Großbuchstaben zu kennzeichnen, werden die Prozesse der CSP-OZ-Semantik wie in [Fis00] klein geschrieben.

erweiterten States ist neben dem Wert der Variable x auch der Wert der Variable nach der Ausführung angegeben (x'). Diese Transition wird für alle erweiterten States aufgebaut, für die die Belegung (hier die Belegung von x und x') die Prädikate der Operation erfüllen. Wegen der τ-Transitionen in der Verzweigung wird der erweiterte State nichtdeterministisch ausgewählt. Dies ist im oberen Teil des folgenden Transitionssystem dargestellt und entspricht den ersten zwei Zeilen der Formel:



Der untere Teil des Transitionssystems entspricht der letzten Zeile von *procO*. Aus allen erweiterten States kann die Operation als Event synchronisiert werden. Dies ist die eigentliche Operationsausführung. Dabei wird der erweiterte State zu einem einfachen State mit der neuen Variablenbelegung reduziert. Wenn, wie in der Formel, Parameter berücksichtigt werden, werden diese mit in den erweiterten State aufgenommen und bei der Synchronisation des Operations-Events als Parameter übermittelt.

Mit dem Prozess *procO* ist die Ausführung einer Operation nach CSP übertragen worden. Da im Z-Teil von CSP jede Operation unabhängig in beliebiger Reihenfolge aufgerufen werden kann, wird die Ausführung der verschieden Operationen in dem Prozess *procS* zusammengefasst:

$$procS(state) \stackrel{c}{=} \ \Box\, op : ops \bullet procO(op, \theta state)$$

modelliert dies als external Choice über alle verfügbaren Operationen. Die Prozesse *procS* und *procO* rufen sich gegenseitig auf. Durch diese indirekte Rekursion, wird jede mögliche Ausführung des Z-Teils definiert.

Eine CSP-OZ-Klasse wird immer initialisiert. In der Gesamtsemantik des Z-Teils *procZ* wird dies durch die Ergänzung der Initialisierung ausgedrückt:

$$procZ \stackrel{c}{=} init \rightarrow procS(\theta state)$$

Das Transitionssystem in Abbildung 3.6 ist ein Beispiel für *procZ*. Das Init-Schema wird dabei als erstes ausgeführt und bildet damit die Wurzel des Transitionssystems. Anschließend werden die Operationen soweit möglich hintereinander ausgeführt.

Auf den ersten Blick sieht es so aus, als ob durch die späte Synchronisation der Operation diese intern ausgewählt wurde. Dem ist nicht so, weil vor der Operation, wie sie gerade dargestellt worden ist, eine nondeterministic Choice zur Auswahl der Operationen genutzt wird und deswegen die τ-Schritte nach Regel 3.17 bzw. Regel 3.18 von der Choice ignoriert werden. In dem Transitionssystem der Klasse zeigt sich, wie die verschiedenen Operation zusammengesetzt werden.

Damit lässt sich die CSP-Semantik des Z-Teils definieren.

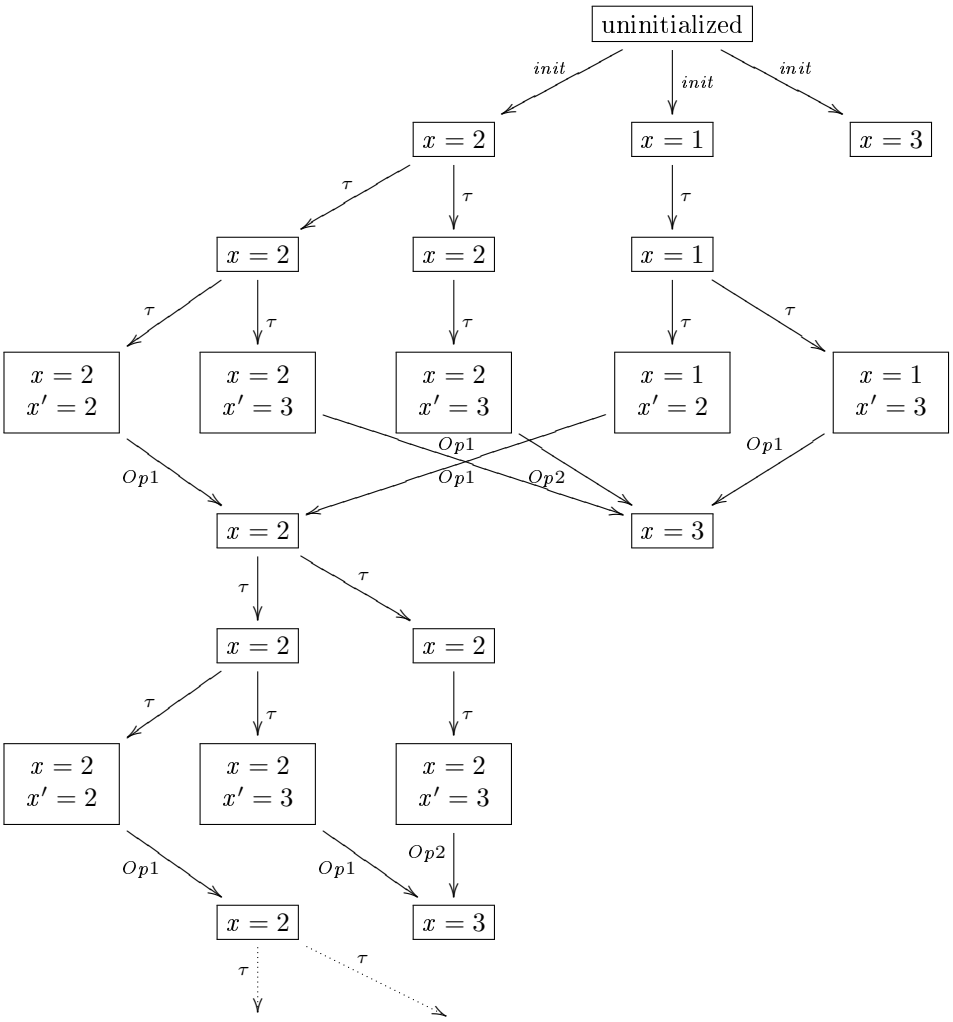


Abbildung 3.6: Transitionssystem des Z-Teils der Klasse *SimpleBsp*

Definition 15 (Lokale Semantik des Z-Teils (vgl. [Fis00]))
Sei $A[X_1, \ldots, X_l]$ ein CSP-OZ-Klasse, dann ist die lokale Semantik $[\![A[X_1, \ldots, X_l]]\!]_{OZ_{CSP}}$ definiert als:

$$[\![A[X_1, \ldots, X_l]]\!]_{OZ_{CSP}} := [\![[\![A[X_1, \ldots, X_l]]\!]_{OZ_Z}]\!]_{Z_{CSP}} = procZ$$

wobei $[\![.]\!]_{Z_{CSP}}$ die CSP-Semantik von Z ist.

3.3.3 Lokale Semantik des CSP-Teil

Die lokale Semantik des CSP-Teils wird in Form von CSP angegeben. Dies scheint überflüssig zu sein, da der CSP-Teil schon als CSP angegeben ist. Das dies nicht überflüssig ist, liegt an drei Aspekten (vgl. [Fis00]):

1. Ähnlich wie bei der Übersetzung der Integer-Zeichenfolge zu natürlichen Zahlen bei FWHILE, muss der CSP-String in einen CSP-Term umgewandelt werden.
2. Einige in dieser Arbeit nicht betrachtete Konstrukte, wie Events von Double-Event Kanälen, benötigen eine Umsetzung.
3. Der CSP-Teil muss mit den Prozessen aus den Oberklassen zusammengeführt werden.

In der hier genutzten Version von CSP-OZ müssen die Prozesse einer Klasse mit den entsprechenden Prozessen der Oberklassen kombiniert werden. Dies wird rekursiv definiert, indem jeder Prozess mit den Prozessen gleichen Namens der Oberklasse parallel kombiniert wird:

$$procC_{PE_i}(A) \stackrel{c}{=} PE_i \,{}_{Events(PE_i)}\|_{\bigcup i:1\ldots n \bullet Events(procC(A_i))} \Big\| i : 1 \ldots n \bullet Events(procC(A_i)) \mid procC(A_i)$$

Die Synchronisationsmengen sind dabei so gewählt, das sich alle sichtbaren Events der Prozesse synchronisieren. Die Funktion *Events* bestimmt alle Events, die in dem übergebenen Prozess sichtbar sind. Die Semantik des CSP-Teils ist $procC_{main}$, die die anderen Prozesse nutzt. Dafür wird im Folgenden kurz $procC$ geschrieben.

Definition 16 (Semantik des CSP-Teils (vgl. [Fis00]))
Die Semantik $[\![.]\!]_{CSPT}$ ist definiert als:

$$[\![A[X_1, \ldots, X_l]]\!]_{CSPT} = procC = procC_{main}$$

Kombination der lokalen Semantiken zur globalen Semantik einer CSP-OZ-Klasse Es sind alle lokalen Semantiken zusammengetragen, um die globale Semantik von CSP-OZ zu definieren. Abbildung 3.7 zeigt den Weg. Ausgehend von CSP-OZ ist den Sichten eine lokale Semantik gegeben worden. Bei der Semantik des Z-Teils ist das in zwei Schritten geschehen. Erst wurde eine Z-Semantik gebildet und daraus die CSP-Semantik berechnet. In diesem Abschnitt werden die lokalen Semantiken zu einer globalen Semantik zusammengeführt.

Die Idee der globalen Semantik ist, dass sowohl der Z-Teil als auch der CSP-Teil der Spezifikation erfüllt sein müssen. Daher wird die globale Semantik von CSP-OZ durch die parallele Komposition der Semantik der beiden Teile gebildet. Dabei werden die gemeinsamen Events beider Teil-Semantiken miteinander synchronisiert:

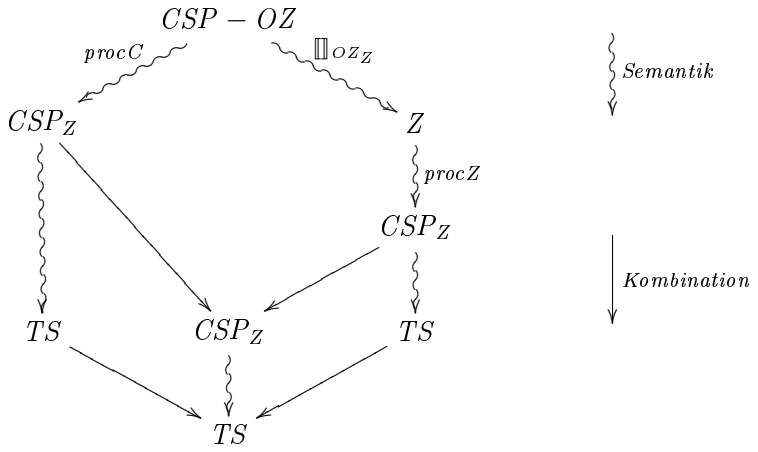


Abbildung 3.7: Struktur der Semantik von CSP-OZ

Definition 17 (Globale Semantik einer CSP-OZ-Klasse [Fis00])
Die globale Semantik $[\![A[X_1,\ldots,X_l]]\!]_{CSP-OZ}$ *einer CSP-OZ-Klasse* $A[X_1,\ldots,X_l]$ *ist die parallele Kombination der lokalen Semantik des CSP-Teils* $[\![A[X_1,\ldots,X_l]]\!]_{CSPT}$ *und der lokalen Semantik des Z-Teils* $[\![A[X_1,\ldots,X_l]]\!]_{OZ_{CSP}}$*:*

$$[\![A[X_1,\ldots,X_l]]\!]_{CSP-OZ} =$$

$$[\![A[X_1,\ldots,X_l]]\!]_{CSPT} \underset{Events([\![A[X_1,\ldots,X_l]]\!]_{CSPT})\cap Events([\![A[X_1,\ldots,X_l]]\!]_{OZ_{CSP}})}{\|} [\![A[X_1,\ldots,X_l]]\!]_{OZ_{CSP}}$$

$$= procC \underset{Events(procC)\cap Events(procZ)}{\|} procZ$$

Klassen werden durch das System-CSP zu einer vollständigen Spezifikation zusammengesetzt.

Semantik des System-CSP Klassen können durch die CSP-Semantik motiviert als CSP-Prozess aufgefasst werden. Dies wird in CSP-OZ genutzt um Klassen im System-CSP zusammenzusetzen. Ein System-CSP ist ein CSP-Ausdruck, an dem an Positionen, an welchen Prozesse erlaubt sind, auch CSP-OZ-Klassen verwendet werden können. Die Semantik des System-CSP ist gegeben, durch die Ersetzung aller Klassen durch deren Semantik. Diese Ausdrücke sind wohldefiniert, da die Semantik einer CSP-OZ-Klasse gerade einen CSP-Prozess ergibt.

Mit der globalen Semantik des System-CSP endet die Betrachtung der integrierten formalen Methode CSP-OZ.

3.4 Refinement

Refinement ist ein nützliches Konzept in vielen Formen der Ingenieurwissenschaft. Die Idee des Refinement ist es, eine Komponente durch eine genauer Spezifizierte zu ersetzen, dabei aber die Eigenschaften der ersetzten Komponente zu erhalten. In den formalen Methoden

ist *Refinement* (Verfeinerung) eine Transformation einer abstrakten (high-level) Spezifikation in eine konkrete (low-level) Spezifikation oder ein ausführbares Programm. Eine schrittweise Verfeinerung ermöglicht diesen Prozess in Etappen durchzuführen.

Die Idee der Verfeinerung ist die der Ersetzbarkeit [DB99]:

> Prinzip der Ersetzbarkeit: Es ist zulässig ein Programm durch ein anderes zu ersetzen, vorausgesetzt es ist für den Nutzer der Programms unmöglich die Ersetzung zu bemerken. Wenn ein Programm eine zulässige Ersetzung eines Programms ist, dann ist dies ein Refinement (Verfeinerung) des ersetzten Programms.

Im Zusammenhang mit den formalen Methoden, die in dieser Arbeit verwendet werden, sind zwei Arten von Refinement interessant:

- Data-Refinement für Object-Z und Z
- Prozess-Refinement für CSP und CSP-OZ

Die Unterscheidung entspricht den Bereichen, für die die Techniken entwickelt wurden. Beim Data-Refinement wird, ausgehend von einem Zustandsraum, das Refinement aufgebaut und die erreichbaren Zustände behandelt. Die Prozesse betrachten mögliche Abläufe.

3.4.1 Data Refinement für Object-Z

Data Refinement [DB99] beschreibt das Entfernen von Nichtdeterminismus aus einer Datenstruktur. Eine erste Spezifikation kann so abstrakt, wie es erforderlich ist, gehalten werden, um die erforderlichen Eigenschaften der Software zu spezifizieren. In den Schritten wird dann das System immer genauer beschrieben.

Für das Data Refinement werden für jeden Zustand die von ihm zu erreichenden Zustände betrachtet. Dazu startet das Data Refinement mit einer Menge von initialen Zuständen. Von diesen Zuständen ausgehend muss sich ein verfeinertes System so verhalten, dass es auch das Ursprüngliche sein könnte. Es verhält sich so, dass ein Beobachter das verfeinerte System für das ursprüngliche halten könnte. Meist wird in einem Verfeinerungsschritt Nichtdeterminismus entfernt. Bei Object-Z werden zur Charakterisierung von Data Refinement Simulationen genutzt. Die Idee der Simulation ist, dass für jeden Schritt eines Systems analysiert wird, welche Schritte für das andere System möglich sind. Ausgehend vom abstrakten System ergibt sich die Upward Simulation. Vom konkreten System ausgehend ergibt sich die Downward Simulation. Wenn für eine Object-Z Spezifikationen eine Upward oder eine Downward Simulation existiert, dann besteht eine Data Refinement Beziehung zwischen ihnen.

Da das Data Refinement in Beispielen für die Verhaltenserhaltung genutzt wird, soll die Darstellung hier auf die Downward Simulationen für Object-Z beschränkt werden. In der Definition wird eine verallgemeinerte Schreibweise für die Object-Z-Klassen verwendet. Dabei stehen $state^A$ und $state^C$ für die effektiven States der Klassen A bzw. C. In den States sind dabei, wie in den anderen Elementen, die Bestandteile aus den Oberklassen eingearbeitet. Entsprechende stehen $init^A$ und $init^C$ für die Initialisierungen und die Mengen $\{Op_i^A\}_{i \in I}$ bzw. $\{Op_i^C\}_{i \in I}$ für die Operationen. Der Index I ist die Menge der Operationsnamen.

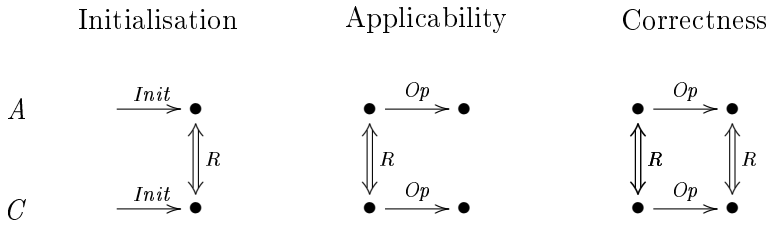


Abbildung 3.8: Prinzip der Downward Simulation beim Data-Refinement

Definition 18 (Downward Simulation für Object-Z [DB01])
Seien $A = (state^A, init^A, \{Op_i^A\}_{i \in I})$ *und* $C = (state^C, init^C, \{Op_i^C\}_{i \in I})$ *Object-Z-Klassen. Dann ist die Relation* R *auf* $state^A \wedge state^C$ *eine* Downward Simulation*, wenn*

1. *Initialisation:* $\forall\, state^C \bullet init^C \Rightarrow (\exists\, state^A \bullet init^A \wedge R)$,
2. *Applicability:* $\forall\, i \in I, \forall\, state^A, state^C \bullet R \Rightarrow (pre\ Op_i^A \Leftrightarrow pre\ Op_i^C)$,
3. *Correctness:* $\forall\, i \in I, \forall\, state^A, state^C, state^{C'} \bullet$
$$R \wedge Op_i^C \Rightarrow \exists\, state^{A'} \bullet R' \wedge Op_i^A.$$

Wenn eine solche Simulation existiert, wird C *eine* downward Simulation *von* A *genannt, geschrieben als* $A \sqsubseteq_{DS} C$.

Die Idee, die in der Downward Simulation steckt, ist in Abbildung 3.8 gezeigt. Die Bedingung Initialisation stellt sicher, dass nach der Initialisierung der konkreten Klasse C auch ein entsprechende Initialisierung in der abstrakten Klasse existiert. Die beiden anderen Bedingungen beschreiben die Simulation der Operationen. Dabei ist zu beachten, dass versteckt gefordert wird, dass die beiden Klassen die selben sichtbaren Operationen besitzen, da nur eine Indexmenge I für beide Klassen genutzt wird. Die Applicability stellt sicher, dass die Operationen der beiden Klassen unter den gleichen Voraussetzungen ausführbar sind. Die Correctness beschreibt, dass die Ergebnisse der Klassen in einem kompatiblen Zustand zueinander sind.

Wenn für zwei Klassen eine Downward Simulation gefunden werden kann, dann stehen sie in einer Data Refinement-Beziehung.

3.4.2 Prozess Refinement für CSP, CSP-OZ und Transitionssysteme

Prozesse haben durch die Modellierung des Ablaufes im Gegensatz zu Datenstrukturen andere Refinement-Begriffe. Im Zusammenhang mit CSP werden häufig drei verwandte Refinement-Konzepte [Ros97] genutzt:

- Trace Refinement
- Failure Refinement
- Failure Divergences Refinement

Trace Refinement Eine einfache Refinement-Beziehung das Trace-Refinement, arbeitet mit Sequenzen von Ereignissen (die Traces des Prozesses), die ein Prozess ausführen kann. Ein Prozess Q ist ein Trace Refinement eines anderen (P), wenn alle möglichen Folgen von Events, die Q ausführen kann, auch für P möglich sind. Trace Refinement wird geschrieben als $P \sqsubseteq_{\mathcal{T}} Q$.

Bevor das Trace Refinement definiert wird, sollen die Traces eingeführt werden: Ein Prozess $P_{bsp} \stackrel{c}{=} (a \rightarrow b \rightarrow \texttt{Skip}) \Box (b \rightarrow c \rightarrow \texttt{Skip})$ kann die Eventfolgen a, b oder b, c ausführen. Im Folgenden werden konkrete Traces mit spitzen Klammern geschrieben. Die Traces von P_{bsp} sind somit $\langle a, b\rangle$, $\langle b, c\rangle$ und die entsprechenden Präfixe der Traces (einschließlich dem leeren Trace).

Definition 19 (Trace-Transitionen) *Sei Traces eine Sequenz von Events, die das Tick-Event einschließt. Dann ist die Verallgemeinerung von Transitionen $P \stackrel{s}{\rightarrow} Q$ auf Traces induktiv definiert als*

1. $P \stackrel{\langle\rangle}{\Rightarrow} Q$, *falls* $P(\stackrel{\tau}{\rightarrow})^* Q$
2. $P \stackrel{\langle a\rangle}{\Rightarrow} Q$, *falls* $P \stackrel{\langle\rangle}{\Rightarrow} \stackrel{a}{\rightarrow} \stackrel{\langle\rangle}{\Rightarrow} Q$
3. $P \stackrel{s \frown t}{\Rightarrow} Q$, *falls* $P \stackrel{s}{\Rightarrow} \stackrel{t}{\Rightarrow} Q$

Bemerkung 5 *Hier bezeichnet $\frown$ wieder die Konkatenation (vgl. Definition 47) und $\langle\rangle$ bezeichnet die leere Trace (vgl. Definition 46).*

Die Menge der Traces eines Ausdruckes wird mit Hilfe der der Trace-Transitionen definiert:

Definition 20 (Trace [Ros97]) *Sei P ein Prozess, dann ist*

$$\mathcal{T}(P) = \{s : Traces \mid \exists\, Q : proc \bullet P \stackrel{s}{\Rightarrow} Q\}$$

die Menge der Traces des Prozesses P.

Mithilfe dieser Definition lassen sich die Traces von P_{bsp} formal ausdrücken:

$$\mathcal{T}(P_{bsp}) = \{\langle\rangle, \langle a\rangle, \langle b\rangle, \langle a, b\rangle, \langle b, c\rangle\}$$

Das Trace-Refinement fordert, dass ein verfeinerter Prozess keine Traces ausführen kann, die nicht schon vom abstrakten Prozess ausgeführt werden können.

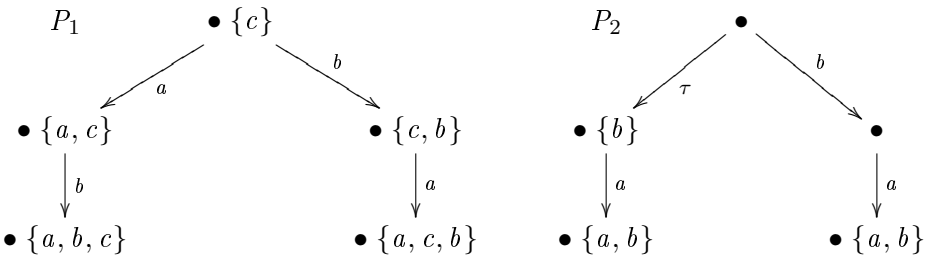
Definition 21 (Trace-Refinement [Ros97]) *Seien P und Q zwei Prozesse. Q ist ein* Trace-Refinement von P, *geschrieben $P \sqsubseteq_{\mathcal{T}} Q$, wenn*

$$\mathcal{T}(Q) \subseteq \mathcal{T}(P)$$

.

Failures und Divergences Für die weiteren Refinementbegriffe werden Failures und Divergences benötigt. Da die beiden Konstrukte eng zusammenhängen, sollen sie erst gemeinsam eingeführt und anschließend zur Definition des Failures bzw. des Failures-Divergences-Refinement genutzt werden.

Ein Failure ist ein Paar (s, X), wobei s ein Trace des Prozesses ist. X ist eine Menge von Events des Prozesses, die von ihm abgelehnt werden, das heißt, dass die Elemente aus X nach der Trace s nicht als nächstes Event auftreten können. Diese Menge X wird Refusal genannt. Die Menge der Failures wird mit $\mathcal{F}$ bezeichnet. In den Transitionssystem P_1 und P_2 [Ros97] sind die maximalen Refusals an die Knoten geschrieben. Jede Teilmenge ist ebenfalls ein Refusal.



Im Folgenden wird mit der Relation ref geprüft, ob durch eine Menge ein Refusals eines Prozesses beschrieben ist. So ist z.B. P_1 ref $\{a\}$.

Der Operator div beschreibt die Traces, die in eine nicht sichtbare Endlosschleife des Transitionssystem führt. Damit sind dies die Abläufe, die in einen Livelock führen.

Mit ref und div können die Divergences und Failures eingeführt werden:

Definition 22 (Divergences [Ros97]) *Sei P ein Prozess, dann ist*

$$\mathcal{D}[\![P]\!] = \{s, t : \mathit{Traces} \mid s \in \mathcal{D}^-(P) \bullet s \frown t\}$$

mit

$$\mathcal{D}^-(P) = \{s : \mathit{Traces} \mid \exists\, Q : \mathit{Proc} \bullet P \overset{s}{\Rightarrow} Q \wedge Q\,\mathrm{div}\}$$

die Menge der Divergences von P.

Definition 23 (Failures [Ros97]) *Sei P ein Prozess, dann ist*

$$\mathcal{F}(P) = \{s : \mathit{Traces};\ X : \mathit{Refusals} \mid \exists\, Q : \mathit{Proc} \bullet P \overset{s}{\Rightarrow} Q \wedge Q\,\mathrm{ref}\,X\}$$
$$\cup\{s : \mathit{Traces};\ X : \mathit{Refusals} \mid s \in \mathcal{D}(P) \bullet (s, X)\}$$

die Menge der Failures von P.

Failure Refinement Eine feinere Unterscheidung als Trace-Refinement zwischen Prozessen kann durch Einschränkung der Ereignisse, die als nächstes ausgeführt werden können, gemacht werden. Das Failure Refinement kann mehr Prozesse unterscheiden als das Trace Refinement. Wenn zwei Teilprozesse sich nicht synchronisieren, ist das im Trace nicht erkennbar.

Definition 24 (Failures Refinement [Ros97]) *Seien P und Q zwei Prozesse. Q ist ein* Failure Refinement*, geschrieben $P \sqsubseteq_{\mathcal{F}} Q$, wenn*

$$\mathcal{F}(Q) \subseteq \mathcal{F}(P)$$

.

Failures-Divergences Refinement So wie das Failures Refinement mehr Prozesse unterscheidet als das Trace-Refinement, kann das Failures-Divergences Refinement zusätzlich Prozesse die sich in ihrer Divergence unterscheiden. Durch die zusätzliche Betrachtung der Divergences, können Prozesse anhand von unterschiedlicher Livelock getrennt werden.

Definition 25 (Failures-Divergences [Ros97]) *Seien P und Q zwei Prozesse. Q ist ein* Failure Refinement*, geschrieben $P \sqsubseteq_{\mathcal{FD}} Q$, wenn*

$$\mathcal{F}(Q) \subseteq \mathcal{F}(P) \text{ und } \mathcal{D}(Q) \subseteq \mathcal{D}(P).$$

Damit sind drei Verfeinerungsbegriffe für CSP definiert. Da CSP als Semantik für CSP-OZ benutzt wird, können diese Verfeinerungsbegriffe auch auf CSP-OZ-Klassen angewandt werden. Dies wird im Kapitel 7.2 genutzt um die Verhaltenserhaltung von CSP-OZ zu definieren.

Es sind drei Refinement-Begriffe für CSP vorgestellt worden, die sich teilweise ähneln. Die Frage ist, worin liegen die Unterschiede und wie hängen die Begriffe zusammen. Der zweite Teil der Frage ist einfach zu beantworten. Die stärkeren Begriffe implizieren die schwächeren Refinements:

$$P \sqsubseteq_{\mathcal{FD}} Q \Rightarrow \; P \sqsubseteq_{\mathcal{F}} Q$$

$$P \sqsubseteq_{\mathcal{F}} Q \Rightarrow \; P \sqsubseteq_{\mathcal{T}} Q$$

Bleibt die Frage, worin sich die Refinements unterscheiden. Die Prozesse P und Q haben die gleichen Traces:

$$\begin{array}{ll} P \stackrel{c}{=} a \rightarrow a \rightarrow \texttt{Stop} & \mathcal{T}(P) = \{\langle\rangle, \langle a\rangle, \langle a, a\rangle\} \\ Q \stackrel{c}{=} (a \rightarrow a \rightarrow \texttt{Stop}) \sqcap \texttt{Stop} & \mathcal{T}(Q) = \{\langle\rangle, \langle a\rangle, \langle a, a\rangle\} \end{array}$$

Dennoch sind beide Prozesse nicht gleich. Q kann im Gegensatz zu P einfach nichts tun. Mit dem Failure-Refinement sind die beiden Prozesse unterscheidbar:

$$\begin{array}{l} \mathcal{F}(P) = \{(\langle\rangle, \{\}), (\langle a\rangle, \{\}), (\langle a, a\rangle, \{\})\} \\ \mathcal{F}(Q) = \{(\langle\rangle, \{\}), (\langle\rangle, \{a\}), (\langle a\rangle, \{\}), (\langle a, a\rangle, \{\})\} \end{array}$$

Q kann sofort a zurückweisen. Andere Prozesse können nicht durch Failures unterschieden werden. Die Prozesse P und R haben die gleichen Failures und somit auch die gleichen Traces:

$$P \stackrel{c}{=} a \rightarrow a \rightarrow \texttt{Stop} \qquad \mathcal{F}(P) = \{(\langle\rangle, \{\}), (\langle\rangle, \{a\}), (\langle a\rangle, \{\}), (\langle a, a\rangle, \{\})\}$$

$$R \stackrel{c}{=} (a \rightarrow a \rightarrow \texttt{Stop}) \sqcap (S) \setminus \{b\} \quad \mathcal{F}(R) = \{(\langle\rangle, \{\}), (\langle\rangle, \{a\}), (\langle a\rangle, \{\}), (\langle a, a\rangle, \{\})\}$$

$$\text{mit } S \stackrel{c}{=} b \rightarrow S$$

R zeigt mit der nicht sichtbaren „Endlosschleife“ ein anderes Verhalten als P. Dies lässt sich erst mit den Divergences unterscheiden:

$$\mathcal{D}(P) = \{\}$$
$$\mathcal{D}(R) = \{\langle\rangle\}$$

Die verschiedenen Refinementbegriffe für CSP können die Prozesse unterschiedlich gut unterscheiden. Eine feinere Unterscheidbarkeit wird durch einen komplexeren Formalismus erkauft.

3.5 Bisimulation

Eine andere Vorgehensweise um Prozesse zu vergleichen sind Bisimulationen [Ros97]. Die Idee der Bisimulation ist, dass die Prozesse sich gegeneinander simulieren. Bei der Simulation wird ein Prozess betrachtet und analysiert, ob der entsprechende Übergang auch beim anderen Prozess möglich ist. Bei Transitionssystemen, die nicht sichtbare Übergänge (bei CSP τ-Transitionen) kennen, wird meist zwischen der starken und schwachen Bisimulation unterschieden. Die schwache Bisimulation ignoriert manche Unterschiede in den τ-Transitionen.

Definition 26 (Starke Bisimulation (vgl. [Mil89])) *Seien (S_i, s_{0i}, La, T_i) mit $i = 1, 2$ über La beschriftete Transitionssysteme.*

1. *Eine Relation $R \subseteq S_1 \times S_2$ heißt* starke Bisimulation *zwischen T_1 und T_2, falls $(s_{01}, s_{02}) \in R$ ist und für alle $(s_1, s_2) \in R$ gilt:*

   (a) $s_1 \xrightarrow{\alpha}_1 s_1',\ \alpha \in La \ \Rightarrow\ \exists\, s_2' :\ s_2 \xrightarrow{\alpha}_2 s_2'$ *und* $(s_1', s_2') \in R$

   (b) $s_2 \xrightarrow{\alpha}_2 s_2',\ \alpha \in La \ \Rightarrow\ \exists\, s_1' :\ s_1 \xrightarrow{\alpha}_1 s_1'$ *und* $(s_1', s_2') \in R$

2. *T_1 und T_2 heißen* stark bisimulationsäquivalent*, geschrieben $T_1 \sim T_2$, falls es eine starke Bisimulation zwischen T_1 und T_2 gibt.*

Die starke Bisimulation unterscheidet nicht zwischen sichtbaren und unsichtbaren Übergängen. In manchen Fällen ist es hilfreich die unsichtbaren Aktionen zu ignorieren. Dies wird bei der *schwache Bisimulation* $\approx$ berücksichtigt. Es gilt dann zum Beispiel:

$$a \to \tau \to P \approx a \to P.$$

Zunächst werden Transitionen benötigt, die von τ-Übergängen abstrahieren. Dazu sei $\overset{\lambda}{\Rightarrow}$ mit $\lambda \in L \cup \{\epsilon\}$[2] wie folgt:

- $P \overset{\epsilon}{\Rightarrow} P'$ gdw. $P \xrightarrow{\tau}{}^{*} P'$, wobei $\xrightarrow{\tau}{}^{*}$ die reflexive, transitive Hülle von $\xrightarrow{\tau}$ bezeichnet,
- $P \overset{\lambda}{\Rightarrow} P'$ gdw. $P \overset{\epsilon}{\Rightarrow}\xrightarrow{\lambda}\overset{\epsilon}{\Rightarrow} P'$, wobei $\overset{\epsilon}{\Rightarrow}\xrightarrow{\lambda}\overset{\epsilon}{\Rightarrow}$ die Komposition der entsprechenden Relationen bezeichnet.

Außerdem definieren wir $\hat{\alpha} := \begin{cases} \epsilon & \alpha = \tau \\ \alpha & \alpha \in L \end{cases}$

Definition 27 (schwache Bisimulation (vgl. [Mil89])) *Seien (S_i, s_{0i}, La, T_i) mit $i = 1, 2$ über La beschriftete Transitionssysteme.*

[2] ϵ bezeichnet das leere Wort

1. *Eine Relation* $R \subseteq Q_1 \times Q_2$ *heißt* schwache Bisimulation *zwischen* T_1 *und* T_2*, falls* $(s_{01}, s_{02}) \in R$ *ist und für alle* $(s_1, s_2) \in R$ *gilt:*

   (a) $s_1 \stackrel{\alpha}{\rightarrow}_1 s_1'$, $\alpha \in La \;\Rightarrow\; \exists\, s_2' :\; s_2 \stackrel{\hat{\alpha}}{\Rightarrow}_2 s_2'$ *und* $(s_1', s_2') \in R$

   (b) $s_2 \stackrel{\alpha}{\rightarrow}_2 s_2'$, $\alpha \in La \;\Rightarrow\; \exists\, s_1' :\; s_1 \stackrel{\hat{\alpha}}{\Rightarrow}_1 s_1'$ *und* $(s_1', s_2') \in R$

2. T_1 *und* T_2 *heißen* schwach bisimulationsäquivalent*, geschrieben* $T_1 \approx T_2$*, falls es eine schwache Bisimulation zwischen* T_1 *und* T_2 *gibt.*

Für die Betrachtung der Verhaltenserhaltung ist der Zusammenhang zwischen den Bisimulationen und den CSP-Refinements interessant. Da die Bisimulation im Zusammenhang mit dem Z-Teil einer Klasse genutzt werden wird, soll die Betrachtung auf diesen eingeschränkt werden. Für den Z-Teil gilt, dass durch seine Definition die schwache Bisimulation schon die starke Bisimulation impliziert:

Satz 27.1 *Seinen* $procZ_1$ *und* $procZ_2$ *die lokalen Semantiken zweier CSP-OZ-Klassen, dann impliziert die schwache Bisimulation zwischen ihnen eine starke Bisimulation:*

$$procZ_1 \approx procZ_2 \Rightarrow procZ_1 \sim procZ_2$$

Beweis: Sei R_1 eine schwache Bisimulation für $procZ_1$ und $procZ_2$, dann kann eine starke Bisimulation R_2 aus R_1 konstruiert werden:

1. Ist $(s_{01}, s_{02}) \in R_1$ dann soll auch $(s_{01}, s_{02}) \in R_2$ sein.

2. Sein $(s_{01}, s_{02}) \in R_1$ und $(s_1, s_2) \in R_1$ mit

   $$s_{01} \stackrel{init}{\rightarrow}_1 s_1 \text{ und } s_{02} \stackrel{init}{\rightarrow}_2 s_2$$

   dann ist $(s_1, s_2) \in R_2$.

3. Sein $(s_1, s_2) \in R_1$ und $(s_1', s_2') \in R_1$ mit

   $$s_1 \stackrel{\alpha}{\rightarrow}_1 s_1', \alpha \in La \text{ oder } s_2 \stackrel{\alpha}{\rightarrow}_2 s_2', \alpha \in La$$

   wobei α eine Operationsevent ist. Eine Operationsanwendung entspricht dem Prozess *procS* unter Verwendung von *procO*. Beide Prozesse fügen vor das Operationsevent genau ein τ-Event ein (vgl. Kapitel 3.3.2). Da dies für jede der möglichen Verzweigungen möglich ist, müssen die Zustände nach den durchgeführten τ-Events zugeordnet werden. Seien *Eventops* die Menge der Events der beiden Prozesse, die sichtbar von s_1 nach s_2 bzw. s_1' nach s_2' führen. Diese existiert eindeutig, dass es sich um eine schwache Bisimulation handelt. $\hat{q}(x_1, x_2)$ mit $x_1, x_2 \subseteq$ *Eventops* bezeichne die Knotenmenge, der aus *procZ* durch die *tau*-Events erreicht werden können. Dabei gibt die Menge x_1 an, für welche Events, das erste τ-Event und x_2 die Menge der Events, für die das zweite τ-Event ausgeführt wurde. Dann ist

   $$\forall\, x_1, x_2 \subseteq Eventops \mid x_2 \subseteq x_1 \,\forall\, s_1'' \in \hat{s_1}(x_1, x_2);\; s_2'' \in \hat{s_2}(x_1, x_2) \bullet (s_1'', s_2'') \in R_2.$$

R_2 ist eine starke Bisimulation. □□

Die starke Bisimulation impliziert die Failure-Äquivalenz [EF02]. Diese Ergebnisse werden später genutzt um die Verhaltenserhaltung einer CSP-OZ-Spezifikation zu zeigen.

Kapitel 4

Refactorings

In diesem Kapitel sollen Refactorings eingeführt und einige existierende Ansätze besprochen werden. Es gibt viele Versuche, Refactorings zu definieren. Die häufigste angegebene Definition ist die nach Fowler [Fow04]:

> Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behaviour.

Fowler trifft den Gedanken vom Refactoring eines Programms gut. Er sagt, dass der Code einer Software verändert wird, auf eine Weise, dass es außen nicht bemerkt werden kann. Die Eigenschaft, dass sich das beobachtbare Verhalten nicht verändert, wird *Verhaltenserhaltung* genannt. Bei Programmen kann dies bedeuten, dass der Anwender des Programms keine Veränderung merkt. Wenn es sich um ein Refactoring einer Programmbibliothek handelt, sollte sich also die Schnittstelle und deren Verhalten nicht ändern.

Ein Problem von Fowlers Definition ist, dass sie auf dem Begriff des Codes aufbaut. Allerdings wird seit einiger Zeit nicht nur Refactoring von Code betrachtet. Einige Arbeiten behandeln das Refactoring von Modellen [AMST09, CW04, EPTJ01, MB05, MGB05] bzw. spezieller, von formalen Modellen [BM06, Cor04, LZ08, McC04, Ruh06, SPT03]. Eine Definition, die sich gut auf Modelle übertragen lässt, ist die Definition, die Roberts [Rob99] verwendet:

> A refactoring is a pair R = (pre; T) where pre is the precondition that the program must satisfy, and T is the program transformation.

Roberts beschreibt ein Refactoring als eine Transformation, die unter bestimmten Voraussetzungen durchgeführt werden darf. Die Transformation beinhaltet bei Roberts die Verhaltenserhaltung unter der Vorbedingung *pre*, da sie bei ihm eine sinnerhaltende Konvertierung darstellt. Das bedeutet, dass wesentliche Eigenschaften beibehalten werden. Obwohl die Definition nach Roberts sich nicht so angenehm lesen lässt wie die Definition von Fowler, trifft sie den Punkt eines Refactorings besser. Ein Refactoring muss nicht unter allen Umständen das Verhalten erhalten, sondern nur wenn bestimmte Gegebenheiten erfüllt sind.

Der Vorteil dieser Sichtweise ist leicht zu erkennen, wenn das Refactoring einer Programmbibliothek betrachtet wird. Eine Programmbibliothek besitzt eine Schnittstelle, die ein Programmierer nutzen kann (Application Programmer Interface, API). Wenn eine Transformation diese API verändert, zum Beispiel einen API-Aufruf umbenennt („Rename API Element“), hat sich die sichtbare Schnittstelle verändert. Das Verhalten hat sich

verändert und eine Benutzung des alten Namens führt zu einem Fehler. Somit kann „Rename API Element“ kein Refactoring sein. Auf der anderen Seite ist ein Umbenennen von Namen „Rename“, die nicht nach außen sichtbar sind, ohne weiteres möglich, wenn alle inneren Aufrufe angepasst werden. Als eine Bedingung für die Ausführung des Refactorings „Rename“ ergibt sich, dass sich das umzubenennende Element nicht in der API befindet.

Bisher sind zwei Definitionen von Refactorings genannt worden. Fowler [Fow04] stellt Refactoring als disziplinierte Verbesserung von Code dar. Roberts [Rob99] fasst Refactorings formaler auf: Refactorings sind eine Transformation unter einer Vorbedingung. Beide Definitionen sind sehr allgemein und beinhalten nicht alle benötigten Informationen. In dieser Arbeit soll daher eine eigene Definition genutzt werden:

Definition 28 (Refactoring) *Sei $P = (A, L, D, [\![.]\!])$ eine Sprache, sei $T : L \to L$ eine Transformation, $pcon : L \to \{true, false\}$ ein Prädikat und $\preceq$: $D \times D$ eine Relation. Des Weiteren sei $L' = dom[\![.]\!]$ die Menge der wohlgeformten Programme (Worte der Sprache). Dann ist $Re = (pcon, T, \preceq)_P$ ein* Refactoring der Sprache P bezüglich eines Verhaltenserhaltungsbegriffes $\preceq$, *wenn folgendes gilt:*

1. $\preceq$ *ist reflexiv und transitiv,*
2. $\forall \lambda \in L' : pcon(\lambda) \Rightarrow [\![\lambda]\!] \preceq [\![T(\lambda)]\!]$

Die Definition baut auf der Definition nach Roberts auf. Zusätzlich fließen die Abhängigkeit von einer Sprache P und ein Verhaltenserhaltungsbegriff $\preceq$ ein. Dies ist notwendig, da die Transformation sprachabhängig ist. Auch die Verhaltenserhaltung hängt von der Sprache ab, da für den Vergleich die Semantik benötigt wird. Wenn klar ist, um welche Sprache es sich handelt, wird im folgenden der Suffix weggelassen. Die Verhaltenserhaltung $\preceq$ ist als Relation angegeben. Häufig wird für die Verhaltenserhaltungsrelation eine Äquivalenzrelation genutzt, dies ist aber nicht immer der Fall. Ein Beispiel für eine solche Relation ist die Refinement-Relation. Diese ist keine Äquivalenzrelation, führt aber dennoch zu einem brauchbaren Verhaltenserhaltungsbegriff (vgl. Kapitel 7.2).

An dieser Stelle muss ein Problem im Sprachgebrauch im Bereich der Refactorings betrachtet werden: In vielen Fällen wird *Re* auch Refactoring genannt, wenn nicht klar ist, ob das Verhalten wirklich erhalten wird. Genau genommen müsste von einem *Refactoringkandidaten* gesprochen werden. Da dies im Allgemeinen nicht genau getrennt wird, soll das auch in dieser Arbeit so gehandhabt werden. Wenn gesagt wird, dass „für ein Refactoring die Verhaltenserhaltung gezeigt“ werden soll, bedeutet dies, dass für einen Refactoringkandidaten gezeigt wird, dass dieser ein Refactoring ist.

Nachdem jetzt Refactorings für eine Sprache definiert sind, soll kurz der mehrsichtige Fall betrachtet werden. Ein mehrsichtiges Modell hat sowohl eine Sprache für das Gesamtmodell, als auch für die einzelnen Sichten. Da die Sichten wegen Definition 10 (mehrsichtige Modelle, Seite 30) eng zusammenhängen, kann ein Refactoring auf einem mehrsichtigen Modell als Refactoring des Gesamtmodells aufgefasst werden. Somit kann die Definition von Refactoring direkt angewandt werden.

Für die Unterscheidung von verschiedenen Typen von Refactorings werden später Informationen über die Vorbedingung bzw. die Transformation benötigt. In einem mehrsichtigen Modell braucht die Vorbedingung des Refactorings nicht vom Gesamtmodell abzuhängen. Ebenso kann eine Transformation einige Sichten verändern, ohne die anderen anzutasten. Die Funktion impact bestimmt auf welchen Sichten pre bzw. die Transformation arbeitet:

Definition 29 *Sei* $P = (P_1, \ldots, P_{imax})$ *eine mehrsichtige Sprache mit mehreren Sichten* $P_j = (A_j, L_j, D_j, [\![.]\!]_j)$ *und seien* $p = (p_1, \ldots, p_{imax})$ *und* $p' = (p'_1, \ldots, p'_{imax})$ *Instanzen von* P.

1. *Der Wirkungsbereich* impact : *Predicate* $\rightarrow \mathbb{P}\{P_1, \ldots, P_M\}$ *einer Refactoring-Vorbedingung pcon vom Typ Predicate ist die kleinste Menge von Sichten* $\{P_{k_1}, \ldots, P_{k_m}\}$, *so dass*

$$\forall\, p_1 : P_{k_1}, \ldots, p_{imax} : P_{k_m}, p'_1 : P_{k_1}, \ldots, p'_{imax} : P_{k_m} \bullet \\ (\bigwedge_{j \in \{k_1, \ldots, k_m\}} p_j = p'_j) \Rightarrow (pcon(p_1, \ldots, p_{imax}) \Leftrightarrow pcon(p'_1, \ldots, p'_{imax}))$$

   gilt.

2. *Der Wirkungsbereich* impact : *Trans* $\rightarrow \mathbb{P}\{P_1, \ldots, P_M\}$ *einer Refactoring-Transformation* T : *Trans ist die kleinste Menge von Sichten* $\{P_{k_1}, \ldots, P_{k_m}\}$, *so dass*

$$\forall\, p_1 \colon P_1, \ldots, p_{imax} : P_{imax}, p'_1 : P_1, \ldots, p'_{imax} : P_{imax} \bullet \\ T(p_1, ..., p_n) = (p'_1, \ldots, p'_n) \Rightarrow \forall\, j : \{1, \ldots, imax\} \setminus \{k_1, \ldots, k_m\} \bullet p_j = p'_j$$

   gilt.

Nachdem Refactorings definiert sind, sollen Refactorings charakterisiert werden. Dabei werden auch die Fragestellungen aus dem Refactoringumfeld betrachtet.

4.1 Einfache und zusammengesetzte Refactorings

Aus praktischen Gesichtspunkten werden Refactorings oft in einfache und zusammengesetzte Refactorings eingeteilt. Ein einfaches Refactoring wird dabei ohne Rückgriff auf bestehende Refactorings beschrieben. Ein zusammengesetztes Refactorings nutzt andere Refactorings, um die Modifikationen auszuführen. So wird beispielsweise bei Fowler [Fow04] beim Extrahieren einer gemeinsamen Oberklasse die Refactorings „Variable nach oben ziehen“ und „Methode nach oben ziehen“ verwendet.

4.2 Einsatz von Refactorings

Bisher ist nur davon gesprochen worden, was ein Refactoring ist. Offen blieb dagegen wozu Refactorings verwendet werden. Der in der Argumentation um Refactorings meistgenannte Grund ist die Verbesserung der (inneren) Softwarequalität. Hierbei ist mit Softwarequalität [BM03, MS01] nicht die Fehlerfreiheit, wie sie beim Testen oder Verifizieren von Software betrachtet wird, gemeint, sondern Eigenschaften wie z.B. Wartbarkeit. Die Bezeichnung *innere Qualität* drückt aus, dass es nicht um von aussen beobachtbare Qualität geht, die auch äußere Qualität genannt wird. Die innere Qualität lässt sich durch die folgenden Begriffe fassen:

- Lesbarkeit
- Übersichtlichkeit
- Verständlichkeit

- Erweiterbarkeit
- Vermeidung von Redundanz
- Testbarkeit

Dabei werden Refactorings hauptsächlich auf Stellen angewandt, die die obigen Qualitätsziele verletzen. Diese Verletzungen werden mit *Bad Smells* [Fow04] bezeichnet, was bildlich mit stinkendem Mist gleichzusetzen ist.

Von den 14 Smells, die Fowler aufführt, sollen hier nur zwei Beispiele betrachtet werden:

Duplizierter Code Häufig findet sich die gleiche Codestruktur an mehreren Stellen im Code. Wenn der Code sich nicht unterscheidet, wird dies Codedoppelung genannt. Häufig ist der Code nicht genau deckungsgleich, sondern es lassen sich nur kleine Unterschiede erkennen, bei denen der Code durch eine passende Parametrisierung angeglichen werden kann. Dies wird *duplizierter Code* genannt. Der Name deutet die häufigste Ursache für diesen Smell an: die Copy-und-Paste-Programmierung. Dabei wird der Code von einer anderen Stelle kopiert und an einer anderen Stelle eingefügt (dupliziert) und dort nur geringfügig verändert. Der Grund ist, dass der duplizierte Code eine ähnliche, wenn nicht sogar gleiche, Aufgabe an der neuen Position erfüllt.

Long Method Methoden sollen übersichtlich sein, damit diese schnell und gut zu verstehen sind. Wenn eine Methode zu lang wird, leidet das Verständnis. Auch ist eine lange Methode oft ein Zeichen mangelnder Strukturierung.

Die beiden Beispiele von Smells zeigen, dass es sich um Probleme handelt, die einsichtig sind und die häufig im Code gefunden werden. Smells beschreiben dabei die beobachtbaren Auswirkungen. Die Idee der Qualitätsverbesserung ist es, gezielt Refactorings zu nutzen um die Smells zu entfernen. So werden bei Fowler für alle Smells Refactorings angegeben, die zu deren Beseitigung genutzten werden sollten. Mit dem Refactoring „Extract Method“ kann beispielsweise der Smell „Long Method“ bekämpft werden, indem die Methode auf mehrere kleinere verteilt wird.

Ein zweiter Einsatzbereich ist die Evolution von Software. Die Evolution beschreibt dabei die Weiterentwicklung einer Software innerhalb ihres Lebenszyklus. Diese Weiterentwicklung kann sowohl Fehlerkorrekturen als auch Funktionsänderungen beinhalten. Jeder Schritt der Evolution führt zu einer neuen Version der Software. Die Evolution entspricht dabei bedingt der Wartung in gängigen Softwareentwicklungsprozessen. Refactorings werden bei der Evolution zur Vorbereitung bzw. Nachbereitung der verändernden Evolutionsschritte verwendet. Wenn z.B. eine Softwarekomponente in zwei Teile geteilt werden soll, können Refactorings genutzt werden, um die Softwarebestandteile umzuverteilen.

4.3 Refactoring und Restructuring

In der Literatur werden nicht alle Verfahren, die sich mit der Verhaltenserhaltung von Veränderung beschäftigen, Refactoring genannt. Auch sagen einige Softwareentwickler über Refactorings, dass es ein „Alter Hut mit neuem Namen“ sei. Ältere Veröffentlichungen

bestätigen dies teilweise, z.B. beschrieben Chikofsky und Cross [CC90] schon Transformationen, die das Verhalten erhalten. Dort wird die Technik *Restructuring* genannt und wird bereits zur Qualitätsverbesserung eingesetzt. Auffallend ist, dass Restructuring nach der Refactoring-Definition von Fowler und Roberts auch Refactorings sind. Worin liegt der Unterschied? Im Folgenden sollen die beiden Ansätze verglichen und zur Charakterisierung von Refactoring genutzt werden.

Beschreibung eines Refactoring/eines Restructuring Beim Restructuring sind die Anweisungen im Gegensatz zu Refactorings gröber. Eine typische Anweisung aus dem Restructuring ist, dass aus `Else`-verschachtelten `If`-Anweisungen eine `Case`-Anweisung erstellt werden soll. Es wird dabei nur das Ziel des Restructurings angegeben und auf welche Konstrukte im Programm es angewendet werden kann. Es ist nicht beschrieben wie die Umsetzung erfolgt. Es ist somit die Aufgabe des Entwicklers, die Umformungen mit Hilfe seiner Erfahrung durchzuführen.

Bei Refactorings wird oft detailliert beschrieben, wie ein Refactoring durchgeführt werden soll. Es wird sich dabei nicht auf das Wissen und die Erfahrung einer Programmierers verlassen. Im gewissen Sinne wird das Wissen und die Erfahrung durch die Beschreibung eines Refactorings konserviert. Somit sind Refactorings auch für unerfahrende Programmierer und Anfänger einsetzbar.

Kataloge Im vorherigen Punkt der Charakterisierung wurde beschrieben, dass Refactorings meist als genau beschreibende Arbeitsanweisungen vorliegen. Die Anweisungen werden in Katalogen gesammelt, so dass dem Entwickler eine Reihe von Refactorings zur Auswahl stehen. Die Kataloge enthalten dabei meist grundlegende Refactorings, die zusammen zum Erreichen eines Zieles eingesetzt werden. Die Kataloge beschränken sich dabei nicht auf die Beschreibung, wie ein Refactoring auszuführen ist, sondern behandeln auch die Vor- und Nachteile der Struktur, die erzeugt wird. Die beiden bekanntesten Kataloge sind der von Fowler [Fow04] und der von Cunningham und Cunningham [Cun].

Für Restructuring gibt es nur Kataloge für die wünschenswerten Zustände des Codes. Auch sind diese Kataloge nicht so ausgeprägt wie beim Refactoring.

Durchführung Die Durchführung von Restructuring ist, auch mangels genauer Spezifikation, Handarbeit. Bei Refactorings war es schon früh wichtig, eine halbautomatische Durchführung in Entwicklungsumgebungen einzubauen. So gehört zu den ersten Veröffentlichungen über Refactorings schon der *Refactoring Browser*, welcher sich in die Entwicklungsumgebung von Smalltalk einfügte [BR98]. Heute unterstützen viele Entwicklungsumgebungen für häufige Sprachen mindestens einige Refactorings.

4.4 Refactoring und Reengineering

Reengineering beschäftigt sich ebenfalls mit der inneren Umgestaltung von Software und behandelt so die gleichen Themen wie das Refactoring und das Restructuring. Auch hier ist die Frage wie sich beide unterscheiden.

Reengineering bezeichnet die Überarbeitung eines Softwaresystems aus verschiedenen Gründen. Dazu gehören Qualitätsverbesserungen, Performancesteigerungen und das Anpassen auf eine neue Plattform. Dabei soll das Verhalten sich nicht wesentlich ändern.

Ein Reengineering kann auf verschiedenen Ebenen durchgeführt werden: Auf Code-Ebene, auf Modell-Ebene oder auf Spezifikationsebene. Häufig müssen hierzu benötigte abstraktere Ebenen aus dem Code wiederhergestellt werden, da sie verloren sind oder durch „undokumentierte“ Änderungen nicht mehr zueinander konsistent sind. Dieser Prozess wird häufig *Reverse Engineering* genannt.

Insgesamt beschreibt Reengineering den Gesamtprozess der Verbesserung. Refactorings können dabei als Technik innerhalb des Reengineering betrachtet werden. Wenn ein Modell durch reverse Engineering wieder hergestellt wurde, kann auf diesem Modell ein Refactoring durchgeführt werden. Anschließend wird die Änderung in die Software übernommen.

4.5 Themenfelder im Zusammenhang von Refactorings

Im Themenfeld Refactoring gibt es mehrere Fragestellungen, die direkt oder indirekt mit Refactorings zusammenhängen. Eine Übersicht geben u.a. [MT04] und [MRG09].

Wie verbessert sich die Qualität durch Refactorings? Im Umfeld von Refactorings wird immer wieder als Grund für Refactorings die Verbesserung der Qualität von Software angeführt. Die Frage in dem Zusammenhang ist, ob sich diese Verbesserung nachweisen lässt und wieso sich die Qualität verbessert. Die Analyse der Verbesserung baut oft auf Metriken auf, die die Qualität der Software beschreiben. [BM03] analysiert wie sich die Werte von Metriken verändern, wenn Refactorings angewandt werden. Die meisten Ergebnisse zu Qualitätsverbesserungen und Metriken sind ein Nebenprodukt der Forschung zum Auffinden von Refactoringmöglichkeiten mittels Metriken [SSL01, DDN00, Boi04]. Für dieses Finden von Refactoringmöglichkeiten ist die Verbesserung der Metrikwerte eine Voraussetzung.

Welche Refactorings sollen wann genutzt werden? Als Grund für Refactorings wird immer wieder die Verbesserung der Qualität der Software angeführt. Von den Smells ausgehend kann ein Entwickler Anhaltspunkte finden, wo diese Technik angewandt werden soll. Dieses Vorgehen stellt hohe Anforderungen an den Entwickler: Er muss sowohl die „stinkende“ Stelle im Code erkennen und finden, als auch geeignete Refactorings auswählen, um den Smell zu beseitigen. Das Ziel an dieser Stelle ist es festzustellen, wo sich Smells befinden und wie diese beseitigt werden können. Für das Finden von Smells gibt es mehrere Ansätze. Metriken, die für Software und Modelle definiert werden können, helfen Messungen der Qualität durchzuführen [ISO02, HM95b, HM95a, CG90, GMP03]. Eine andere Möglichkeit besteht in der expliziten Analyse von Problemen. Das meist betrachtete Problem ist das der Code-Duplizierung. Zur Feststellung dieses Problems gibt es spezielle Tools, die duplizierte Stellen erkennen und helfen sie zu beseitigen [BKA+07a, BKA+07b, KBS08]. Diese Bereiche werden in der Idee des „Refactoring Opportunity Detection“ zusammengefasst. Hierunter werden Techniken verstanden, die nicht nur die Probleme aufzeigen, sondern anschließend analysieren, wie diese beseitigt werden können [TM03, YLM+98, MRG+06].

Einen anderen Weg schlägt [DDN00] ein. Hier wird die Veränderung von Code durch einen Entwickler betrachtet und anschließend analysiert, durch welche Refactorings diese Änderungen begründet werden können.

Automatische Anwendung von Refactorings Mit der Identifizierung von Problemen in einem Programm oder Modell, wie es im letzten Abschnitt beschrieben wurde, ist der Weg zu einer vollständig automatischen Ausführung von Refactorings nicht weit. Dies ist insbesondere bei großen Systemen wünschenswert. Für die Verbesserung dieser Systeme werden erfahrene Entwickler benötigt, denen zusätzlich ein großes Zeitkontigent zugestanden werden muss. Obwohl dies bisher noch ein Wunsch geblieben ist, gibt es viel versprechende halbautomatische Ansätze in diese Richtung: z.B. erlaubt [SPTJ01] Refactorings aufgrund von Regeln anzuwenden. Dies ermöglicht die zusammenhängende Bereinigung von häufig in einem System auftretenden Smells. Eine andere Möglichkeit wurde im Rahmen der Projektgruppe RMC [KDO+08] in Form eines Diagnosemodells betrachtet. Aufgrund von Messungen werden Diagnosen gestellt, die an einem Ort des Modells geknüpft sind. So ein Ort kann z.B. eine Methode oder eine Klasse sein. Aufgrund der Analyse werden die Refactorings mit einem Vorschlag für die Anwendung gegeben. Dabei werden die Parameter des Refactorings möglichst vollständig bestimmt.

Was sind brauchbare Verhaltenserhaltungsbegriffe? Während bei Programmen der Begriff der Verhaltenserhaltung leicht zu definieren ist, gibt es bei Modellen größere Probleme. Eine Zusammenstellung von möglichen Verhaltenserhaltungsbegriffen findet sich in [MT04]. Meist wird abhängig vom bearbeiteten Umfeld ein passender Verhaltenserhaltungsbegriff definiert. Für Modelle, die einen Refinement-Begriff besitzen, bietet es sich an, diesen zur Definition der Verhaltenserhaltung zu nutzen [Cor04]. In dieser Arbeit wird dieses Themenfeld genauer in Kapitel 7.2 betrachtet.

Erhalten Refactorings das Verhalten? Die Definition nach Fowler [Fow04] fordert, dass sich das Verhalten durch ein Refactoring nicht verändern soll. Um dies sicherzustellen wird in den Anfängen von Refactorings meist das Prüfen der Durchführung mittels Tests genutzt. Da Tests immer unvollständig sind, wurde begonnen, die Verhaltenserhaltung mit anderen Techniken sicherzustellen. In [ERW07, EW08] wird beispielsweise nach der Durchführung des Refactorings die Verhaltenserhaltung durch Model Checking nachgewiesen. Die Prüfung der Verhaltenserhaltung ist eines der beiden Hauptthemen dieser Arbeit und wird in Kapitel 7.2 betrachtet. Weitere Techniken sind in dem Kapitel 8.1 „Verwandte Arbeiten“ aufgelistet.

Mit der Übersicht der Themenbereiche in der Forschung über Refactorings soll die allgemeine Betrachtung von Refactorings schließen. Nachdem im nächsten Kapitel mit der Betrachtung von mehrsichtigen Modellen, insbesondere CSP-OZ, die Grundlagen abgeschlossen werden, sollen anschließend einige der Themenbereiche, die hier aufgezählt wurden, genauer behandelt werden.

Kapitel 5

Beschreibung von Refactorings

Refactorings sind unbestritten ein wichtiges Werkzeug in der Entwicklung und Evolution von Software und Softwaremodellen. Oft werden Refactorings zwischen Entwicklern diskutiert oder in Entwicklungsumgebungen eingebaut. Für die Nutzung müssen Refactorings so beschrieben werden, dass die Anwendung eines Refactorings allgemein verständlich ist. Wenn die Durchführung eines Refactorings in ein Tool eingebaut werden soll, muss die Beschreibung in einer Form vorliegen, die ein Programm verarbeiten kann. Ziel dieses Kapitels ist es, eine Refactoringbeschreibung für mehrsichtige Sprachen zur Verfügung zu stellen. Dabei muss diese Sprache verschiedene Eigenschaften erfüllen. Dazu werden bestehende Beschreibungstechniken für Refactorings vorgestellt und verglichen. Alle Ansätze haben für ihren Einsatzzweck passende Eigenschaften. Es wird sich herausstellen, dass sich für den in dieser Arbeit genutzten Ansatz keine geeignete existierende Beschreibungstechnik finden lässt. Im Hauptteil des Kapitels wird deshalb eine neue Beschreibungstechnik eingeführt.

5.1 Bestehende Beschreibungssysteme

Für die Beschreibung von Refactorings sind verschiedene Möglichkeiten in der Literatur vorgestellt worden. In diesem Unterkapitel werden einige dieser Möglichkeiten zusammengefasst. Eine der Bekannteren ist die *natürlichsprachliche Beschreibung*, wie Fowler sie in seinem Buch [Fow04] verwendet. Die Beschreibung mittels Vorbedingung und Transformation von Roberts [Rob99] stellt eine wichtige theoretische Grundlage für viele Refactoring-Beschreibungssysteme dar. Roberts beschreibt dabei die Transformation als Nachbedingung, die, wenn die Vorbedingung erfüllt ist, ausgeführt werden kann. Hierzu zählt u.a. die Beschreibung von Refactorings als Graphtransformationen, die u.a. bei UML angewandt werden können. Eine Spezialisierung von Graphen sind die Abstract-Syntax-Trees (AST, vgl. Kapitel 2), auf welchen spezialisierte Graphmanipulationen durchgeführt werden können. Die letzte hier beschriebene Beschreibungstechnik ist die direkte Implementierung von Refactorings, wie sie zum Beispiel in Eclipse umgesetzt ist. Hier wird das Refactoring mittels Vorbedingung im Rahmen einer integrierten Entwicklungsumgebung durchgeführt.

Ein Problem bei der Vorstellung der verschiedenen Beschreibungstechniken ist, dass meist keine Sammlung von Refactorings verfügbar ist. Die einzelnen Publikationen beschreiben wenige Refactorings, die besonders gut zu den einzelnen Sprachen passen. Aus diesem Grund werden im Folgenden Refactorings aus den Veröffentlichungen genutzt und kein durchgängiges Beispiel gewählt.

5.1.1 Natürlichsprachliche Beschreibung

Eine einfache und durch die Arbeiten von Fowler [Fow04] weit verbreitete Art der Beschreibung von Refactorings ist die natürlichsprachliche Beschreibung. Der Vorteil dieser Methode liegt in der Universalität. Die Technik ist nicht auf bestimmte Programmier- oder Spezifikationssprachen eingeschränkt und allgemein verständlich. In einem ersten Teil der Beschreibung (Tabelle 5.1) werden die Voraussetzungen für das Refactoring aufgezählt. Anschließend werden in einzelnen Schritten die Aktionen abstrakt beschrieben (z.B. „Fügen Sie eine neue Klasse ein und machen Sie diese Klasse zur Unterklasse Ihrer gegeben Klasse“). Soll auf ein anderes Refactoring zurückgegriffen werden, wird es mit Hinweisen auf den Ort der Refactoringanwendung und der Häufigkeit der Anwendung beschrieben. Das Verständnis wird häufig durch abstrakte UML-(Klassen)-Diagramme vereinfacht.

5.1.2 Vor- und Nachbedingung nach Roberts

Roberts führt in seiner Dissertation [Rob99] eine formale Beschreibung von Refactorings ein. Nach seiner Definition ist ein Refactoring eine parametrisierte Transformation, welche zusätzlich eine Vorbedingung (*pre*) enthält, die erfüllt sein muss, bevor das Refactoring durchgeführt werden darf. Das Ergebnis des Refactorings wird bei Roberts mit einer Nachbedingung (*post*) beschrieben (Tabelle 5.2). Alle Umformungen, die die Nachbedingung erfüllen, sind gültige Durchführungen dieses Refactorings. Um die Vor- bzw. Nachbedingung ausdrücken zu können, benutzt Roberts semantische Hilfsfunktionen, die frei von Seiteneffekten sein müssen. Roberts nennt sie deshalb *Analysefunktionen*. Die Funktion *IsClass* testet z.B., ob ein übergebenes Objekt eine Klasse ist. Ein anderes Beispiel ist *InstanceVariablesDefinedBy*, welches die Instanzvariable einer Klasse selektiert.

Das Beispiel in Tabelle 5.2 zeigt das Verschieben einer Methode. Die Parameter des Refactorings sind alle implizit vom Typ String, so dass für den Parameter *class* sichergestellt werden muss, das dieser eine Klasse referenziert. Entsprechend wird geprüft, ob die Operation *selector* in der Klasse existiert (DefinesSelector(*class*, *selector*)). Die Methode soll in die Klasse verschoben werden, deren Typ der Zielvariablen *destvar* entspricht. Das Verschieben erfolgt somit aus der Klasse in eine über die Instanzvariablen *destvar* erreichbare Klasse. In dieser Klasse und deren Unterklassen darf es noch keinen Klassenbestandteil (Methode oder Variable) mit dem selben Namen (*newSelector*) geben:

$$\forall\, c \in \text{ClassOf}(class, destVar).\neg\text{UnderstandsSelector}(c, newSelector)$$

Der Rest der Vorbedingung stellt sicher, dass in der Methode keine Instanzvariablen aus der Ausgangsklasse genutzt werden.

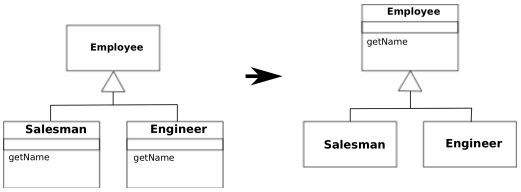
Die Nachbedingung beschreibt anschließend in ähnlicher Weise die refaktorisierte Lösung. Hierbei beschreiben die gestrichenen Variablen (Method′) einen Wert nach dem Refactoring. Die erste Bedingung beschreibt, dass in der Klasse, die dem Typ der Zielvariablen entspricht, die Methode existiert und der alten Methode entsprechen muss. Die zweite Bedingung beschreibt die Veränderung der Referenzen auf die Methode. Die Referenzen müssen immer die neue Instanzvariable nutzen, die die Klasse mit der verschobenen Methode enthält.

5.1.3 OCL / OCL-Script / QVT

UML benutzt eine MOF-Struktur [MOF02] (vgl. Kapitel 3), um die Syntax der Sprache abzubilden. Da auf den MOF-Schichten die Object Constraint Language (OCL) [KW00]

Beispiel: Methode nach oben verschieben nach [Fow04]

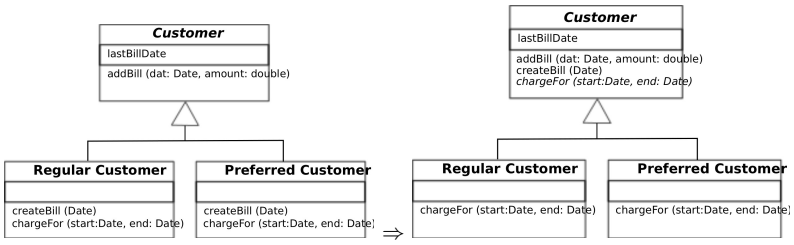
Sie haben Methoden mit identischen Ergebnissen in verschiedenen Unterklassen. Verschieben Sie sie in die Oberklasse.



Vorgehen:

- Untersuchen Sie die Methoden, um sicherzustellen, dass sie identisch sind.
  => Wenn es so scheint, dass die Methoden das Gleiche leisten, aber nicht identisch sind, wenden Sie Algorithmus ersetzen auf eine Methode an, um sie identisch zu machen.
- Wenn die Methoden verschiedene Signaturen haben, ändern Sie die Signatur in der Methode, die Sie in der Oberklasse verwenden wollen.
- Erstellen Sie eine neue Methode in der Oberklasse, kopieren Sie den Rumpf einer der Methoden in diese Methode, passen Sie sie an und wandeln sie um.
  => Wenn Sie in einer streng typisierten Sprache arbeiten, und die Methode eine andere aufruft, die in beiden Unterklassen, nicht aber in der Oberklasse vorkommt, deklarieren Sie ein abstrakte Methode in der Oberklasse.
  => Falls die Methode ein Feld der Unterklasse verwendet, so verwenden Sie Feld nach oben verschieben oder eigenes Feld kapseln, deklarieren und verwenden Sie eine abstrakte set-Methode.
- Löschen Sie die Methode in einer der Unterklassen.
- Wandeln Sie um und testen Sie.
- Fahren Sie fort, die Methoden in den Unterklassen zu entfernen und zu testen, bis nur die Methode in der Oberklasse übrig bleibt.
- Untersuchen Sie die Aufrufer, ob Sie deren geforderten Typ in den Typ der Oberklasse ändern können.

Beispiel Wir betrachten eine Klasse Customer (Kunde) mit zwei Unterklassen: RegularCustomer und PreferredCustomer.



Die Methode createBill ist für beide Klassen identisch:

```
 void createBill (Date date) {
      double chargeFor = charge (lastBillDate, date);
      addBill (date, charge);
}
```

Ich kann die Methode nicht einfach in die Oberklasse verschieben, da chargeFor in jeder Unterklasse anders ist. Als erstes deklariere ich sie in der Oberklasse als abstrakt:

```
class Customer ...
   abstract double chargeFor(Date start, Date end);
```

Nun kann ich createBill aus einer Unterklasse kopieren. Danach wandle ich um und entferne anschließend createBill aus einer der Unterklassen, wandle um und teste. Dann entferne ich sie aus der anderen, wandle um und teste.

Tabelle 5.1: Ein Refactoring nach Fowler [Fow04]

MoveMethod($class, selector, destVar, newSelector$)	
pre :	IsClass($class$) $\wedge$DefinesSelector($class, selector$) $\wedge \forall\, c \in$ ClassOf($class, destVar$).$\neg$UnderstandsSelector($c, newSelector$) $\forall\, c \in$ superclass*($class$) $\cup \{class\}. \forall\, v \in$ InstanceVariablesDefinedBy(c). ($class, selector$) $\notin$ ReferenceToInstaceVariable(c, v)
post :	Method$'$ = $\forall\, c \in$ ClassOf($class, destVar$). Method[($class, selector$)] $\setminus$ ForwarderMethod($newSelector, dfestVar$)] [($c, newSelector$) $\setminus$ MovedMethod(Method($class, selector$))] $\forall\, e \in \{d \mid$ IsClass(d) $\cap$ UnboundVariable(Method($class, selector$)). ClassReference$'$ = ClassReferences [$e \setminus$ [$ClassReferences(e) - \{(class, selector)\} \cup \{(c, newSelector)\}$]

Tabelle 5.2: Ein Refactoring nach Roberts [Rob99]

genutzt werden kann, bietet es sich an, OCL auch für die Beschreibung von Refactorings zu nutzen. Oft finden Erweiterungen von OCL wie z.B. OCL-Script, bei der OCL um Operationen erweitert wurde, die das Modell ändern können, Anwendung [CW04]. In Abbildung 5.3 ist zu sehen, dass sich diese Refactorings nahe an die Arbeit von Roberts [Rob99] halten und eine Vorbedingung und eine Nachbedingung definieren. Im Beispiel (Abbildung 5.3) wird ein Refactoring definiert, welches im Code die Nutzung des Ausdruckes durch die Nutzung einer Variable ersetzt, die in ihrer Initialisierung den Ausdruck verwendet. Das Refactoring nimmt zwei Parameter, welche den Ausdruck (`sourceExpression`) bzw. die Variable (`variable`) enthalten. Die Vorbedingung wird durch zwei Bedingungen ausgedrückt. In der ersten Bedingung wird durch einen Subgraphenvergleich sichergestellt, dass die Variable auf den Wert des dem Refactoring übergebenen Ausdruck initialisiert wird:

```
pre: sourceExpression.areOclSubGraphsEquivalent (variable.initExpression)
```

Die zweite Bedingung prüft, ob die Variable an dem Ort des zu ersetzenden Ausdruckes sichtbar ist:

```
pre: sourceExpression.variablesInScope()->includes(variable)
```

Der Effekt wird durch eine Postcondition beschrieben (`Post:`), was etwas verwirrend ist, da diese die notwendigen Änderungen explizit durchführt. In der Postcondition wird ein Teilgraph, der die Variable referenziert, aufgebaut und alle Verweise auf die `sourceExpression` auf diese `VariableRefExpression` umgeleitet. Anschließend wird der Teilgraph der alten Expression gelöscht.

Eine andere OCL Erweiterung ist das *Query View Transformation* Framework [Kur08] für MOF. Dies erlaubt allgemeine Transformationen des Modells. Für QVT gilt ähnliches wie für OCL-Script. Dabei ist QVT stärker als OCL-Script in MOF eingebettet. Als Besonderheit bietet QVT sowohl eine imperative als auch eine deklarative Erweiterung von OCL.

5.1.4 Graphtransformationen

Programme und Modelle können als Graphen dargestellt werden (z.B. als AST oder in UML). Es liegt die Möglichkeit nahe, Graphtransformationen für Refactorings [MVEDJ05]

```
 context RefactoringComponent::ReplaceExpressionByVariable
 (sourceExpression : OclExpression, variable : VariableDeclaration)
-- sourceExpression is the root of a sub-graph that will be replaced by
-- a variable expression
-- variable is a node corresponding to the Variable Declaration that
-- will be linked to a new  variable expression
-- sourceExpression must have a sub-graph equivalent to the one
-- referred by the target  variable declaration. areOclSubGraphsEquivalent is
-- a query operation that evaluates
-- equivalence between OCL expressions
pre:sourceExpression.areOclSubGraphsEquivalent (variable.initExpression)
-- variable must be visible in the source expression. variablesInScope is
-- a query operation that returns all variables visible to
-- an OCL expression.
pre: sourceExpression.variablesInScope()->includes(variable)
post:
let ownedElements : Set(OclModelElement) =
sourceExpression.owner@pre.ownedElements,
newVarExp : VariableExp = ownedElements->any(element |
element.oclIsTypeOf(VariableExp) and
element.oclIsNew()).oclAsType(VariableExp) in
newVarExp.referredVariable = variable and
ownedElements->excludes(sourceExpression)
actions:
varExp : VariableExp; -- variable definition
-- creates an instance of VariableExp, a link between this instance and the
-- variable parameter, and assigns to varExp variable a reference to the
-- created instance
varExp := new VariableExp(referredVariable => variable);
-- connects the variable expression to the owner node and disconnects
--  sourceExpression from it. ReplaceLink is a polimorphic operation since each
-- expression type has  specific associations.
sourceExpression.owner.replaceLink(sourceExpression, varExp);
-- deletes the source expression from the graph
sourceExpression.deleteOclSubGraph();
```

Tabelle 5.3: Ein Refactoring in OCL-Script [CW04]

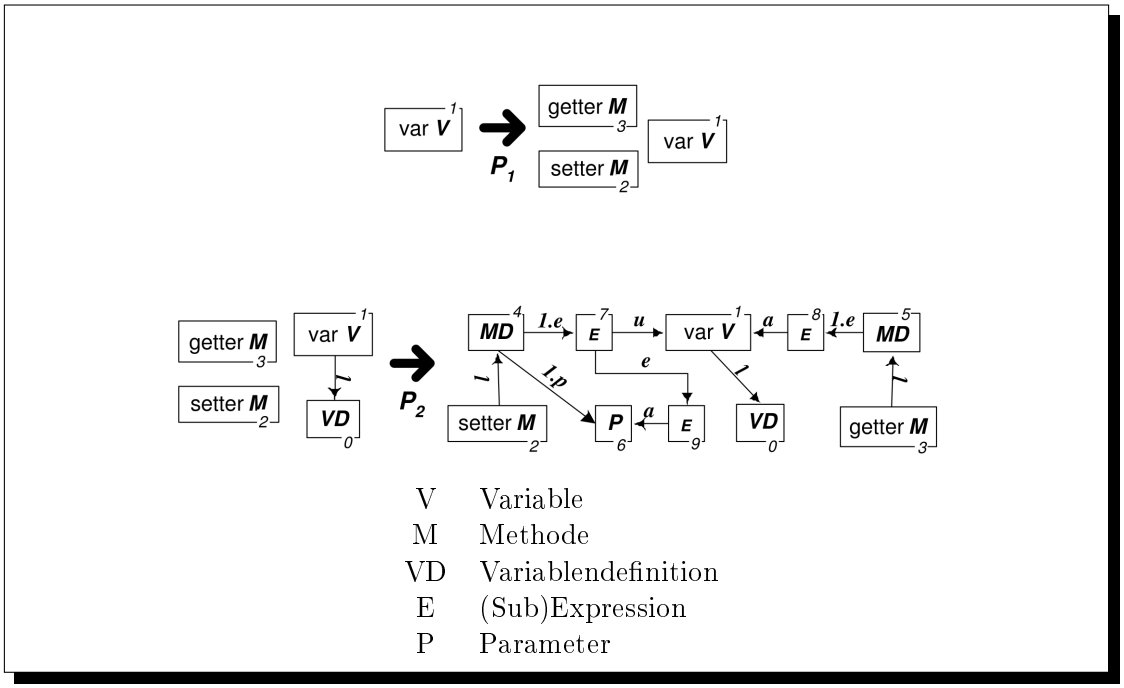


Abbildung 5.1: Zwei Graphtransformationsregeln aus [MVEDJ05], die Teil des größeren Refactorings *Variable Kapseln* sind. Die Zahlen an den Knoten dienen der Zuordnung der Knoten der „linken Seite“ zur „rechten Seite“.

zu verwenden. Graphtransformationssysteme ersetzen Teilgraphen, in dem sie Graphersetzungsregeln anwenden. Solch eine Regel besteht aus einer linken Seite, dem zu ersetzenden Teilgraphen und einer rechten Seite, dem ersetzenden Teilgraphen. Eine Regel kann angewandt werden, wenn für die Regel ein Match gefunden werden kann, das heißt die linke Seite ist isomorph zu einem Teilgraph. Mithilfe einer *Nichtanwendbarkeitsregel* können zusätzliche Bedingungen definiert werden, an denen die Regel trotz eines Matches nicht ausgeführt werden darf. Anschließend wird der Graph so umgebaut, dass der Teilgraph der rechten Seite der Regel entspricht.

In Abbildung 5.1 sind zwei Graphtransformationen P_1 und P_2 zu sehen mit denen in [MVEDJ05] ein Teil des Refactorings „Variable Kapseln“ umgesetzt wird. Die linke Seite wird von der Rechten durch den Pfeil getrennt. Die Knoten haben einen Knotentyp, der in Fettschrift angegeben ist. So ist z.B. ein Knoten vom Typ **M** ein Methoden-Knoten und **V** stellt ein Variablen-Knoten dar. Die erste Regel P_1 sorgt dafür, dass die mit dem Aufruf übergebenen „Setter“ und „Getter“ dem Graphen hinzugefügt werden. Die zweite Transformation P_2 erzeugt die notwendigen Verbindungen zwischen den neuen Methoden und der Variable. Weitere Regeln können dann genutzt werden, um die Nutzung der Variablen durch die neuen „Setter“ und „Getter“ zu ersetzen. Graphtransformationen werden häufig und sehr erfolgreich bei Refactorings von graphischen Modellen angewandt [MEDJ05, EJ04].

5.1.5 JunGL

Ein anderer Ansatz als die bisher vorgestellten Beschreibungen wird bei JunGL [Ver08] verfolgt. JunGL ist eine auf der funktionalen Sprache ML aufbauende Refactoringskript-

```
using CSharp.Ast , CSharp.Binding , CSharp.Flow
{
namespace CSharp.ExtractMethod
{
(*checksforawell-definedregion *)
let dominates entryNode startNode endNode =
( startNode == endNode ) or
Utils.isEmpty { ( ) | [ entryNode ] ( local ? z : cfsucc[?z] & ? z != startNode) * [endNode] }

let postDominates startNode endNode exitNode =
( startNode == endNode ) or
Utils.isEmpty {( ) | [ startNode] ( local ? z : cfsucc[?z] & ? z != endNode ) * [exitNode] }

let haveSameParent x y =
not Utils.isEmpty { ( ) | [ x ] parent ; child [ y ] }

(* transformation code *)
let createVoid TypeRef ( ) =
new TypeRef {
path = new NamespacePath {
entityRef = new EntityRef { name = " void " }
}
}

let createParamDecl name typeRef direction =
new ParamDecl {
name = name , typeRef = typeRef , direction = direction
}

let createArgname direction =

... ca. 240 Zeilen weiterer Code
```

Tabelle 5.4: Ausschnitt eines Refactorings in JunGL[Ver08]

sprache, die für den Einsatz in Visual Studio entworfen wurde. Im Grunde handelt es sich also um eine normale Programmiersprache, die für die Beschreibung von Refactorings erweitert wurde. Das Hauptaugenmerk bei dieser Sprache liegt auf der guten Einbettung in die Sprachsysteme von Visual Studio und die freie Beschreibbarkeit von Refactorings. JunGL wurde direkt aus ML abgeleitet, wobei die meisten Erneuerungen in einem Framework zum Zugriff und der Veränderung des Abstract Syntax Trees liegen. Die Sprache sieht keine explizite Strukturierung der Refactorings nach einem Schema vor. Im Beispiel aus Abbildung 5.4 ist zu sehen, dass alle Fälle durch kleine Funktionen abgedeckt werden. Die Beschreibung eines Refactorings auf diese Weise ist in der Regel um ein Vielfaches umfangreicher als die bisher vorgestellten Methoden. Da das Beispiel, obwohl es sich um ein einfaches Refactoring handelt, bereis umfangreich und dennoch nicht vollständig dargestellt ist, wird auf die Beschreibung verzichtet.

5.1.6 AST-Rewriting - Eclipse

Eine weit verbreitete Entwicklungsumgebung ist die Eclipse IDE, welche einige Refactorings anbietet. Dabei sind die meisten Refactorings nur für Java verfügbar. Die Implementierung in Eclipse teilt sich in zwei Bereiche: ein allgemeines Framework und einen sprachenspezifischen Teil. Ähnlich wie das JunGL Framework stellt das Framework in Eclipse grundlegende Funktionen bereit. Dieses beinhaltet neben simplen AST Operationen auch die GUI-Schnittstelle. Die eigentlichen Refactorings sind im sprachenspezifischen

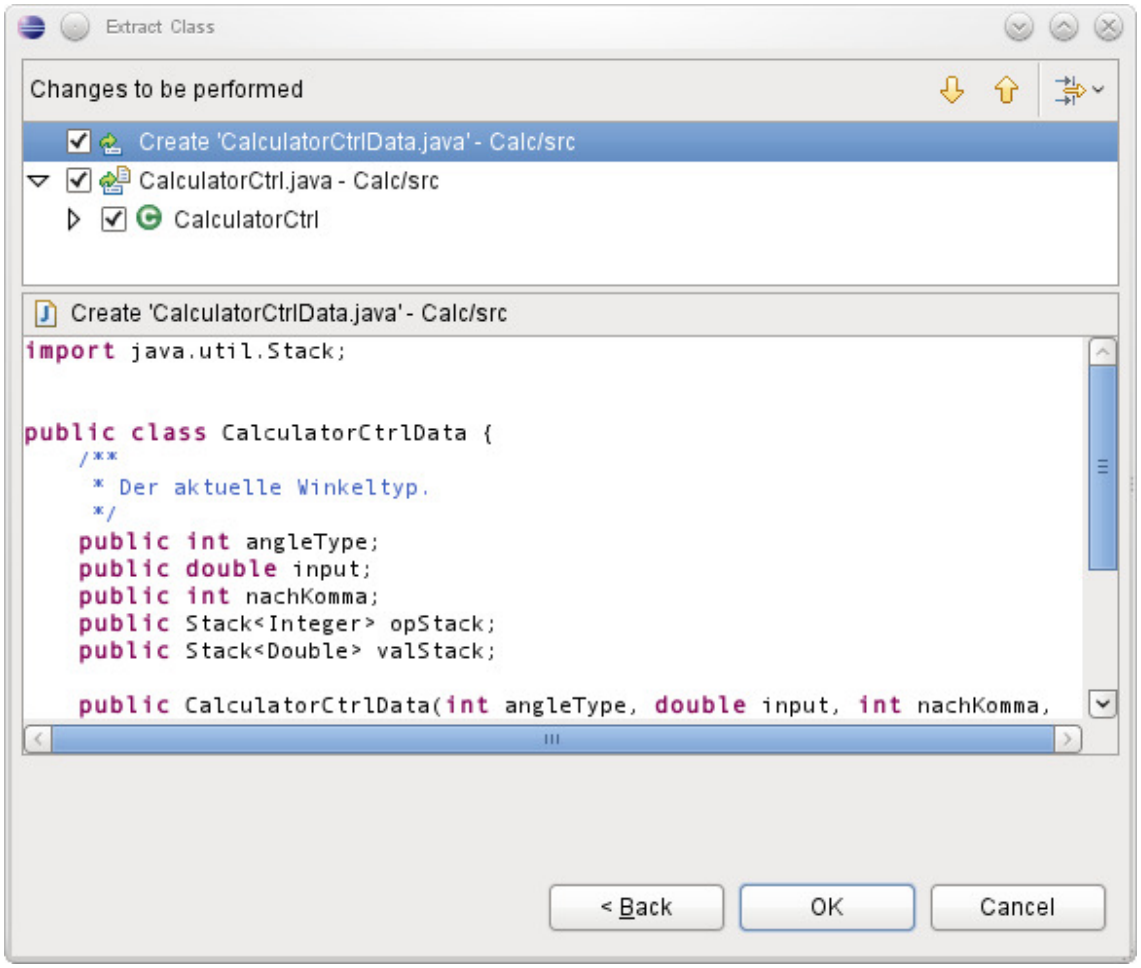


Abbildung 5.2: Reviewfenster, welches von Eclipse angezeigt wird, wenn der Programmierer die Codeänderungen verifizieren soll

Teil definiert. Dort wird jedes Refactoring mit Hilfe des Frameworks implementiert. Für diese Implementation wird Java verwendet, da dies eine universell einsetzbare Programmiersprache ist. Da die Refactorings groß und kompliziert zu verstehen sind, wird hier auf ein Beispiel verzichtet.

Die Datenstruktur, auf der Eclipse arbeitet, ist ein AST. Die Veränderungen an diesem AST werden durch das Memento-Pattern [GHJV85] gesteuert, d.h. es werden Veränderungen erzeugt, die explizit wieder rückgängig gemacht werden können. Dies wird in Eclipse auch genutzt, da in einigen Fällen die Verhaltenserhaltung nicht geprüft wird und dann dem Benutzer diese Prüfung obliegt (vgl. Abbildung 5.2). Wenn der Benutzer die Änderungen ablehnt, werden diese mittels Memento-Pattern wieder rückgängig gemacht.

5.2 Anforderungen an eine Beschreibungssprache für Refactorings

In dem vorherigen Unterkapitel wurden einige Möglichkeiten vorgestellt, Refactorings zu beschreiben. Sie sind alle mit unterschiedlichen Zielsetzungen entwickelt worden. So hat Fowler [Fow04] ein Lehrbuch geschrieben mit der Anforderung, eine möglichst einfache Beschreibung zu nutzen, die von möglichst vielen Programmierern verstanden werden kann und für Refactorings in vielen verschiedenen Programmiersprachen brauchbar ist. Roberts [Rob99] brauchte für seine Dissertation eine formale Ausgangsbasis, in der es nicht wichtig war, wie ein Refactoring ausgeführt wird. Es war dagegen wichtig zu wissen, was nach dem Refactoring gilt. Alle vorgestellten Ansätze lassen sich in drei Gruppen einteilen:

- Sprachen zur Kommunikation zwischen Entwicklern (natürlichsprachliche Texte, Beispiele)
- Sprachen zur formalen Analyse (Roberts)
- Sprachen zur Implementierung (OCL, AST-Rewriting)

Aus diesen Anwendungsgruppen von Refactorings ergeben sich verschiedene Anforderungen, die ein Refactoring erfüllen soll. Im Folgenden sollen die Anforderungen an Refactorings betrachtet werden, die für diese Arbeit interessant sind. Wie gut die Beschreibungstechniken diese Anforderungen erfüllen, wird anschließend analysiert. Der Zweck dieser Analyse ist es, eine Sprache zu finden, die den Anforderungen für die Beweistechniken dieser Arbeit entspricht. Es sollen Refactorings mit mehreren Sichten analysiert werden. Als Versuchssprache wird dabei CSP-OZ verwendet. CSP-OZ ist eine integrierte formale Methode, die sowohl eine statische Sicht als auch eine dynamische Sicht umfasst. Als formale Methode kann diese nicht ausgeführt werden und ist deshalb nicht für Tests zugänglich, weshalb formale Analysen benötigt werden. Da es zu kompliziert ist, die Refactorings per Hand durchzuführen, soll für Fallstudien eine automatische Ausführung der Refactorings möglich sein (Implementierung). Da anschließend aus den Refactorings Schlussfolgerungen für allgemeine mehrsichtige Formalismen gezogen werden sollen, soll sie auch den Anforderungen für die Kommunikation zwischen Entwicklern genügen. Den folgenden Bewertungskriterien sind Bewertungsgrundsätze beigefügt, die ein Bewertungsspektrum von sehr gut ($++$) über ausgeglichen ($\circ$) bis unzureichend ($--$) abdecken.

Verständlichkeit Um einen Text zu verstehen, müssen verschiedene Voraussetzungen erfüllt werden, zum Beispiel sollten die Worte der Sprache sowie die Struktur bekannt sein. Ähnlich ist es mit der Beschreibung von Refactorings. Ein Nutzer oder Entwickler eines Refactorings muss die Sprache verstehen können. Eine wichtige Frage dabei ist, welches Wissen von einem Nutzer oder Entwickler des Refactorings erwartet werden kann. Unabhängig von den anderen Voraussetzungen kann sicherlich angenommen werden, dass der Entwickler und Nutzer des Refactorings die Sprache, auf der das Refactoring ausgeführt werden soll, versteht. Deshalb ist jeder Formalismus, der *nicht* syntaktisch und semantisch nahe der Zielsprache ist, als negativ zu bewerten. Ein Sonderfall ist die natürlichsprachliche Beschreibung, weil sie keine anderen Mechanismen als die Ziel-Sprache und eine Anleitung enthält. Diese kann im Allgemeinen als verständlich angesehen werden. Ein Ansatz, der eine Sprache oder einen Formalismus nutzt, welche nichts mit der Zielsprache gemein haben, ist im Allgemeinen als unverständlich anzusehen. Der Nutzer bzw. Entwickler des

Refactorings muss erst den Formalismus oder die Sprache erlernen, um das Refactoring zu verstehen, zu erstellen oder anzuwenden.

Für die Bewertung der Verständlichkeit soll die Menge der Formalismen dienen, die ein Entwickler eines Refactorings erlernen muss, wenn er nur die Zielsprache kennt. Die Verständlichkeit ist groß (d.h. gut +), wenn nur wenige Techniken gelernt werden müssen. Je mehr gelernt werden muss, je schlechter ist die Bewertung.

Genauigkeit der Beschreibung Ein Refactoring soll eindeutig beschrieben werden. Es sollen keine Mehrdeutigkeiten existieren. Besonders Unklarheiten über die Ausführung von einzelnen Schritten können zu Problemen führen, da die richtige (also eigentlich gemeinte) Ausführung das Verhalten erhält, aber eine andere diese Eigenschaft verletzt. Die Ausführung muss daher genau erklären was passiert.

Die Bewertung orientiert sich an der Genauigkeit der Beschreibung. Des Weiteren geht eine gut an Refactorings angepasste Beschreibung positiv in die Bewertung ein. Eine genaue, eindeutige Folge von Anweisungen ist gut (+). Am besten (++) ist es wenn die Beschreibung durch einen Computer ausführbar ist. Eine prüfbare Nachbedingung (zum Beispiel als assert) ist ausreichend (○). Eine Benutzung einer allgemeinen Programmiersprache wird weniger gut (–) eingestuft, da diese offen lässt, ob alle Fälle und Besonderheiten codiert wurden. Als schlecht ist eine offene Beschreibung zu werten, die nicht eindeutig beschreibt, wie das Refactoring durchgeführt werden soll und dadurch Interpretationsspielraum lässt (– –).

Syntaktische Korrektheit Die Korrektheit eines Refactorings setzt sich aus mehreren Kriterien zusammen. Das erste Kriterium ist die syntaktische Korrektheit: Code soll nach dem Refactoring syntaktisch korrekt sein. Das heißt, dass ein Refactoring für alle syntaktisch korrekten Eingaben syntaktisch korrekten Code liefern soll. Hierfür ist es notwendig, dass an der Beschreibung der Ausführung eines Refactorings geprüft werden kann, ob der erzeugte Code syntaktisch korrekt ist. Die Korrektheit der Syntax soll nach dieser Kontrolle implizit durch das Refactoring sichergestellt sein.

Wenn eine Sprache es ermöglicht, anhand der Beschreibung des Refactoring zu prüfen, ob der erzeugte Code syntaktisch korrekt sein wird, wird dies hier als gut (+) bewertet. Eine Beschreibung, die nach ausführlichen Tests syntaktisch korrekten Code zuverlässig erzeugt, wird als brauchbar (○) bewertet. Als schlecht (–) wird eingestuft, wenn die syntaktische Korrektheit nicht sichergestellt werden kann.

Typ-Korrektheit/Wohlgeformtheit Während die syntaktische Korrektheit bei Refactorings oft einfach zu zeigen ist, weil Refactorings meist auf einer syntaktischen Ebene definiert sind, ist die Typ-Korrektheit bzw. Wohlgeformtheit eine Eigenschaft, deren Prüfung aufwendig ist. Ein Refactoring sollte sicherstellen, dass die Nebenbedingungen an einen Code (z.B. alle Variablen müssen deklariert sein) erfüllt werden. Ähnlich wie bei der syntaktischen Korrektheit wird hier gefordert, dass diese Bedingungen an der Refactoringbeschreibung geprüft werden können und diese dann durch das Refactoring sichergestellt werden.
Die Bewertung entspricht der Bewertung der syntaktischen Korrektheit.

Verhaltenserhaltung Ein wesentliches Merkmal von Refactorings ist, dass sie das Verhalten des Codes erhalten sollen. Das Verhalten kann durch ein Refactoring erhalten wer-

den oder durch eine Nachkontrolle kann geprüft werden, ob das Verhalten erhalten wurde. Wenn ein Refactoring das Verhalten erhält, kann dies durch einen Beweis sichergestellt sein oder es wurde aufgrund vieler Tests eine vertrauenswürdige Basis geschaffen. Auch die Nachkontrolle kann durch einen Beweis oder durch Tests geschehen.

Ein formaler Beweis der Verhaltenserhaltung wird hier als besonders gut (++) bewertet. Eine plausible Verhaltenserhaltung durch Erfahrung und Tests anhand der Refactoringbeschreibung wird als brauchbar (○) angesehen. Genauso bewertet wird die formale Nachkontrolle, bei der mittels Beweisen die Verhaltenserhaltung nach dem Refactoring gezeigt wird. Als schlecht (–) wird die Kontrolle ausschließlich mittels Tests nach der Durchführung des Refactorings angesehen.

Universelle Anwendbarkeit Die Beschreibungssprache soll möglichst nicht nur für eine Sprache geeignet sein. Diese Eigenschaft ermöglicht es, eine Refactoringsprache für verschiedene Sprachen zu verwenden. Somit brauchen Refactoringtools nicht für jede Sprache neu entwickelt zu werden, sondern es kann eine Umsetzung für verschiedene Sprachen verwendet werden. Im Optimalfall (++) ist eine Beschreibungssprache sowohl allgemein für graphische als auch für Text-Sprachen geeignet. Der negative Fall (–) ist, dass eine Sprache nur einen engen Einsatzbereich hat.

Automatisierung Refactorings werden nur selten per Hand ausgeführt. Ein Entwickler möchte Refactorings (teil-)automatisiert durchführen. Eine Refactoringsbeschreibung, und somit das Refactoring, sollte durch eine integrierte Entwicklungsumgebung (IDE) ausgeführt werden können. Auch die Skalierbarkeit des Algorithmus ist an dieser Stelle wichtig. Ein Refactoring sollte bei großen Software-Systemen schnell durchgeführt werden, um den Arbeitsfluss des Entwicklers nicht zu stören.

Damit sind die Kriterien, nach dem die Ansätze aus Abschnitt 5.1 bewertet werden sollen, aufgezählt. Wie zuvor schon erläutert wurde, sind die Kriterien für die verschiedenen Einsatzzwecke von Refactoringsprachen unterschiedlich wichtig. Eine Zuordnung zu den Einsatzzwecken findet sich in Tabelle 5.6. So ist die IDE-Eignung für den Einsatz in der Kommunikation und in der Analyse nicht wichtig, aber für die Implementierung nahezu unentbehrlich.

5.3 Vergleich der Ansätze

Verschiedene Refactoring-Sprachen verfolgen verschiedene Ziele. In diesem Abschnitt werden die Ziele und die Sprachen mit Hilfe der Kriterien aus dem vorherigen Abschnitt verglichen und bewertet. Die Bewertung der Sprachen ist in Tabelle 5.5 dargestellt. Die Ergebnisse in der Tabelle nutzen die Kriterien die im vorherigen Abschnitt beschrieben wurden. Zum Vergleich sind in Tabelle 5.6 die Anforderungen für die Einsatzzwecke von Refactorings beschrieben. Die einzelnen Sprachen passen gut zu ihren Einsatzgebieten. Ein Problem ist es, eine Sprache zu finden, die für mehrere Einsatzgebiete geeignet ist. So fällt z.B. auf, dass bei Beschreibungstechniken, die für den Einsatz in IDEs entwickelt wurden, die Verständlichkeit und die Beweisbarkeit der Verhaltenserhaltung nicht ausgeprägt sind. Beide Eigenschaften werden in diesem Kontext als nicht besonders wichtig erachtet.

Einige der Bewertungen in der Tabelle sollen genauer betrachtet werden. Die Bewertung der Sprachunabhängigkeit der natürlichsprachlichen Beschreibung ist ungenügend bewer-

tet, weil die Beschreibungen meist auf Java eingehen, aber spezielle Fälle von anderen Sprachen nicht betrachten. So ist die Mehrfachvererbung (C++) nicht behandelt. Damit ist die Beschreibung zwar von der Idee her übertragbar, aber nicht ohne weitere Überlegungen. Bei den Graphtransformationen ist die Bewertung der Verständlichkeit schlechter, als von der Graphtransformations-Community beschrieben. Das hat hier den Grund, dass BNF-basierte Sprachen betrachtet werden. Graphtransformationen passen sehr gut zu Graphen und sind in diesem Zusammenhang sehr gut verständlich. In dem in dieser Arbeit betrachteten Fall liegt mit dem AST zwar eine passende Präsentation vor, aber die Graphtransformationen auf einem AST sind umständlich, da viele Nutzungsbeziehungen nicht explizit dargestellt sind. Für deren Anwendung müsste eine Graphendarstellung gewählt werden, in denen allen Nutzungbeziehungen explizit angegeben werden. Ein solcher annotierter AST steht für viele Sprachen zur Verfügung, hat aber nur noch wenig mit der Darstellung des Textes der Sprache gemein und muss separat erlernt werden. Da das Verständnismass als die Menge der Techniken, die gelernt werden müssen, definiert wurde, schneiden die Graphtransformationen schlecht ab.

Auf der anderen Seite wird durch die Graphtransformationen nicht die spezielle Struktur einer text-basierten Sprache berücksichtigt. Somit ist die Anwendung von Graphtransformationen mit einem gewissen Overhead versehen. Details zu diesem Thema finden sich im Vergleich der Ansätze in Abschnitt 5.11.2.

Folgerungen Da im zweiten Teil dieser Arbeit eine Beschreibungssprache gebraucht wird, die alle drei Aufgabenbereiche abdeckt, muss hier festgestellt werden, dass es keine Beschreibungstechnik für Refactorings gibt, die den Anforderungen dieser Arbeit entspricht. Deshalb wird in den nächsten Unterkapiteln eine neue Sprache mit Namen Re$\mathcal{L}$ entwickelt, die den Anforderungen genügt.

Die betrachteten Sprachen weisen jeweils einen Teilbereich der geforderten Eigenschaften auf. Es bietet sich daher an, erfolgreiche Konzepte aus diesen Sprachen als Ausgangspunkt für die Entwicklung von Re$\mathcal{L}$ zu verwenden. In diesem Sinn ist Re$\mathcal{L}$ besonders von den Arbeiten von Roberts und den Arbeiten zu Graphtransformationen beeinflusst. Im Weiteren soll erst die neue Sprache vorgestellt werden. Ein Vergleich der verwendeten Konstrukte und wie sie sich in den Sprachen verhalten, findet sich in Abschnitt 5.11.

5.4 Refactoring-Beschreibungssprache Re$\mathcal{L}$

Bei den Refactorings aus dem vorherigen Abschnitt sieht man, dass es einige Gemeinsamkeiten in den Beschreibungstechniken gibt. Aus erprobten Gemeinsamkeiten werden in diesem Kapitel Eigenschaften für die neue Sprachfamilie Re$\mathcal{L}$ abgeleitet.

Die auffälligste Gemeinsamkeit ist, dass erst die Struktur vor dem Refactoring und anschließend die Durchführung oder der Zustand nach dem Refactoring definiert wird. Die Grundlage für diese Struktur hat Roberts [Rob99] in seiner Dissertation gelegt und diese ist vielfach adaptiert worden. Sehr deutlich zu sehen ist dieses Vorgehen an der natürlichsprachlichen Beschreibung in Fowler, obwohl die meisten anderen Sprachen auch als Ausgangspunkt dienen könnten.

Bedingungen vor dem Refactoring beschreiben das, was für das Refactoring wichtig ist und lassen alle anderen Codebestandteile unspezifiziert. Das lässt sich anschaulich am Beispiel der deutschen Sprache beschreiben. Sätze, die eine Eigenschaft von etwas beschreiben, können als „x ist y.“ ausgedrückt werden, wobei ein Nomen für x eingesetzt werden kann und

Ansatz	Verständ-lichkeit	Genauig-keit	Universalität	Automatisier-barkeit	Sprach–unabhängigkeit	Syntaktische Korrektheit	Typ-Korrektheit/ Wohlgeformtheit	Beweisbarkeit
Umganssprachlich	++	-	+	– –	–	○	++	– –
Vor/Nachbed.	○	++	++	○	○	++	○	+
OCL-Script	–	+	+	+	+	+	+	– –
QVT	–	+	+	+	+	+	+	– –
Graphtranfoma.	–	+	+	+	+	+	+	– –
JunGL	–	+	+	+	+	+	+	– –
Eclipse	– –	+	+	+ +	– –	+	– –	– –

Tabelle 5.5: Vergleich existierender Refactoringssprachen

Ansatz	Verständ-lichkeit	Genauig-keit	Universalität	Automatisier-barkeit	Sprach–unabhängigkeit	Syntaktische Korrektheit	Typ-Korrektheit/ Wohlgeformtheit	Beweisbarkeit
Kommunikation	++	○	○	–	○	○	○	–
Analyse	○	++	○	–	++	++	++	+
Implementierung	○	++	○	++	++	++	++	–

Tabelle 5.6: Für den Einsatzzweck benötigte Anforderungen

y für ein Adjektiv. Ein Beispiel ist „Peter ist krank.“. Diese Technik wird für die Erstellung von ReL genutzt und die Beschreibung in einer Sprache mit Lücken wird hier *Template* genannt. Die Lücken werden *Meta-Variablen* genannt. Der Zusatz *Meta* macht deutlich, dass es sich nicht um Variablen in der Zielsprache handelt. Im natürlichsprachlichen Beispiel wäre dies nicht nötig, da keine Variablen vorhanden sind. Für Programmiersprachen ist dies dagegen notwendig, wobei Meta-Variablen nicht alle Arten von Satzstücken enthalten können sollen. Wenn im Beispiel für die Meta-Variable y ein Substantiv eingesetzt wird, ergibt das keinen Sinn: „Peter ist Haus.“. Dieses Problem kann behoben werden, in dem die Meta-Variablen mit einem Typ deklariert werden. Der Typ leitet sich dabei von dem syntaktischen Konstrukt der Zielsprache ab (hier: Substantiv für x und Adjektive für y). Die Templates leiten sich von der Syntax der Ausgangssprache ab, die Templates sind somit sprachabhängig. Daraus folgt, das eine allgemeine Refactoringsprache entweder auf einem gemeinsamen Modell aller Sprachen arbeiten muss oder eine sprachspezifische Sprache sein wird. Für ReL wurde der sprachspezifische Ansatz genutzt. Eine Diskussion dieser Entscheidung ist in Abschnitt 5.11.3 zu finden.

Mit den Templates kann nicht nur der Zustand vor einem Refactoring beschrieben werden, sondern auch der Zustand nach einem Refactoring. Ein Template „x ist gerade y.“, das den veränderten Zustand beschreibt, kann aus dem obigen Beispiel mit „Peter ist krank.“ einen neuen Satz „Peter ist gerade krank.“ erzeugen. Dabei sind die Variablen, die durch das erste Template mit einem Wert („Peter“ bzw. „krank“) belegt wurden, hier wieder verwendet und es wird mit der Belegung der neue Satz erzeugt. Um dieser Unterscheidung gerecht zu werden, wird im Folgenden vom *BeforeTemplate* gesprochen, wenn es sich um eine Beschreibung des Ausgangszustandes handelt und um ein *AfterTemplate*, wenn die neue Struktur beschrieben wird. Für die Verwendung in ReL werden die Templates um Wiederholungen erweitert um Aufzählungen abzudecken. So eine Aufzählung könnte dann einen Satz abdecken wie „Peter ist krank, blond und kurzhaarig.“. Die Notation von Wiederholungen wird dabei an die Notation von BNFs angelehnt. Somit gibt es prinzipiell drei Arten von Wiederholungen, die sich durch ihre *Multiplizität* unterscheiden: bis zu einem Mal (ZeroOrOne: ?), beliebig oft (ZeroOrMore: *) und mindestens einmal (OneOrMore: +). Da die Wiederholungen sowohl im Before- als auch im AfterTemplate genutzt werden und eine entsprechende Zuordnung zueinander gewünscht ist, werden die Wiederholungen mit einer ID versehen.

Wenn Templates mit den Beschreibungstechniken aus dem vorherigen Abschnitt verglichen werden, zeigt sich, dass sie einen großen Bereich der Beschreibungen abdecken, aber nicht alles Notwendige erfassen. So werden beispielsweise in den Vorbedingungen von Roberts [Rob99] und Fowler [Fow04] oft zusätzliche Bedingungen gefordert, die sich nicht durch die Templates abbilden lassen. Eine solche Bedingung ist, dass ein neuer Operationsname nicht schon in der Klasse existiert. Er muss also neu sein. So eine Bedingung kann über die Meta-Variablen mittels Prädikatenlogik erster Ordnung ausgedrückt werden. Für die Prädikate werden *Analysefunktionen* zu Hilfe genommen. Diese Funktionen sind auch von Roberts [Rob99] genutzt worden. Sie haben keine Seiteneffekte, verändern also weder den Inhalt der Meta-Variablen noch den Code.

ReL unterstützt die Erstellung einer Sammlung von Refactorings. In diesem *Repository* werden nicht nur die eigentlichen Refactorings gesammelt, sondern zusätzlich Eigenschaften definiert. Dies ermöglicht die Angabe, welches Verhalten durch das entsprechende Refactoring erhalten wird. Das Repository enthält des Weiteren einen Header, in dem unter anderem die Sprache, die mit diesem Repository bearbeitet werden kann, angegeben wird.

Zum Thema der verschiedenen Möglichkeiten der Verhaltenserhaltung siehe Kapitel 7.2.

Nachdem die grundlegende Idee für Re$\mathcal{L}$ beschrieben ist, wird im nächsten Abschnitt ein Refactoring eines FWHILE-Programms betrachtet.

```
Refactoring : Extract Function
Parameter:
    functionname : FNAME ;
    operations   : MSTAT
Preserves InputOutput                                                    5
Before Template:
    Define:
        a,b : MSTAT;
        f : FUNC;
    Index: sub1 Scope: FWHILE                                           10
        main
                a;
                operations;
                b
        endmain                                                         15
                #Ifunc(f)*
Precondition:
    \forall a \in names(f) \bullet a != functionname
        | writevars(operations) \cap readvars(b) | = 1
Calculation:                                                            20
    Define:
            parameterList : PARAM;
                returnvar : VAR;
    Calc:
        parameterList = readvars(operations)                            25
        returnvar \in writevars(operations) \cap readvars(b)
After Template:
    Scope: FWHILE
        main
            a;                                                          30
            returnvar = functionname(parameterList);
            b
        endmain
        #Ifunc(f)*
        func functionname (parameterList)                               35
            operations
            return returnvar;
        endfunc
```

Listing 5.1: Beispiel-Refactoring: Extract Function für While-Programme

5.5 Re$\mathcal{L}_{FWHILE}$ – Aufbau von Re$\mathcal{L}$ am Beispiel eines Refactorings von FWHILE

Bevor in den nächsten Kapiteln die Erstellung einer Instanz von Re$\mathcal{L}$ und die Semantik von Re$\mathcal{L}$ beschrieben werden, sollen in diesem Kapitel die wichtigsten Bestandteile von Re$\mathcal{L}$ am Beispiel eines Refactoring eines FWHILE-Programms (siehe Listing 5.1) erklärt werden. Dabei wird erst die grobe Struktur erklärt, die Details folgen am gleichen Beispiel später. Diese Instantiierung von Re$\mathcal{L}$ wird im Weiteren Re$\mathcal{L}_{FWHILE}$ genannt.

Eine Beschreibung eines Refactorings in Re$\mathcal{L}$ besteht aus fünf Teilen:

Header Im Header werden der Name und die Parameter des Refactorings angegeben. Weitere Eigenschaften wie die Verhaltenserhaltungsbegriffe, die das Refactoring erfüllt, können optional aufgezählt werden.

BeforeTemplate Die Beschreibung der Code-Struktur vor dem Refactoring. Das Before-Template enthält benannte Lücken (Meta-Variablen), die veränderliche Bereiche im Template beschreiben.

Precondition In der Precondition werden Bedingungen ausgedrückt, die vor der Ausführung des Refactorings erfüllt sein müssen. Für die Darstellung der Precondition werden Ausdrücke aus der Prädikatenlogik erster Ordnung verwendet, die in LaTeX gesetzt werden.

Calculation Die Calculation wird genutzt, um entweder Hilfswerte zu berechnen oder aus anderen, einfacheren Refactorings ein zusammengesetztes Refactoring zu bilden.

AfterTemplate Das AfterTemplate beschreibt, wie der Code nach dem Refactoring aussehen soll. Dazu werden die Meta-Variablen genutzt.

Im Weiteren dieses Abschnittes soll ReL ausführlich beschrieben werden. Bevor auf die Bestandteile von ReL eingegangen wird, werden die Meta-Variablen und die damit verwandten Parameter vorgestellt. Beide spielen eine wichtige Rolle in den einzelnen Komponenten einer Refactoringbeschreibung und sollen deshalb vorab erläutert werden.

Meta-Variablen und Parameter Meta-Variablen sind Variablen in der Beschreibung der Refactorings. Der Zusatz *Meta* soll diese Variablen von Variablen der Zielsprache abgrenzen, so dass diese nicht verwechselt werden. Meta-Variablen sind innerhalb von ReL ein grundlegendes Konstrukt, welches in allen Teilen eines Refactorings genutzt wird. Eine Meta-Variable kann Code-Fragmente enthalten, welche sich durch ein Nicht-Terminal der Zielsprachen-BNF erzeugen lassen. So kann eine Meta-Variable vom Typ *Expression* den textuellen Ausdruck „1 + 1“ enthalten. Meta-Variablen sind, nachdem sie in einem Teil der Refactoring Spezifikation eingeführt wurden, bis zum Ende des Refactorings gültig. Eine Meta-Variable, die im BeforeTemplate eingeführt wurde, ist somit im BeforeTemplate, der Precondition, der Calculation und dem AfterTemplate gültig. Eine Meta-Variable aus der Calculation ist dagegen nur in der Calculation und dem AfterTemplate nutzbar. Die Parameter des Refactorings sind Meta-Variablen, deren Wert vor der Ausführung des Refactorings gesetzt wird. Sie sind in allen Teilen des Refactorings gültig. Die Gültigkeit beschränkt sich auf das aktuelle Refactoring; ein aufrufendes Refactoring hat keinen Zugriff auf die Meta-Variablen des aufrufenden Refactorings, außer sie werden als Parameter übergeben.

Meta-Variablen müssen zwingend vor ihrer Benutzung deklariert werden. Solche Deklarationen befinden sich im Beispiel Listing 5.1 in den Zeilen 8 und 9. Der Typ einer Meta-Variablen entspricht den Nichtterminalen der Zielsprache, wobei zusätzliche Mengenoperationen möglich sind. Hier können auch Analysefunktionen verwendet werden (vgl. Kapitel 6).

Die Deklaration kann mit einer Variablen-Invariante ergänzt werden. Die Invariante wird bei einigen Refactorings genutzt, um die Wohlgeformtheit nach dem Refactoring zu gewährleisten. Dies ist beispielsweise bei Typcasting notwendig. Eine Meta-Variablen-Deklaration hat somit folgende Struktur:

```
name ("," name )* ":" type ("{" invariant "}")?
```

Wenn die Invariante immer wahr sein soll, kann diese weggelassen werden. Es gilt daher folgende Äquivalenz:

```
varname : Nichtterminal {true} ≡ varname : Nichtterminal
```

In dem Beispiel in Listing 5.1 sind die Parameter in Zeile 3 und 4 zu finden. Es finden sich dort auch zwei Deklarationsbereiche in Zeile 7–9 bzw. in Zeile 21–23.

5.5.1 Header

Der Header setzt sich aus dem Namen des Refactorings, seinen Parametern und den erfüllten Erhaltungsbegriffen zusammen (Zeilen 1–5 in Listing 5.1). Der Header dient zur Referenzierung innerhalb des Repositories, so dass das Refactoring mittels seiner Signatur (Name, Parametertypen und Erhaltungsbegriff) angesprochen werden kann. Ein Refactoring kann mehrmals im Repository definiert sein. In diesem Fall wird aus dem Repository das erste definierte Vorkommen des Refactorings, dessen Vorbedingung erfüllt ist, genutzt (zu Details siehe Kapitel 5.6). Der Header endet mit einer Liste von den Erhaltungsbegriffen, die das Refactoring erfüllt. Sie werden innerhalb von Re$\mathcal{L}$ als Bezeichner gehandhabt, die bei der Auswahl der möglichen Refactorings genutzt werden. Die möglichen Werte werden im Header des Repositories definiert.

5.5.2 Templates

Templates sind Codestücke der Zielsprache, welche durch Platzhalter (Meta-Variablen), Wiederholungen und weitere Konstrukte zu einer Schablone für den Code werden. Die Templates werden in zwei Arten verwendet: Als BeforeTemplate wird das Template auf den zu ändernden Code eingepasst und das AfterTemplate wird zu neuem Code aufgefüllt. Im Folgenden wird das BeforeTemplate beschrieben. Das AfterTemplate ist syntaktisch eine Teilmenge des BeforeTemplate. Die Unterschiede werden am Ende des Abschnittes erläutert. Das BeforeTemplate (Listing 5.1 Zeilen 6-16)besteht aus drei Teilen: Dem Template-Header (Zeile 6), einem Deklarationsteil (Zeilen 7–9) und dem eigentlichen Template (Zeilen 10–16). Das eigentliche Template kann mehrfach auftreten. Zur sprachlichen Unterscheidung werden diese hier Subtemplates genannt.

Template-Header Der Header besteht aus dem Schlüsselwort „BeforeTemplate“ (bzw. „AfterTemplate“) und zeigt den Start der Templates an.

Deklarationen Das BeforeTemplate wird genutzt, um Teile der zu refaktorisierenden Spezifikation Meta-Variablen zuzuweisen. Die neu eingeführten Meta-Variablen werden in der Deklaration (Listing 5.1 Zeilen 7–9) des Templates angegeben. Die Namen der neu deklarierten Meta-Variablen dürfen nicht mit vorher deklarierten Namen von Meta-Variablen übereinstimmen. Meta-Variablen werden im Abschnitt *Meta-Variablen und Parameter* beschrieben deklariert. Diese Deklarationen gelten immer für alle Subtemplates. In Ergänzung zu den Meta-Variablen, wie sie oben beschrieben wurden, kann im BeforeTemplate angegeben werden, wie sich die Meta-Variablen im Bezug zu Wiederholungen verhalten. Dies wird, nachdem die Wiederholungen eingeführt worden sind, behandelt.

Subtemplate Das Subtemplate besteht aus einem mit Meta-Variablen und Wiederholungen erweiterten Code der Zielsprache. Dieses Codestück muss zu einem Teil des Codes der Zielsprache passen. Durch den Scope wird aufgegeben welchen Teil der Zielsprache des Subtemplates entspricht. Bei einem Klassen-Scope werden durch das Template Klassen und bei einem Methoden-Schema werde Methoden angesprochen. Prinzipiell kann als

```
Define:
   x : Production
Index: I1 Scope: Production2
    x
    #I2("a" x)+
```

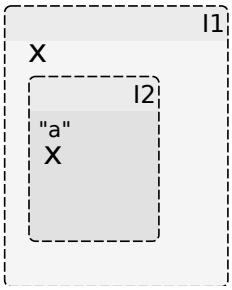


Abbildung 5.3: Schachtelung von Wiederholungen: Gegenüberstellung eines Subtemplates mit einer schematischen Ansicht

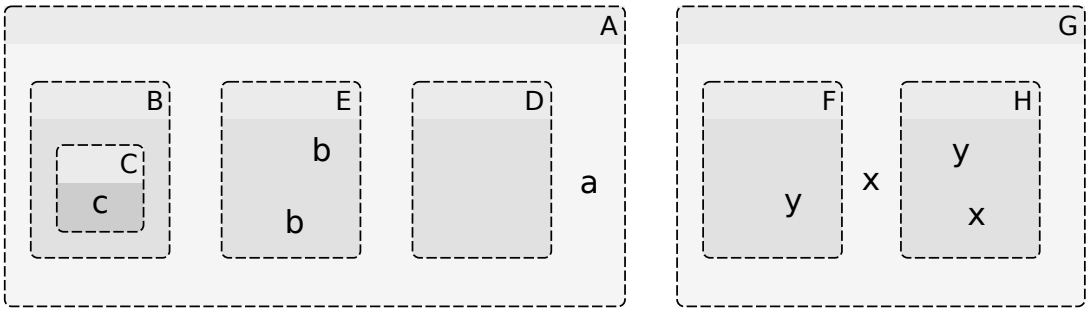


Abbildung 5.4: Schachtelung von Wiederholungen: Einige denkbare Anordnung von Meta-Variablen und Wiederholungen

Scope jedes Nichtterminal angegeben werden. Im Template können außer den in der Sprache vorhandenen Syntaxkonstrukten auch Meta-Variablen und Wiederholungen verwendet werden. Die Meta-Variablen können dabei in den Templates nur an Positionen verwendet werden, an denen auch das syntaktische Konstrukt möglich wäre, welches dem Typ, das heißt dem Nicht-Terminal, der Meta-Variable entspricht. Jedem Subtemplate wird ein Index als Name (hier: `sub1`) vergeben, damit später die Information zur Verfügung steht, welches Subtemplate aus dem BeforeTemplate durch welches Subtemplate aus dem After-Template ersetzt werden soll.

Wiederholungen In manchen Fällen reicht die Mächtigkeit der einfachen Meta-Variablen nicht aus. Dies ist beispielsweise der Fall, wenn ein mehrfaches Auftreten eines Konstrukts gematcht werden soll, wie eine Parameterliste. Ziel ist es, auszudrücken dass ein Teil eines Templates wiederholt auftreten kann. In Re$\mathcal{L}$ wird dies durch *Wiederholungen* zusammen mit einer Erweiterung der Meta-Variablen umgesetzt. Wiederholungen können wie in regulären Ausdrücken gekennzeichnet werden. Die Wiederholungen werden zur Referenzierung zusätzlich mit einem Namen versehen (vgl. Abbildung 5.3 links). Allgemein werden sie durch eine Klammerung, die dem Index vorangestellt ist, und dem Wiederholungskennzeichen beschrieben:

```
"#" Indexname "(" Nichtterminal ")" ("*" | "+" | "?")
```

$*$	beliebig oft	$0 \leq x \leq \infty$	(ZeroOrMore)
$+$	mindestens einmal	$1 \leq x \leq \infty$	(OneOrMore)
?	maximal einmal	$0 \leq x \leq 1$	(ZeroOrOne)

Der Indexname kann ein beliebiger noch nicht verwendeter Name sein. Ein Wiederholungszeichen kann nur an solchen Stellen angegeben werden, wo in der BNF der Zielsprache Wiederholungen möglich sind. Wenn an anderen Stellen Wiederholungen erlaubt wären, könnten diese nur in einer Weise matchen. Das würde den Code nur unübersichtlich machen. Da in der BNF verschiedene Arten von Wiederholungen existieren, werden Nebenbedingungen für die Template-Wiederholungen gemacht: Die Wiederholungen im Template sollten nicht durch die Wiederholungen in der BNF der Zielsprache unmöglich oder unsinnig werden. Wenn die BNF eine *ZeroOrOne*-Wiederholung enthält, ergibt eine *OneOrMore*-Wiederholung im Template keinen Sinn: Die Zusammensetzung der Bedingungen zwingt die Wiederholung *genau* einmal zu matchen, welches durch die Nutzung einer einfachen Meta-Variablen ausgedrückt werden kann.

Es bleibt die Frage offen, welche Werte Meta-Variablen innerhalb einer Wiederholung annehmen können. Häufig sollen die Meta-Variablen beim Ausrollen der Wiederholungen einen anderen Wert beinhalten. Beispielsweise lassen sich auf diese Weise Parameterlisten einfach in einem Template ausdrücken. Dazu werden die Meta-Variablen erweitert, so dass sie Listen von Werten enthalten können. Diese Listen, die eine Sequenz darstellen, können geschachtelt auftreten. Mit jeder Schachtelung einer Wiederholung wird eine neue geschachtelte Sequenz eingeführt. Jede Sequenz darf dabei nur Objekte der gleichen Art beinhalten. Entweder in der Sequenz sind alle Elemente wieder Sequenzen oder alle sind Codestücke.

Die Sequenzen in den Meta-Variablen führen zu einer weiteren Schwierigkeit. In einigen Fällen ist nicht klar, ob sich Sequenzen oder Werte in der Meta-Variablen befinden sollen. In Abbildung 5.3 ist eine Meta-Variable `x` deklariert, welche innerhalb einer Wiederholung mit Index `I2` genutzt wird. Diese Meta-Variable wird auch außerhalb dieser Wiederholung genutzt. Die Frage ist, soll die Meta-Variable `x` einen einzelnen Wert oder eine Sequenz von Werten enthalten? In diesem Fall würde nur der einzelne Wert brauchbar sein, weil die Verwendung eines Arrays außerhalb einer Wiederholung nicht definiert ist.

Dies führt zur Deklaration von Meta-Variablen als *wiederholungsglobal*. Wenn eine Variable *wiederholungsglobal* ist, dann ist der Wert der Variablen innerhalb der angegebenen Wiederholung immer gleich. Wenn sie nicht wiederholungsglobal ist, ist der Wert der Variablen eine Sequenz, die durch die Wiederholungen durchlaufen wird. Da Wiederholungen innerhalb anderer Wiederholungen verwendet werden können, ist die Unterscheidung in „wiederholungsglobal“ und „nicht wiederholungsglobal“ nicht ausreichend. Ein Schachtelung von Wiederholungen ist eine Baumstruktur. Es ist sinnvoll eine Variable für ganze Teilbäume von Wiederholungen als *wiederholungsglobal zu einem Wiederholungsbaum* zu kennzeichnen und gleichzeitig eine nicht kompatible Nutzung zu verbieten.

Abb. 5.4 versinnbildlicht mittels der Kästen verschiedene Wiederholungen (A bis G), die teilweise geschachtelt sind. Eine Wiederholung darf nicht die Kante einer anderen Wiederholung schneiden. In den Wiederholungen werden verschiedene Variablen benutzt. Die klein geschriebenen Buchstaben (a-c,x,y) bezeichnen Stellen, wo die Meta-Variablen genutzt werden. So wird die Variable b mehrmals innerhalb der Wiederholung mit Index E verwendet. Die Wiederholung E befindet sich ihrerseits in der Wiederholung A. Die Multiplizitäten der Wiederholung sind für die folgenden Betrachtungen unerheblich und werden deshalb ausgelassen.

Bei der Variable y fällt auf, dass sie in verschiedenen Wiederholungen genutzt wird. Nun ist nicht klar, was das bedeuten soll, da die Wiederholungen F und H unterschiedlich oft auftreten können. Ähnlich ist es mit der Variable x: Wird sie mit den Werten aus der Wiederholung G belegt oder mit den Werten aus der Wiederholung H? Die hier auftretenden Fragen können bei einer globalen Definition von x und y einfach beantwortet werden: Da jedes Auftreten unabhängig von der Wiederholung den gleichen Wert aufweisen muss, sind beide Variablen eindeutig belegt. Wenn x und y nicht global sind, kann für diese keine sinnvolle Belegung gefunden werden. Teilweise werden Variablen benötigt, die sich bezüglich einiger Wiederholungen als nicht global, aber gegenüber anderen Wiederholungen als global verhalten. Wenn eine Meta-Variable bzgl. einer Wiederholung global ist, muss sie auch für alle eingebetteten Wiederholungen global sein. Dies wird wie oben eingeführt *wiederholungsglobal* genannt.

Da Wiederholungen sich nicht überschneiden können und verschachtelt sind, ist es zweckmäßig für jede Variable anzugeben, ab welcher Wiederholung sie als wiederholungsglobal angesehen werden soll. Innerhalb der Wiederholung und deren eingebetteten Wiederholungen kann die Meta-Variable immer nur einen Wert annehmen. In den darüber gelegenen Wiederholungen wird der Inhalt der Meta-Variablen an die Wiederholungsstruktur gebunden. Diese Angabe wird bei der Deklaration der Meta-Variablen gemacht. Es wird vor die Meta-Variable geschrieben, ab welchem Index sie wiederholungsglobal ist. Wenn sie global zum ganzen Template ist, wird das Schlüsselwort *global* verwendet. Wenn keine Angabe gemacht wird, ist sie *lokal*, dass heißt zu keiner Wiederholung global. Somit sieht die angepasste Meta-Variablendeklaration wie folgt aus:

```
(Indexname | "global")? name ("," name)* ":" type ("{" invariant "}")?
```

Wie das interne Mappen der Meta-Variablen innerhalb von den Wiederholungen funktioniert, wird im Kapitel 5.9 über die Semantik behandelt.

Mehrfach matchende Subtemplates Innerhalb eines Template sind die Wiederholungen ein wichtiges Mittel, um flexibel ein Refactoring zu definieren. Oft ist es auch notwendig Subtemplates mehrfach zu matchen. Von der Idee her, funktioniert dies genauso wie bei den Wiederholungen in den Templates.

Eine Mehrfachanwendung eines Subtemplates wird im Header des Subtemplates angegeben. Dazu dient das Schlüsselwort *Rep:*, hinter welchem eine Art von Repetition angegeben werden kann. Dabei werden die gleichen Symbole zur Kennzeichnung genutzt wie bei den Wiederholungen. Da bei diesen Mehrfachanwendungen das selbe Problem mit den Variablen auftreten kann, werden auch die Indizes für die Subtemplates bei der Deklaration von wiederholungsglobalen Meta-Variablen erlaubt. Das Verhalten der Meta-Variablen ist somit analog.

Subtemplates, die nicht ersetzt werden Die bisher beschriebenen Subtemplates werden benötigt, um Code-Bereiche zu identifizieren, die durch das Refactoring geändert werden. Wer einige Refactorings modelliert hat, stellt fest, dass manche der Subtemplates im AfterTemplate wiederholt werden, da nur einige Informationen über den Code gewonnen werden sollten. Dabei wird der Code später im AfterTemplate unverändert wieder zusammengesetzt. Um diese Nutzung eines Subtemplates zu vereinfachen, erlaubt Re$\mathcal{L}$ ein Schlüsselwort `noReplace` zu verwenden. In diesem Fall wird das Template zwar gematcht

und die Variable belegt, aber es wird bei der Anwendung des AfterTemplate nicht durch neuen Code ersetzt.

Die `noReplace`-Subtemplates werden zusammen mit einer Erweiterung auch für die Einschränkung der Teilbäume des (E)AST benötigt. Wenn ein Methoden-Rumpf in einer Klasse gemachted werden soll, ist es einfach, die Klasse mit einem `noReplace`-Subtemplates zu bestimmen und dann in dem ergebenden Unterbaum die Methode zu finden. Dies ist nicht mit Templates möglich, die ersetzt werden, da der Inhalt des äußeren Templates in einer Weise verändert werden könnte, die die Anwendung des inneren Templates nicht erlaubt. Wenn in einer solchen Konstellation die äußeren Templates nur matchen, wie es beim `noReplace`-Subtemplate der Fall ist, kann im Innern ohne Schwierigkeiten Code ersetzt werden. Dies wird in Re$\mathcal{L}$ mit dem Schlüsselwort `In` gekennzeichnet. Im Header eines Subtemplates kann angegeben werden, dass dieses Subtemplate nur in einem bestimmten Bereich matchen kann. Da das `noReplace`-Subtemplate nicht verändert werden darf, kann hier ein Meta-Variablenname angegeben werden, der durch ein `noReplace`-Subtemplate definiert wird. Dieser Meta-Variablenname darf in keinem anderen Template verwendet werden.

Unterschiede zwischen BeforeTemplate und AfterTemplate Nachdem die Struktur eines Templates anhand des BeforeTemplate erläutert wurde, sollen in diesem Abschnitt die Unterschiede zwischen den Templates behandelt werden. Die meisten Unterschiede resultieren aus den unterschiedlichen Zwecken innerhalb einer Refactoringbeschreibung.

BeforeTemplate Das BeforeTemplate beschreibt den Code vor dem Refactoring. Es sind dabei alle Konstrukte für Templates möglich, die bisher beschrieben wurden. Im BeforeTemplate werden alle für das Refactoring wichtigen Bestandteile beschrieben und alle anderen durch Meta-Variablen verborgen. In der Durchführung des Refactorings, wird eine Meta-Variablen-Belegung gesucht, so dass das ausgefüllte Template (die Meta-Variablen wurden durch ihre Inhalte ersetzt) dem des Codes entspricht. Damit der Code nicht von der Wurzel des AST an beschrieben werden muss, wird mit dem Scope angegeben, auf welcher syntaktischen Ebene das BeforeTemplate aufsetzt. Beim Matchen wird also nicht nur eine Meta-Variablen-Belegung gesucht, sondern auch ein Ort, an den das Template passt. Dieser wird im Folgenden *Location* genannt, weil es sich um den Ort handelt, an dem ein Teil des Refactorings durchgeführt wird.

AfterTemplate Zur Beschreibung des Codes nach dem Refactoring wird das AfterTemplate (Listing 5.1 Zeilen 27–38) genutzt. Daher werden im AfterTemplate nicht alle möglichen Konstrukte eines Templates benötigt. Die Neudefinition von Variablen ist beispielsweise nicht nötig. Für jedes Subtemplate des BeforeTemplate muss es ein Subtemplate des AfterTemplate geben. Da auch der Scope des AfterTemplate mit dem Scope des BeforeTemplate übereinstimmen muss, wird diese Angabe beim AfterTemplate ausgelassen. Das AfterTemplate ersetzt, nachdem es mit den Meta-Variablen gefüllt wurde und die Wiederholungen ausgerollt sind, die entsprechenden Teile des Codes, welche durch das BeforeTemplate beschrieben wurden. Nicht syntaktische Unterschiede zwischen After- und BeforeTemplate finden sich in deren Semantik (vgl. Kapitel 5.9.1).

5.5.3 Precondition

Mit dem BeforeTemplate lassen sich viele strukturelle Eigenschaften ausdrücken. Dies reicht nicht aus um die Vorbedingung eines Refactorings zu beschreiben. Die Precondition (Listing 5.1 Zeilen 17–19) gibt hier die Möglichkeit, Bedingungen für das Refactoring zu definieren. Nachdem mit Hilfe des BeforeTemplates die Meta-Variablen mit Werten belegt wurden, stehen der Precondition die Meta-Variablen aus den Parametern und die im BeforeTemplate definierten Variablen zur Verfügung. Mit Hilfe dieser Meta-Variablen werden die Bedingungen ausgedrückt. Wenn alle Bedingungen erfüllt werden, kann das Refactoring ausgeführt werden, sonst nicht. In der Precondition werden keine zusätzlichen Hilfswerte für das Refactoring berechnet, dies ist der Calculation vorbehalten. Für die Beschreibung der Bedingungen werden häufig *Analysefunktionen* (vgl. Kapitel6.3), wie sie auch in [Rob99] eingeführt werden, verwendet. Diese Funktionen dürfen keine Seiteneffekte haben und liefern Aussagen über den Code. Sie ähneln auch den semantischen Hilfsfunktionen in Object-Z wie sie von Fischer definiert werden [Fis00].

In den Bedingungen können beliebige Z-Prädikate verwendet werden. Wenn mehrere Prädikate in Zeilen untereinander stehen, gelten diese als konjugiert. In den Z-Prädikaten können auch die semantischen Funktionen aus [Smi00] verwendet werden.

5.5.4 Calculation

Die Calculation (Listing 5.1 Zeilen 10-26) wird zusammen mit dem AfterTemplate genutzt, um die neue Spezifikation zu erstellen. Es gibt dabei zwei Arten, wie die neue Spezifikation erstellt werden kann. Entweder werden andere Refactorings aufgerufen (zusammengesetztes Refactoring) oder es wird mit Hilfe des AfterTemplate direkt eine neue Spezifikation erzeugt (einfaches Refactoring). Diese Unterscheidung wird besonders hier deutlich, da die Calculation in zwei verschiedenen Modi benutzt werden kann. Bei einem einfachen Refactoring werden durch Z-Ausdrücke, wie in Object-Z, Werte definiert, welche im AfterTemplate genutzt werden. Hier wird die Nachbedingung angegeben, weil diese eine gute Vorstellung gibt, was berechnet werden soll, ohne sich in algorithmischen Details zu verlieren.

Die zweite Möglichkeit besteht in einem durch Kontrollstrukturen gesteuertem Aufruf von anderen Refactorings. Dadurch wird es möglich, Refactorings aus anderen Refactorings zusammenzusetzen.

Berechnung von Werten Eine Aufgabe der Calculation ist es, neue Meta-Variablen zu berechnen, die nicht aus dem BeforeTemplate abgeleitet werden können. Wenn, wie in Listing 5.1, für die Parameterliste der neuen Funktion die Variablen eines Blockes benötigt werden, kann die Calculation genutzt werden um diese zu berechnen. Die hier verwendete Beschreibung gibt nur die Nachbedingung für solch eine Berechnung an. Somit können die Werteberechnungen ähnlich der Precondition angegeben werden. Der Hauptunterschied besteht darin, dass ein Deklarationsteil angegeben wird.

Kombination von Refactorings Zusammengesetzte Refactorings werden durch die Calculation realisiert. Dabei wird eine kompakte Kontrollsprache genutzt, um die Aufrufe der Subrefactorings zu steuern. Die Kontrollsprache enthält ein Schleifenkonstrukt (`for each`) und eine Verzweigung (`if then else`). Der Aufruf eines anderen Refactorings funktioniert ähnlich wie Funktionsaufrufe in anderen Sprachen. Eine Besonderheit ist, dass

```
main
  result := 0;
  if value2 < 0 then
      value2 := 1;
  fi
  while value2 != 0 do
      result := result + value1;
        value2 := value2 - 1;
    od
  result := result - 3;
endmain
```

```
main
  result := 0;
  if(value2 < 0)
      value2 := 1;
  fi
  result := mult(value2,value1,result);
  result := result - 3;
endmain
func mult(value2,value1,result)
    while value2 != 0 do
      result := result + value1;
      value2 := value2 - 1;
    od
    return result;
endfunc
```

Abbildung 5.5: Beispiel Refactoring: Programm vor und nach dem Refactoring aus Listing 5.1

alle benötigten Werte, wie z.B. die Mengen, die in der For-Each-Schleife verwendet werden, im Vorhinein durch die Calculation berechnet sein müssen.

```
CONTROL        = CONTROL; CONTROL
               | "IF" PREDICATE "THEN" CONTROL "ELSE" CONTROL
               | CALL_REFACTORING
               | "FOR EACH" MVAR "IN" EXPRESSION "DO" CONTROL "ENDFOREACH"
```

5.6 Das Refactoring-Repository

Im letzten Abschnitt wurde dargestellt, wie die einzelnen Refactorings beschrieben werden. Hier soll das Refactoring-Repository eingeführt werden, welches zum einen Refactorings sammelt und darüber hinaus auch Informationen zur Verhaltenserhaltung gibt. Der dritte Zweck des Repositories ist, polymorphe Refactorings bereitzustellen, so dass spezielle Refactorings für verschiedene Fälle gesammelt werden können und dann das passende Refactoring durchgeführt wird. Das vereinfacht sowohl die Erstellung, den Beweis, als auch das Verständnis der Durchführung eines Refactorings.

5.6.1 Verhaltenserhaltungs-Typen

Verschiedene Anwendungsbereiche fordern verschiedene Arten der Verhaltenserhaltung, die zwangsläufig zu einer unterschiedlichen Menge von gültigen Refactorings führen. So gibt es bei CSP-OZ mindestens drei Möglichkeiten die Verhaltenserhaltung zu definieren (siehe auch Kapitel 7.2). Dabei sind manche Refactorings für mehrere Erhaltungsbegriffe gültige Refactorings, andere nur für einen Begriff. Da bei unterschiedlichen Einsatzzwecken unterschiedliche Begriffe genutzt werden, wird bei einem Refactoring im Header optional `preserves` hinzugefügt, welchem eine Liste von Verhaltenserhaltungen folgt. Die Definition der Verhaltenserhaltungsbegriffe wird im Repository vorgenommen. Dazu wird nach dem Schlüsselwort `BEHAVIOUR` eine Menge von Verhaltenserhaltungsbegriffen angegeben. Im Listing 5.2 sind dies Trace, Failure, FailureDivergence.

```
ReL Repository
Language: CSP-OZ
BEHAVIOUR: { Trace, Failure, FailureDivergence }
OPTIONS {
Option1 = "Value";                                                    5
.....
}
...

Refactoring: splitClass                                               10
Preserves: Trace, Failure
Parameter: ...
```

Listing 5.2: Beispiel einer Repository-Struktur

5.6.2 Polymorphe Refactorings

Mit Re$\mathcal{L}$ beschriebene Refactorings haben durch den Template-Ansatz den Vorteil, gut verständlich zu sein. Durch das Konzept der Meta-Variablen und der Wiederholungen können viele Fälle in einer einzelnen Beschreibung zusammengefasst werden. Von Zeit zu Zeit wird dies unhandlich, weil die zu konstruierenden Refactorings kompliziert werden. In diesem Fall wird das Refactoring fast nur in der Calculation durchgeführt, was der Verständlichkeit zuwider läuft. Ein weiterer Vorteil ist es, Sonderfälle, für die es spezielle, anders aufgebaute Lösungen gibt, separat zu behandeln. In Re$\mathcal{L}$ ist es daher möglich Refactorings mit gleichen Namen und gleichen Parametern zu definieren. Wenn mehrere Refactorings in Namen und Parametern übereinstimmen, wird das erste Refactoring im Repository ausgeführt, welches ausführbar ist (BeforeTemplate matcht und die Precondition wird erfüllt).

5.6.3 Erweiterung von Re$\mathcal{L}$ um die Repository-Struktur

Die Erweiterung erfolgt in zwei Schritten. Als Erstes wird ein Repository-Header definiert, in dem die Sprache für die Refactorings und die Verhaltenserhaltungsbegriffe angegeben werden. In dem Header ist auch ein Bereich für Optionen definiert, der für Versionskennzeichnungen und Erweiterungen genutzt werden kann.

Als Zweites wird Re$\mathcal{L}$ um das Schlüsselwort `preserves` erweitert, welches ähnlich wie `extends` in Java, der Signatur des Refactorings angehängt wird.

5.7 Ableitung von Re$\mathcal{L}$ für eine BNF-Sprache

Im vorherigen Unterkapitel wurde Re$\mathcal{L}$ anhand der Instantiierung für FWHILE erklärt. Nur Teile von Re$\mathcal{L}$ sind von der Zielsprache abhängig. So sind die Typen der Meta-Variablen und die Inhalte der Templates sprachspezifisch, während der Aufbau der Precondition und der Calculation unabhängig ist (vgl. Listing 5.1). Wenn die sprachspezifischen Teile von Re$\mathcal{L}$ ausgetauscht werden, kann eine Refactoringbeschreibungssprache für andere Zielsprachen entwickelt werden. Bei der Definition von Re$\mathcal{L}$ wurden Techniken aus dem Bereich der *domänenspezifischen Sprachen* (DSL) [RGTL09, MHS05] genutzt, um eine universelle Sprache zu schaffen, die sich für unterschiedliche Zielsprachen instantiieren lässt (vgl. Abschnitt 5.11.3). Nachdem Re$\mathcal{L}_{FWHILE}$ erklärt wurde, soll eine Ableitung einer Instanz der Sprache gezeigt werden. Dabei wird gezeigt, wie für eine beliebige BNF-basierte Sprache X die Instantiierung Re$\mathcal{L}_X$ erstellt wird. Re$\mathcal{L}_X$ wird als BNF beschrieben. Dabei gibt es Teile der Sprache, die immer gleich sind (z.B. der Rahmen mit BeforeTemplate, Precondition,

Calculation und AfterTemplate) und Teile, die sprachabhängig sind. Als Beispiel wird die schon genutzte Sprachinstanz Re$\mathcal{L}_{FWHILE}$ entwickelt. Die Sprache, in der die Refactorings ausgeführt werden sollen, wird im folgenden *Zielsprache* genannt.
Die Erstellung gliedert sich in vier Schritte:

1. Einführung von Meta-Variablen
2. Einführung von Wiederholungen
3. Zusammensetzen mit der Core-BNF von Re$\mathcal{L}$
4. Berechnung der NonTerminal Namen (`NTTYPE`) und der Scope-Namen in Kombination mit den entsprechenden Produktionen (`NTSCOPETYPE`)

Die ersten beiden Schritte erstellen aus der BNF der Zielsprache einen Teil von Re$\mathcal{L}$, der für die Inhalte der Templates benutzt wird. Die Templates benutzen die Elemente der Ausgangssprache, die um die Verwendung von Meta-Variablen und Wiederholungen erweitert wurden. Dies spiegelt sich auch in der Ableitung wieder. Anschließend wird das Ergebnis mit einer BNF für die unveränderlichen Produktionen von Re$\mathcal{L}$ verbunden. Die festen Teile beinhalten die Grundstruktur der Refactorings, die Meta-Variablen und so weiter. Im letzten Schritt werden die Nicht-Terminale berechnet, deren Definition in der BNF noch fehlen. Diese sind im Wesentlichen die Benennungen von Typen.

Voraussetzungen Damit die Transformation funktioniert, soll die Ausgangs-BNF der Sprache folgende Bedingung erfüllen:

- Aneinanderreihungen und Aufzählungen sollen nicht rekursiv definiert sein.

Diese Voraussetzung ist keine echte Einschränkung, da jede BNF in eine entsprechende BNF umgewandelt werden kann. Zum Beispiel werden Parameterlisten gerne rekursiv beschrieben:

```
Parameterlist ::= Parameter | Parameter "," Parameterlist
```

Dies lässt sich einfach äquivalent als

```
Parameterlist ::= Parameter ("," Parameter)*
```

schreiben, was die geforderte Voraussetzung erfüllt.

Diese Voraussetzung vereinfacht die Erstellung der Re$\mathcal{L}$-Instanz erheblich, da die Rekursionen nicht untersucht werden muss, um festzulegen, welche Rekursionen als Wiederholungen realisiert werden müssen und welche nicht.

Die Erstellung der Re$\mathcal{L}$-Instanz arbeitet auf Produktionen. Bei den Produktionen in einer BNF werden meist die Erstellung von komplexen Token und Produktionen, die die Sprache selber erzeugen, gemischt. Bei der Definition von BNF in Kapitel 2 wurde für komplexe Token eine Schreibweise mit spitzen Klammern eingeführt. Im Folgenden werden die Umwandlungen nicht auf Teilproduktionen der komplexen Token wie `<LETTER>` oder `<DIGIT>` angewandt.

Einführung der Meta-Variablen Im ersten Schritt werden die Meta-Variablen eingeführt. Dabei wird nur die Verwendung der Meta-Variablen in den Templates betrachtet. Die Deklaration und die Verwendung in der Calculation bzw. Vorbedingung sind in die ReL-Core-BNF eingebettet.

Zu Beginn werden alle Produktionen durch ein Anhängen von „A“ umbenannt und es wird für jeden alten Produktionsnamen eine Produktion der Form

```
Productionname := MVAR | Productionname_A
```

erstellt, ohne dabei die Namen auf der rechten Seite der Produktionen zu verändern. Für FWHILE werden aus

```
FWHILE  = MAIN (FUNC)*
FUNC    = "func" FNAME "(" PARAM ")" (STAT)* "endfunc"                    2
...
```

folgende Produktionen

```
FWHILE     = MVAR | FWHILE_A
FWHILE_A   = MAIN (FUNC)*                                                 2
FUNC       = MVAR | FUNCTION_A
FUNC       = "func" FNAME "(" PARAM ")" (STAT)* "endfunc"
...
```

Einführung von Wiederholungen Die Einführung der Wiederholungen ist etwas komplizierter. In einer BNF können drei verschiedene Arten von Wiederholungen genutzt werden: ZeroOrOne (?), ZeroOrMore (*) und OneOrMore (+). Die Umsetzung dieser Wiederholungen ist ähnlich. Der Hauptunterschied besteht in den möglichen Wiederholungsnutzungen in ReL. So ergibt z.B. bei einem OneOrMore in der BNF keine Wiederholung der Art ZeroOrOne Sinn, weil nur genau eine Wiederholung möglich wäre. Diese umständliche (und den Entwickler verwirrende) Schreibweise soll unterbunden werden. Eine andere Schwierigkeit ist, dass die Wiederholungen innerhalb einer Produktion genutzt werden und verschachtelt auftreten können.

Die Einführung der Wiederholungen wird in zwei Schritten durchgeführt. Als Erstes werden die Wiederholungen in separate Produktionen ausgelagert und anschließend in diesen Produktionen die Wiederholungen im Sinne von ReL eingearbeitet.

Die neuen Produktionen enthalten einfach die Wiederholung mit ihren Inneren. Wenn mehrere Produktionen verschachtelt sind, werden die nacheinander aufgelöst. Somit wird aus einer Produktion

```
P0 := P1 (P2 (P4)* P3)+
```

die Produktionsfolge

```
P0 := P1 P0_B1
P0_B1 := (P2 P0_B2 P3)+
P0_B2 := (P4)*
```

Zur Optimierung können identische Produktionen mit unterschiedlichen Namen zu einer zusammengefasst werden. Die Optimierung verkleinert die Anzahl der Produktionen, ist für die korrekte Durchführung aber nicht nötig.

Die Produktionen mit Wiederholungen haben jetzt alle die gleiche Form. Das äußere Element der Produktionen ist die Wiederholung. Die Produktionen können nach folgendem Schema ersetzt werden:

1. `Production1 := (Production2)*`
   → `Production1 := (Production2 | "#" REP_INDEX "(" Production2 ")" ("*"|"+"|"?") )*`

2. `Production1 := (Production2)+`
   → `Production1 := (Production2 | "#" REP_INDEX "(" Production2 ")" ("*"|"+"|"?") )+`

3. `Production1 := (Production2)?`
   → `Production1 := (Production2)?  | "#" REP_INDEX "(" Production2 ")" "?"`

Die Regeln unterscheiden sich nur im Detail. Ihnen ist gemein, dass sie die Wiederholung um das syntaktische Wiederholungskonstrukt aus ReL als Alternative erweitern. Die ReL-Wiederholung wird durch ein #, gefolgt von dem Bezeichner der Wiederholung eingeleitet. In Klammern kann dann die alte Produktion verwendet werden. Am Ende folgt die Angabe der Multiplizität. Die Wiederholungen in ReL können gemischt mit den normalen Produktionen mehrfach hintereinander auftreten. Dies wird genutzt um beispielsweise eine Aufteilung in mehrere Meta-Variablen zu ermöglichen.

Auf den ersten Blick scheint es nicht richtig zu sein, dass in der Regel 2 alle drei Arten von Wiederholungen gebraucht werden. Der Grund ist, dass eine explizite Angabe einer Meta-Variablen gefolgt von einer ReL-Wiederholung des gleichen Typs modelliert werden kann. Dann stellt die explizite Angabe das mindestens einmalige Auftreten sicher. Unsinnige Fälle können nicht durch die Syntax abgefangen werden, das muss in der Wohlgeformtheit (vgl. Kapitel 5.8) sichergestellt werden. Die Regel 3 unterscheidet sich von den anderen, da eine explizite Angabe (auch die leere Angabe ist explizit) oder die ReL-Wiederholung auftreten kann, nicht beides.

Für das obige Beispiel ergibt sich folgende Produktionsfolge:

```
P0 := P1 P0_B1
P0_B1 := ( P2 P0_B2 P3  | "#" REP_INDEX "(" P2 P0_B2 P3  ")" ("*"|"+"|"?"))+
P0_B2 := (P4 | "#" REP_INDEX "(" P4 ")" ("*"|"+"|"?") )*
```

Bevor die weiteren Schritte durchgeführt werden, ist in Listing 5.3 die resultierende TEMPLATE BNF, nachdem alle Regeln angewandt wurden, abgebildet.

Zusammensetzen mit der Core-BNF von ReL In den bisherigen Schritten ist ausgehend von der Zielsprachen-BNF die Sprache für das Innere der Subtemplates gebildet worden. Ein großer Teil der Struktur von ReL ist bei jeder Instantiierung gleich. Dieser allgemeine Teil wird in der Core-BNF von ReL beschrieben. Das Zusammenfügen der Templatesprache und der Core-BNF (siehe Listing 5.4) gliedert sich in zwei Teile. Der erste Schritt ist das Hinzufügen der Core-BNF, das heißt deren Produktionen zu den bisher abgeleiteten Teilen. Der zweite Schritt ist die Erstellung von Produktionen, die die Verbindung zwischen den beiden Teilen herstellen.

Die Core-BNF beinhaltet das Repository und die Grobstruktur der Refactorings. Auch die Calculation und die Precondition sind vollständig in der Core-BNF beschrieben. Einzig die Definition der Typen und Scopes bzw. die Anbindung der Templates fehlen. Diese werden in den folgenden Schritten berechnet und vervollständigen die Erstellung der Grammatik.

Die Core-BNF besteht aus verschiedenen Teilen, welche verschiedene Bereiche von ReL modellieren. Der mengenmäßig größte Teil definiert die Prädikatenlogik erster Ordnung, wie sie für die Precondition und die Calculation benötigt werden. Die Prädikatenlogik ist an die der Definition der Prädikate aus Z angelehnt [Int02]. Der Unterschied zu Z besteht

```
FWHILE    =  MVAR | FWHILE_A
FWHILE_A  =  MAIN | FWHILE_B
FWHILE_B  =  (FUNC | "#" REP_INDEX "(" FUNC ")" ("*"|"+"|"?") )*
MAIN      =  MVAR | MAIN_A
MAIN_A    =  "main" MAIN_B "endmain"                                          5
MAIN_B    =  (MSTAT| "#" REP_INDEX "(" MSTAT ")" ("*"|"+"|"?") )*
FUNC      =  MVAR | FUNC_A
FUNC_A    =  "func" FNAME "(" PARAM ")" FUNC_B "endfunc"
FUNC_B    =  (STAT| "#" REP_INDEX "(" STAT ")" ("*"|"+"|"?") )*
MSTAT     =  MVAR | STAT_A                                                    10
MSTAT_A   =  ( "SKIP"
             | VAR ":=" ITERM ";"
             | "if" BTERM "then" MSTAT_B1 MSTAT_B2 "fi"
             | "while" BTERM "do" MSTAT_B1 "od"
             )                                                               15
MSTAT_B1  =  (STAT| "#" REP_INDEX "(" STAT ")" ("*"|"+"|"?") )*
MSTAT_B2  =  ("else" MSTAT_B1| "#" REP_INDEX "(" "else" MSTAT_B1 ")" "?" )?
STAT      =  MVAR |STAT_A
STAT_A    =  (MSTAT | "return" (ITERM | BTERM)";")
FNAME     =  MVAR | FNAME_A                                                   20
FNAME_A   =  IDENT
PARAM     =  MVAR | PARAM_A
PARAM_A   =  (VAR PARAM_B) | "#" REP_INDEX "(" VAR PARAM_B ")" "?" )?
PARAM_B   =  ("," VAR| "#" REP_INDEX "(" "," VAR ")" ("*"|"+"|"?") )*
BTERM     =  MVAR | BTERM_A                                                   25
BTERM_A   =  (ITERM RELSYM ITERM)| FUSE
ITERM     =  MVAR | ITERM_A
ITERM_A   =  (VAR | CONST| FUSE) ITERM_V
ITERM_B   =  (("+" | "-") (VAR | CONST| FUSE)
             | "#" REP_INDEX "(" ("+" | "-")(VAR | CONST| FUSE) ")" ("*"|"+"|"?"))*   30
FUSE      =  MVAR | FUSE_A
FUSE_A    =  FNAME "(" PARAM ")"
VAR       =  MVAR | VAR_A
VAR_A     =  IDENT
IDENT     =  MVAR | IDENT_A                                                   35
IDENT_A   =  <LETTER> (<LETTER> | <DIGIT>)*
CONST     =  MVAR | CONST
CONST_A   =  ("+" | "-")? <DIGIT> (<DIGIT>)*
<LETTER>  =  ["a"-"z","A"-"Z"]
<DIGIT>   =  ["0"-"9"]                                                        40
RELSYM    =  MVAR | RELSYM_A
RELSYM_A  =  "=" | "!=" | "<" | ">"
```

Listing 5.3: Ergebnis der Umwandlung in die Template BNF

```
/* Globale Definitionen von Rel */
    REFAC_DESC  :=  REL_HEADER PARAMETER BEFORETEMPLATE PRECONDITION CALCULATION AFTERTEMPLATE
//TEMPLATES                                                                                    3
    TEMPLATE    :=  "Index: " NAME "Scope: " NTSCOPETYPE
    TEMPLATE_REP    :=  "Index:" NAME ( "Repeat:" REPEAT )? "In:" NAME "Scope:" NTSCOPETYPE
    REPEAT  :=  ( "+" | "*" | "?" )
    BEFORETEMPLATE  :=  "BeforeTemplate:" TEMPLATE_DECL_PART ( TEMPLATE_REP )+
    AFTERTEMPLATE   :=  "AfterTemplate:" ( ( TEMPLATE )+ | "NONE" )                             8
    TEMPLATE_DECL_PART  :=  "Define:" MVAR_DECL_MOD
// Precondition
    PRECONDITION    :=  "Precondition:" ( ZPREDICATE | "NONE" )
//Parameter and meta variables
    MVAR    :=  <IDENTIFIER>                                                                    13
    PARAMETER   :=  "Parameter:" MVAR_DECL
    MVAR_DECL   :=  ( MVAR_DEC ";" )*
    MVAR_DECL_MOD   :=  ( MVAR_DEC_MOD ";" )*
    MVAR_DEC    :=  Identitier ( "," Identitier )* ":" NTTYPE ( "[" ZPREDICATE "]" )?
    MVAR_DEC_MOD    :=  FLATMOD Identitier ( "," Identitier )* ":" NTTYPE ( "[" ZPREDICATE "]" )?   18
    FLATMOD :=  ( FLATMOD_GLOBAL | FLATMOD_LOCAL | FLATMOD_LOCATION | FLATMOD_REPETITION )
    FLATMOD_GLOBAL  :=  "global"
    FLATMOD_LOCAL   :=  "local"
    FLATMOD_LOCATION    :=  ( ":" Identifier )
    FLATMOD_REPETITION  :=  ( "#" Identifier )                                                  23
    Identitier  :=  <IDENTIFIER>
// Calculation
    CALCULATION :=  "Calculation:" CALC_DECL_PART CALC_DECL_LOCAL CALC_MAIN
    CALC_DECL_PART  :=  ( "Define:" MVAR_DECL )?
    CALC_DECL_LOCAL :=  ( "Local: " MVAR_DECL )?                                                28
    CALC_MAIN   :=  ( "SCHEMA:" ZPREDICATE )? ( "DO" CONTROL )? "DONE"
    CONTROL :=  ( "IF" ZPREDICATE "THEN" CONTROL "ELSE" CONTROL "ENDIF" | CALL_REFACTORING | "FOR EACH"
         MVAR "IN" EXPRESSION "DO" CONTROL "ENDFOREACH" ) ( ";" CONTROL )?
    REL_HEADER  :=  "Refactoring:" NAME "Preserves:" "{" ( NAME ( "," NAME )* )? "}"
/* Repository */
    REL_REPOSITORY  :=  ( "ReL" "Repository" "Language:" NAME REL_REPO_BEHAVIOUR REL_REPO_OPTIONS (   33
         REFAC_DESC )* )
    REL_REPO_BEHAVIOUR  :=  "Behaviour:" "{" ( NAME ( "," NAME )* )? "}"
    REL_REPO_OPTIONS    :=  "OPTIONS" "{" ( REL_REPO_OPTION )* "}"
    REL_REPO_OPTION :=  <IDENTIFIER> "=" ( "true" | "false" | <INT> | <STRING_LITERAL> ) ";"
/* Imported from ISO-Z */
    ZPREDICATE  :=  Z_PredicateList                                                             38
```

Listing 5.4: Der Anfang der Core-BNF, in der die wichtigsten globalen ReL Konstrukte definiert werden. Die Definition der Prädikate ist weggelassen worden.

darin, das einige Schema-Konstrukte wie Schema-Erweiterung oder Selektion aus einem Schema in der Precondition und der Calculation nicht enthalten sind, da Meta-Variablen keinen Schema-Typ haben können.

Ein weiterer Teil der Core-BNF ist die Definition der Deklarationen in den verschieden Sektionen. In Listing 5.4 wird der entsprechende Teil der Core-BNF gezeigt. Die BNF beginnt mit der Definition einer Refactoringbeschreibung (`REFACDESC`). Der nächste Abschnitt beschreibt die globalen Elemente der Templates. Hier wird auch die noch zu definierende Produktion `NTSCOPETYPE` genutzt, welche die abgeleitete Sprache für die Templates einbindet. Nachdem alle Teile eines Refactorings definiert sind, schließt der Ausschnitt mit der Definition des Repositories.

Berechnung von `NTTYPE` und `NTSCOPETYPE` Um die Erstellung der ReL-Instanz abzuschließen, fehlen noch zwei Produktionen, welche die Core-BNF mit den anderen Produktionen verbinden. Die Produktion `NTTYPE` wird genutzt, um die Typen der Meta-Variablen

```
NTTYPE  =   "FWHILE"
          | "MAIN"                                                                    2
          | "FUNC"
          | "MSTAT"
          | "STAT"
          | "FNAME"
          | "PARAM"                                                                   7
          | "BTERM"
          | "ITERM"
          | "FUSE"
          | "VAR"
          | "IDENT"                                                                  12
          | "CONST"
          | "RELSYM"

NTSCOPETYPE =  "FWHILE" "\n" FWHILE
             | "MAIN"   "\n" MAIN                                                    17
             | "FUNC"   "\n" FUNC
             | "MSTAT"  "\n" MSTAT
             | "STAT"   "\n" STAT
             | "FNAME"  "\n" FNAME
             | "PARAM"  "\n" PARAM                                                   22
             | "BTERM"  "\n" BTERM
             | "ITERM"  "\n" ITERM
             | "FUSE"   "\n" FUSE
             | "VAR"    "\n" VAR
             | "IDENT"  "\n" IDENT                                                   27
             | "CONST"  "\n" CONST
             | "RELSYM" "\n" RELSYM
```

Listing 5.5: Die berechneten Produktionen `NTTYPE` und `NTSCOPETYPE`

anzugeben und die Produktion `NTSCOPETYPE` beinhaltet Tupel aus Scope und der passenden Produktion. Als Erstes wird ein Nicht-Terminal `NTTYPE` berechnet, welches alle Namen von Nicht-Terminalen der Ausgangs-BNF enthält. Dabei wird `NTTYPE` als Choice über alle Produktionsnamen gebildet. Das zweite Nicht-Terminal `NTSCOPETYPE` wird für die Scopedefinition benutzt. Es enthält daher den Namen der Nicht-Terminale und nach einem Zeilenwechsel die Nicht-Terminal-Produktion selbst. Das Ergebnis für FWHILE ist in Listing 5.5 gezeigt.

Damit ist die Erstellung der BNF für eine Re$\mathcal{L}$ Instanz abgeschlossen.

5.8 Wohlgeformtheit eines Re$\mathcal{L}$-Refactorings

Wie bei vielen Sprachen, ist es nicht möglich jeder syntaktischen Beschreibung aus Re$\mathcal{L}$ eine sinnvolle Semantik zu geben (vgl. Kapitel 2). So müssen in Re$\mathcal{L}$ für alle Subemplates, die im BeforeTemplate spezifiziert sind, ein entsprechendes Subtemplate im AfterTemplate existieren, sofern es sich nicht um ein `noReplace`-Subtemplate handelt. Ein Re$\mathcal{L}$-Code ist wohlgeformt, wenn folgendes gilt:

- Alle Meta-Variablen sind deklariert.
- Alle Meta-Variablen und ihre Nutzungen sind wohlgetypt.
- Wiederholungen und Sequenzen sind wohlgeformt.
- Die Templates sind wohlgetypt.

- Die Scopes und Multiplizität der Subtemplates sind kompatibel.
- Alle Z-Ausdrücke sind wohlgeformt bzw. wohlgetypt.
- Die Precondition stellt sicher, dass die Calculation berechenbar ist.

Im Folgenden sollen die einzelnen Bedingungen der Wohlgeformtheit (einschließlich der Wohlgetyptheit) genauer betrachtet werden. Dabei werden diese, soweit es sich um Standardvorgehensweisen handelt, informal erläutert.

Meta-Variablen Deklaration In Re$\mathcal{L}$ müssen alle Meta-Variablen vor ihrer Benutzung deklariert sein. Da die einzelnen Bestandteile eines Refactorings nacheinander abgearbeitet werden, ergeben sich die erlaubten Nutzungsorte für eine Meta-Variable:

Deklaration in	Nutzung in
Header (= Parameter)	BeforeTemplate, Precondition, Calculation und AfterTemplate
BeforeTemplate	BeforeTemplate, Precondition, Calculation und AfterTemplate
Calculation	Calculation und AfterTemplate

In anderen Teilen des Refactorings ist es nicht möglich neue Meta-Variablen zu deklarieren.

Wohlgetyptheit der Meta-Variablen Meta-Variablen in Re$\mathcal{L}$ sind über die Namen der Nichtterminale der Zielsprache getypt. Daher ist die Wohlgetyptheit der Meta-Variablen von der Zielsprache abhängig. Das Typsystem wird aus der BNF abgeleitet. Dazu müssen zwei Themenblöcke betrachtet werden. Erstens muss das Auftreten einer Meta-Variable wohlgetypt sein. Zweitens muss klar sein, wie sich die Typung über die Wiederholungskonstrukte und daraus abgeleiteten Sequenzen fortpflanzt.

Auf jeden Fall sollen an Stellen im Code, an der eine Produktion verwendet werden kann, eine Meta-Variable vom Typ der Produktion verwendet werden können. Häufig ist es sinnvoll auch andere kompatible Typen bzw. Wiederholungen von kompatible getypten Meta-Variablen zuzulassen. Ersteres erspart in den Templates für inkompatible Auftreten von Produktionen zusätzliche Meta-Variablen einzuführen. Die Wiederholungen sind von zentraler Bedeutung für die Mächtigkeit der Templates. Für beide Ansätze wird eine Typhierarchie benötigt, die aus der BNF der Zielsprache abgeleitet werden kann. Die Funktion **stype** berechnet für einen Produktionsnamen NT die Menge der hier erlaubten Typen. Eine *Leerableitung* ist in der Funktion ein Nichtterminal, welches zu einem leeren Text ableitbar ist. Die Funktion *unfold* ersetzt den Namen einer Produktion durch ihre Definition. Das T steht für ein Terminal und die Variablen a und b werden für BNF-Ausdrücke verwendet:

$$
\begin{aligned}
\mathbf{stype}(NT) &= \{NT\} \cup \mathbf{stype}(\mathit{unfold}(NT)) \\
\mathbf{stype}(T) &= \varnothing \\
\mathbf{stype}((a)) &= \mathbf{stype}(a) \\
\mathbf{stype}((a)*) &= \mathbf{stype}(a) \\
\mathbf{stype}((a)?) &= \mathbf{stype}(a) \\
\mathbf{stype}((a)+) &= \mathbf{stype}(a) \\
\mathbf{stype}(a \mid b) &= \mathbf{stype}(a) \cup \mathbf{stype}(b) \\
\mathbf{stype}(ab) &= \begin{cases}
\mathbf{stype}(a) & b \text{ ist leerableitbar;} \\
 & a \text{ ist nicht leerableitbar} \\
\mathbf{stype}(b) & a \text{ ist leerableitbar;} \\
 & b \text{ ist nicht leerableitbar} \\
\mathbf{stype}(a) \cup \mathbf{stype}(b) & a, b \text{ sind nicht leerableitbar} \\
\varnothing & \mathit{sonst}
\end{cases}
\end{aligned}
$$

Damit berechnen sich die erlaubten Typen der Produktion folgenden Beispiele für FWHILE:

$$\mathbf{stype}(\texttt{STAT}) = \{\texttt{STAT}, \texttt{MSTAT}\}$$
$$\mathbf{stype}(\texttt{ITERM}) = \{\texttt{ITERM}, \texttt{VAR}, \texttt{CONST}, \texttt{FUSE}, \texttt{INDENT}\}$$

Mithilfe der **stype** Funktion kann die Wohlgetyptheit der Meta-Variablen definiert werden werden.

Definition 30 (Wohlgetyptheit von Meta-Variablen) *Eine Meta-Variable ist wohlgetypt, wenn sie an einer Stelle genutzt wird, an der die Produktion Prod erwartet wird und der Typ Ty der Meta-Variable in der Menge der kompatiblen Typen der Produktion enthalten ist:*

$$\mathit{Ty} \in \boldsymbol{stype}(\mathit{Prod}).$$

In der Definition wird der Begriff der erwarteten Produktion verwendet. In den meisten Fällen ist dies die Produktion, die durch die Ableitung der Ausgangs-BNF an dieser Stelle erwartet wird. Dies deckt nicht den Fall ab, dass in der Produktion der Ausgangssprache eine Wiederholung vorliegt und diese im Template genutzt wird. Dies ist beispielsweise der Fall, wenn in einem FWHILE-Programm eine Folge von Statements zerlegt werden soll. Seien $st1$ und $st2$ Meta-Variablen vom Typ `MSTAT` eines FWHILE-Programms. Nun kann ein FWHILE-Hauptprogramm wie folgt mit einem Subtemplate zerlegt werden.

```
Index: I                                                    1
Scope: MAIN
main
  #i1(st1)*
  st2
endmain                                                     6
```

Dieses Subtemplate trennt mit Meta-Variablen `st2` die letzte Anweisung aus dem Hauptprogramm ab. Somit ist die Verwendung der Teilproduktion `(MSTAT)*` durch eine Sequenz von zwei Re$\mathcal{L}$-Ausdrücken ersetzt worden. Der Erste ist seinerseits eine Wiederholung, der Zweite die direkte Nutzung einer Meta-Variablen. In diesem Fall ist aus der Ausgangs-BNF nicht direkt ersichtlich, dass an diesen Stellen die Meta-Variablen vom Typ `MSTAT` erlaubt

sind. Dies gilt insbesondere für die zweite Meta-Variable, da durch den Ausdruck `#i1(st1)` die Wiederholung aus der BNF auf dem ersten Blick vollständig erfüllt scheint.

Für die Typung muss die Frage beantwortet werden, wie sich die Typung über die Wiederholungen und Sequenzen fortpflanzt. Die Syntax stellt im Vorhinein sicher, dass nur an durch die Ausgangs-BNF festgelegten Stellen eine Sequenz bzw. eine Wiederholung verwendet werden darf. Beide Konstrukte werden anstelle einer Produktion angewendet. In dem Beispiel sind in der Wiederholung und der Sequenz Meta-Variablen vom Typ `MSTAT`, weil im Innern der Wiederholung die durch die Sequenz und die Re$\mathcal{L}$-Wiederholung ersetzt wurde, die Produktion `MSTAT` genutzt wurde.

Allgemein werden in einer Sequenz oder einer Wiederholung die (Teil-)Produktionen genutzt, die anstelle der Sequenz oder Produktion erwartet würden.

Eine letzte Bedingung, die die Meta-Variablen erfüllen müssen, erwächst aus der Verwendung innerhalb von Wiederholungen. Jede Meta-Variable muss immer in der gleichen Einbettungstiefe innerhalb von Wiederholungen genutzt werden. Sonst kann der geordneten Wertemenge der Meta-Variablen keine Entsprechung in den Templates gegeben werden. Bei der Berechnung der Tiefe werden nur Wiederholungen betrachtet zu der die Meta-Variable nicht wiederholungsglobal ist.

Wohlgeformtheit der Wiederholungen und Sequenzen Die Wohlgeformtheit der Wiederholungen und Sequenzen beinhaltet einen Fall, der durch die Syntax nicht abgedeckt werden kann. Wenn in der Ausgangsproduktion eine Wiederholung vom Typ OneOrMore (+) auftritt, muss sichergestellt sein, dass im Template auch mindestens eine Wiederholung angegeben wird. Das heißt, dass in der verwendeten Sequenz mindestens eine Meta-Variable direkt eine Wiederholung vom Typ OneOrMore (+) genutzt wird.

Templates Die Relation aus BeforeTemplate und AfterTemplate beschreibt die Transformation. Dabei werden die in den Subtemplates beschriebenen Codestücke durch Code der Subtemplates aus dem BeforeTemplate ersetzt. Daher muss für alle Subtemplates des BeforeTemplate ein Gegenstück im AfterTemplate existieren. Eine Ausnahme ist, wenn es sich um eine Subtemplate handelt, welches durch `noReplace` gekennzeichnet ist. Die zueinander gehörigen Subtemplates müssen den gleichen Index haben. Des Weiteren müssen die Templates den entsprechenden Scope entsprechen, also ein Template für gültigen Code darstellen.

Z-Ausdrücke wohlgeformt und wohlgetypt An verschiedenen Stellen in Re$\mathcal{L}$ werden Z-Ausdrücke verwendet. Diese müssen nach den Richtlinien von Z wohlgeformt und wohlgetypt sein. Die genauen Regeln finden sich in [Int02] bzw. [Smi00].

Vorbedingung und BeforeTemplate bedingen Calculation Bei einem einfachen Refactoring muss aus der Vorbedingung folgen, dass die Calculation berechenbar ist, und zwar in einer Weise, dass das AfterTemplate gültig gefüllt werden kann. Meist bedeutet dies, dass die Meta-Variablen gültig und nicht-leer belegt werden können.

5.9 Semantik von ReℒL

Im vorigen Abschnitt ist die Sprache beschrieben worden, die hier für die Beschreibung des Refactorings genutzt wird. Als nächstes soll die Semantik von ReℒL behandelt werden. Die Semantik wird im Abschnitt 5.9.1 informell erläutert. Anschließend wird eine formale Semantik gegeben, die als semantische Domäne Object-Z nutzt.

5.9.1 Durchführung eines Refactorings mittels ReℒL

Bevor die Semantik im Detail behandelt wird, soll in diesem Abschnitt ein Verständnis geschaffen werden, wie ein Refactoring mit ReℒL durchgeführt wird. Dabei wird von technischen Details abstrahiert; diese werden in der formalen Semantik erklärt. Die Durchführung eines Refactorings ist in zwei Phasen aufgeteilt. Als Erstes wird geprüft, ob das Refactoring durchführbar ist. Diese Aufgabe wird vom BeforeTemplate und der Precondition übernommen. Der zweite Schritt ist die eigentliche Transformation des Codes. Dieser wird durch das BeforeTemplate, die Calculation und das AfterTemplate beschrieben. Das BeforeTemplate spielt eine besondere Rolle, da es in beiden Phasen genutzt wird. Dies wirkt sich auf den Ablauf eines Refactorings mit ReℒL aus: Die beiden Phasen werden zusammen durchgeführt, wobei im ersten Teilschritt (BeforeTemplate matchen und prüfen der Vorbedingung) ein Abbruch des Refactorings möglich ist. Dies ist für die Aufteilung der Phasen kein Problem, da in den ersten Schritten der Transformation nichts verändert wird: Die Veränderungen werden erst in der Calculation (Composite Refactoring) bzw. im AfterTemplate (einfaches Refactoring) durchgeführt. Insgesamt ergibt sich folgendes Vorgehen:

1. Matchen des BeforeTemplate (Belegen der deklarierten Variablen), sodass die Vorbedingung erfüllt wird
   Fehlschlag → Abbruch des Refactorings - Nicht anwendbar

2. Durchführen der Calculation

   - Einfaches Refactoring → Berechnen der Werte
   - Zusammengesetztes Refactoring → Durchführung der Sub-Refactorings

3. AfterTemplate füllen

   - Einfaches Refactoring → Ersetzen des Anziehungspunktes durch den neuen Code
   - Zusammengesetztes Refactoring → Da Template leer, keine Aktion

In dieser Übersicht sind die funktionalen Unterschiede eines einfachen und zusammengesetzten Refactorings zu erkennen. Ein einfaches Refactoring ist eine geschlossene Einheit, die selber den Code verändert. Ein zusammengesetztes Refactoring ist eine Art Programm, welches verschiedene andere Refactorings (einfach oder zusammengesetzt) aufruft.

Ein mit ReℒL beschriebenes Refactoring kann direkt auf eine Spezifikation oder ein Programm angewandt werden. Abbildung 5.5 zeigt links ein FWHILE-Programm, auf welches das Refactoring aus Abbildung 5.1 angewendet werden soll. Die Abbildung stellt das Matchen des BeforeTemplates dar. Links ist das Programm gegeben, in der Mitte das BeforeTemplate in dem die Parameter eingefügt wurden und rechts ist das resultierende

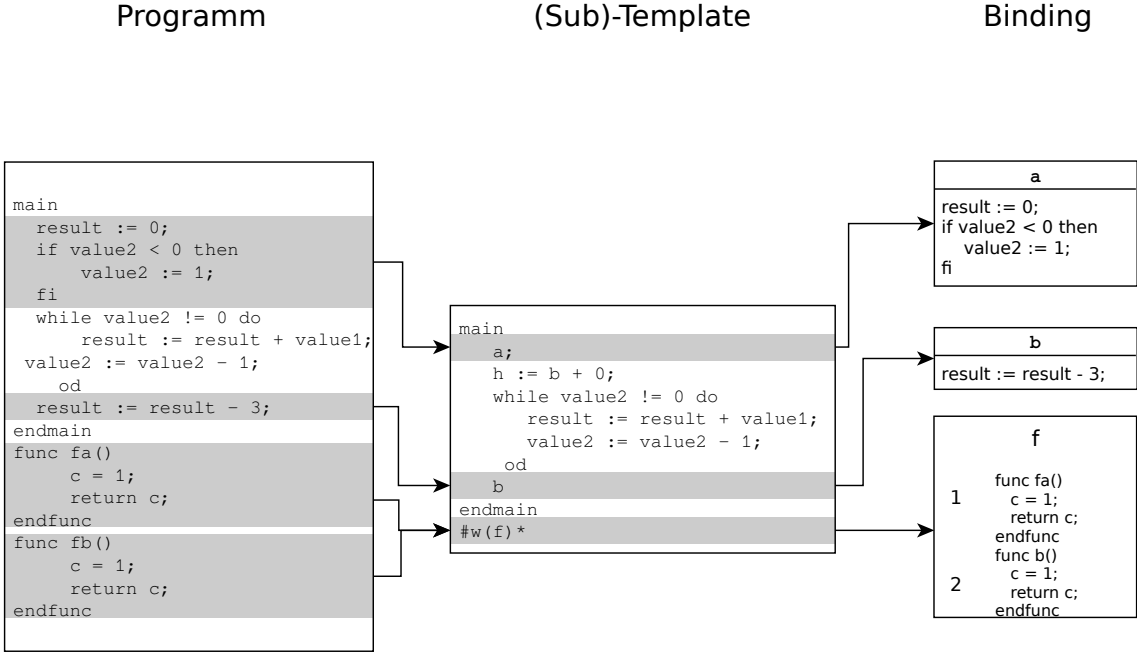


Abbildung 5.6: Matchen des BeforeTemplates anhand des Refactorings aus Abbildung 5.1

Binding der Meta-Variablen aus dem Template gegeben. Beim Matchen wird das BeforeTemplate zuerst mit den Werten der Parameter gefüllt. Das heißt, die Meta-Variablen sind durch den Inhalt der Parameter ersetzt worden. In der Abbildung ist dies bereits geschehen. Das resultierende BeforeTemplate (Mitte) wird mit dem Code (links) verglichen. Die übereinstimmenden Bereiche (weiß) werden für das Einpassen genutzt. Die grau unterlegten Bereiche werden anschließend in den Meta-Variablen abgelegt. Es ergibt sich ein Meta-Variablen-Binding, welches im linken Bereich dargestellt ist. Eine Besonderheit dieses Matchen ist, das eine Meta-Variable, die sich innerhalb einer Wiederholung befindet, zu der sie nicht global ist, auf eine Sequenz von Code-Bereichen abgebildet wird. In dem Beispiel ist dies bei den Funktionen `fa` und `fb` der Fall, die auf die Meta-Variable f gematcht wird, welche sich in der Wiederholung w befindet.

Nachdem das BeforeTemplate gematcht ist, wird die Vorbedingung geprüft. Die Vorbedingung in dem Beispiel ist erfüllt, was durch Vergleich der Belegungen aus Abbildung 5.6 mit der Vorbedingung aus Abbildung 5.1 bestätigt wird. Dabei ist in f die Sequenz der Funktionen abgelegt (vgl. Abbildung 5.10). Mit der Analysefunktion **names** (Siehe Anhang B.3), welche die Funktionsnamen bestimmt, es ergibt sich $\mathbf{names}(f) = \{\texttt{fa}, \texttt{fb}\}$. Die Funktionen writevars und readvars (vgl. Anhang B) bestimmen die Variablen, auf die schreibend bzw. lesend zugegriffen wird. Damit ergibt sich die Gültigkeit des ersten Teiles der Bedingung:

$$\forall\, a \in \mathbf{names}(f) \bullet a \neq \text{“}mult\text{“}$$
$$\mid \text{writevars}(operations) \cap \text{readvars}(b) \mid = 1$$

Die zweite Zeile stellt sicher, dass genau eine Variable existiert, welche sowohl im ausgelagerten Code geschrieben wird, als auch auf welche im späteren Code zugegriffen wird. Diese Bedingung wird benötigt, weil nur ein Wert als Rückgabe einer Funktion erlaubt ist.

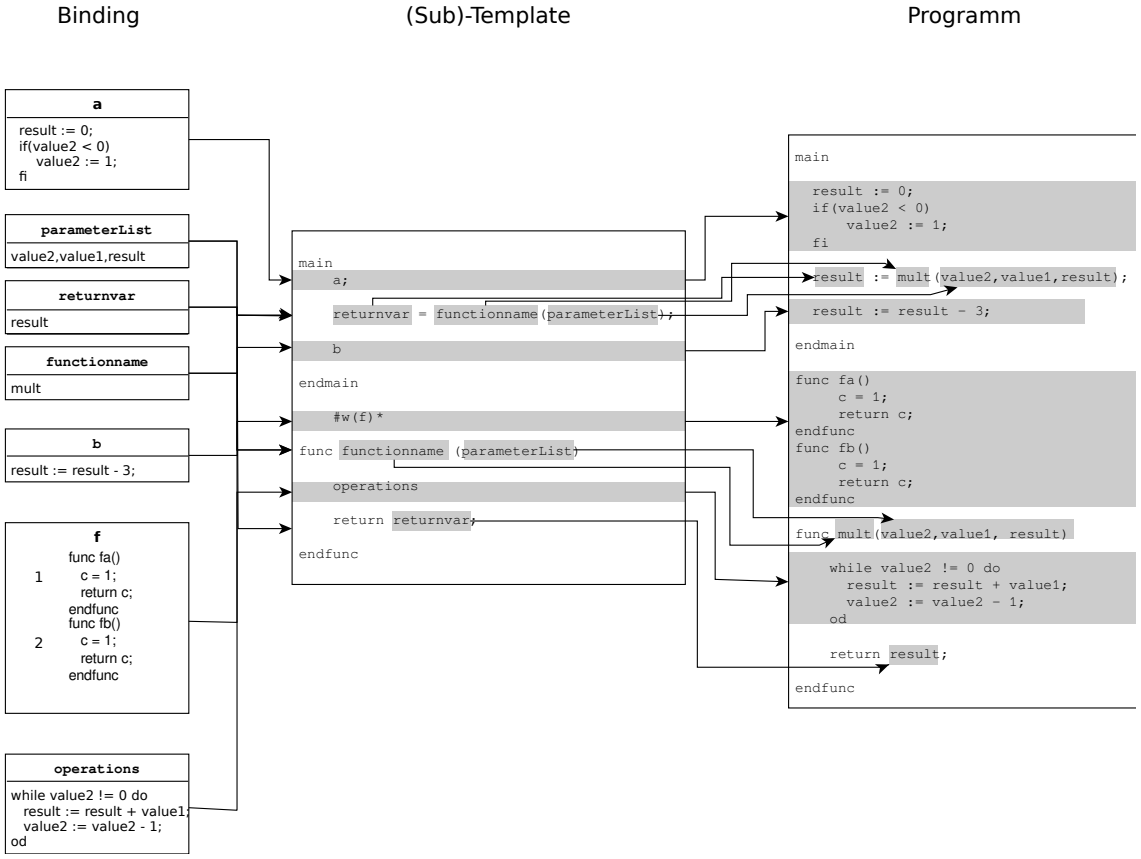


Abbildung 5.7: Erzeugen des Codes mittels des AfterTemplates (Mitte) und der Belegung der Meta-Variablen (Links) des Refactorings aus Abbildung 5.1

Wenn die Vorbedingung nicht erfüllt ist, muss ein anderes Matching gesucht werden, bis entweder ein Matching, welches die Vorbedingung erfüllt, gefunden wurde oder alle Matchings getestet wurden. Wenn kein Matching existiert, welches die Vorbedingung erfüllt, kann das Refactoring nicht ausgeführt werden. Wenn keine weitere Version im Repository vorhanden ist, schlägt das gesamte Refactoring fehl.

Als nächstes werden in der Calculation weitere Variablen berechnet. In der Beschreibung wird nur eine Nachbedingung angegeben. Die Belegung der neu deklarierten Variablen muss anschließend den Bedingungen genügen.

$$\texttt{parameterList} = \text{readvars}(\texttt{operations}) = \{\texttt{zaehler}, \texttt{value1}, \texttt{result}\}$$
$$\texttt{returnvar} \in \{\texttt{result}\} = \text{writevars}(\texttt{operations}) \cap \text{readvars}(\texttt{b})$$

Mit der resultierenden Variablenbelegung kann das refaktorisierte Programm zusammengesetzt werden. Im AfterTemplate werden alle Variablen durch ihre Belegung ersetzt, wobei die Wiederholungen ausgerollt werden. Das Ergebnis findet sich in Abbildung 5.7 auf der rechten Seite. Links steht die Belegung der Meta-Variablen nachdem die Calculation durchgeführt wurde. Die Werte der Meta-Variablen werden eingesetzt. Auf der rechten Seite ist der resultierende Code gegeben.

Abbildung 5.8: Grundaufbau eines Refactorings

5.9.2 Formale Semantik

Nachdem ein informelles Verständnis der Semantik der Refactoringsprache aufgebaut ist, wird in diesem Abschnitt die formale Semantik erläutert. Die semantische Domäne ist Object-Z, welches benutzt wird um die Transformation darzustellen. Im Grundaufbau (Abbildung 5.8), wird durch einen InputTransformer *InASTT* ein AST aus dem Code (unter Berücksichtigung der Parameter) erzeugt und dem eigentlichen Refactoring, welches durch die Operation *Refactoring* beschrieben wird, übergeben. Nach dem Refactoring wird durch ein OutputTransformer *OutASTT* wieder Code erzeugt.

Die semantische Domäne setzt sich aus mehreren Bestandteilen zusammen: Die Datenstruktur, die den Code, die Templates, den Inhalt von Parametern und den Meta-Variablen beschreibt und das eigentliche Refactoring, welches auf den vorgenannten Datenstrukturen arbeitet.

Die Datenstrukturen, die hier verwendet werden, sind ein *Abstract Syntax Tree* (AST) und eine Erweiterung des AST (EAST für Extended Abstract Syntax Tree). Der Code wird als AST dargestellt. In Parametern bzw. Meta-Variablen können Teilbäume eines AST verwendet werden. Der EAST erweitert einen AST um Klassen, die für die Templates verwendet werden. Diese Klassen sind MetaVariableNode (Meta-Variablen), RepetitionNode (Wiederholungen) und die Klassen, die die Orte, an denen die Wurzel eines Template matcht, beschreiben (LocationNode, NoReplaceLocationNode).

Die einzelnen Refactorings werden durch je eine Klasse für ein Refactoring dargestellt. Der Inhalt dieser Klassen wird durch eine Übertragung der Inhalte des Refactoring erzeugt. Für die Erstellung dieser Klassen gibt es zwei Vorgehensweisen. Eine erstellt aus einem einfachen Refactoring eine Klasse, die ein Refactoring direkt beschreibt. Ein zusammengesetztes Refactoring wird durch die andere Übertragungsvorschrift erzeugt. Der Hauptunterschied besteht darin, das diese Klasse auf andere Refactorings (einfache oder zusammengesetzte) aufbaut, die schon in eine Object-Z-Klasse überführt wurden. Der Rest der Semantikerklärung gliedert sich in die Beschreibung einiger Hilfsstrukturen, die anschließend zur Definition vom (E)AST verwendet werden. Das Kapitel wird durch die Beschreibung der Übertragung eines einfachen und eines zusammengesetzten Refactorings abgeschlossen.

Typdefinitionen Für die Definition des AST und des EAST werden einige Z-Type Deklarationen benötigt. Zum Einen sind es die freien Typen *Varname* und NT, welche durch die Namen der Meta-Variablen bzw. der Nicht-Terminale gegeben werden. Eine zweite Struktur wird benötigt um das Binding der Variablen zu beschreiben. Wegen der Struktur der Variablenbindings, die auch die Wiederholungen bzw. Mehrfachanwendungen von Subtemplates verwaltet, kann nicht die in Object-Z bestehenden Binding Strukturen (θ vgl. [Smi00]) verwendet werden. Hier wird dies durch die Bindingfunktion realisiert, welche mit

Hilfe der Value-Funktion definiert ist. Der Wert ist dabei als ein (Teil)-AST gegeben. Mit der generische Variable X wird gesteuert, ob es sich um einen AST oder EAST handelt. Da das Binding im Zusammenhang mit den Repetitions mehrere Werte enthalten kann, die der jeweiligen Repetition zugeordnet werden können sollen, wird dies durch eine gestufte Indizierung der Werte modelliert. Die Indexmenge ist immer ein Teilbereich der natürlichen Zahlen.

$$[Varname, NT, Invariant]$$
$$Binding[X] == Varname \pfun Value[X]$$

Die Werte (*Value*) berücksichtigen, dass durch verschiedene Schachtelungen von Repetition, verschiedene Anzahlen von Parametern benötigt werden. Innerhalb einer Funktion müssen die Parameterlängen gleich sein:

$$\begin{array}{|l} Value[X] : \mathbb{P}(\mathrm{seq}\,\mathbb{N} \pfun X) \\ \hline \forall f : Value[X] \exists i \in \mathbb{N} \forall m \in \mathrm{dom}(f) \bullet i = \#m \end{array}$$

Im Fall, dass die Sequenz leer ist, kann die Funktion direkt mit dem einzigen Wert identifiziert werden.

AST und EAST Der Abstract Syntax Tree (AST, vgl. Kapitel 2) ist in der Semantik die zentrale Datenstruktur. Sowohl der Eingabe- und Ausgabe-Code als auch interne Strukturen wie die Belegung der Meta-Variablen werden durch einen AST bzw. eine Erweiterung des AST beschrieben (EAST). Der AST wird wie üblich zur Beschreibung des Codes verwendet. Die Erweiterungen, die zum EAST führen, erlauben es einzelne Bestandteile eines Refactoring in einer den AST ähnelnden Struktur darzustellen. Dazu werden die einzelnen Subtemplates als eigene Bäume dargestellt. Der Wurzelknoten eines EAST für ein Subtemplate ist ein Nichtterminal (Klasse: *NonTerminal*). Als Blätter sind darin neben den Terminalen (Klasse: *Token*) auch Knoten für die Meta-Variablen möglich (Klasse: *MetaVariableNode*). Im Verlauf der Refactoringausführung, werden diese, für ein Refactoring festen, Bestandteile zusammen mit vom Code abhängigen weiteren Teil-EASTen zu der zentralen Datenstruktur des Refactorings zusammengebaut. Die einzelnen Teilbäume werden über verschiedene Zuordnungen miteinander verbunden. Ein Beispiel für eine solche Zuordnung ist das Binding der Meta-Variablen. Diese werden über eine Funktion vom Typ $Binding[X]$ beschrieben.

Da beide Strukturen einen Baum bilden, bietet sich zur Darstellung das Composite-Muster nach [GHJV85] an. In Abbildung 5.9 ist diese Struktur des EAST dargestellt. Die Klassen ASTElement, NonTerminal und Token bilden den AST, die Klassen MetaVariableNode, LocationNode, NoReplaceLocationNode und Repetition erweitern diesen zum EAST. NonTerminale entsprechen den Nicht-Terminalen der BNF der Ausgangssprache. Die Token sind feste Zeichenketten. Ein Nicht-Terminal enthält eine geordnete Liste von ASTElementen. Im erweiterten AST werden durch zusätzliche Knoten die einzelnen Konstrukte hinzugefügt, die für die Beschreibung bzw. Durchführung eines Refactorings benötigt werden. Ein MetaVariableNode ist das EAST-Äquivalent zu einem Auftreten einer Meta-Variable in einem der Templates. Hier werden nur die Informationen über das Auftreten einer Meta-Variable gespeichert, nicht die Belegung oder der Typ. Die Klasse Repetition definiert das Auftreten einer Wiederholung in den Templates. Eine Klasse, welche nicht gleich aus den Templates ersichtlich ist, ist der LocationNode. Er modelliert den

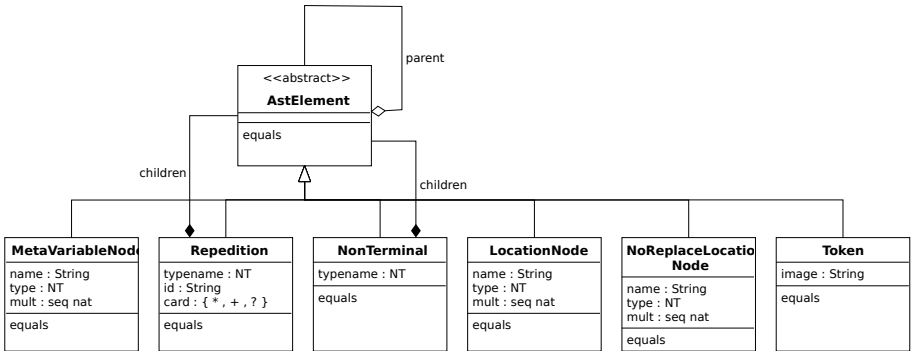


Abbildung 5.9: Übersicht über die Klassen des erweiterten AST

Ort, an dem ein Subtemplate den Code ersetzt. Oft wird ein Refactoring auf einen Teil der Spezifikation angewandt, z.B. nur auf eine Klasse, wobei die anderen Klassen unverändert bleiben. Der unveränderte Rest der Spezifikation stellt einen Rahmen dar, in dem der geänderte Code wieder eingefügt wird. Die Anknüpfungspunkte in diesem Rahmen (hier auch Frame genannt) sind die LocationNodes, welche die Orte der Refactoringanwendung kennzeichnen. Der NoReplaceLocationNode stellt die Location für ein `noReplace`-Subtemplate dar.

Die Idee ist, wie oben angedeutet, den Code der als AST gegeben ist, in die Strukturen von Re$\mathcal{L}$ zu zerteilen. Diese Strukturen sind die Subtemplates, die Inhalte der Meta-Variablen und der umgebende Code (Frame). Die neue Struktur ist ein EAST, der aus verbundenen Teilbäumen besteht. Die Verbindungen sind durch Bindings gegeben. Bevor das Zusammenspiel der Teilbäume beschrieben wird, sollen in den folgenden Unterabschnitten die einzelnen Klassen, die den AST bzw. EAST bilden, beschrieben werden.

ASTElement Die gemeinsame Oberklasse aller Klassen eines EAST bzw. AST ist die Klasse ASTElement. Von ihr soll es keine eigenen Instanzen geben, was durch den Stereotypen Abstract angedeutet wird. ASTElement ist, wie die meisten anderen Klassen, eine generische Klasse (vgl. Kapitel 3.3). Durch den generischen Parameter können die Klassen angegeben werden, die in dem (Unter)-Baum auftreten dürfen. Durch die Angabe von $Token \cup NonTerminal$ wird der EAST auf einen AST eingeschränkt. Dies wird später genutzt, um zwischen einem Frame (Klassen: Token, NonTerminal und Location), einem Template (Klassen: Token, NonTerminal, Repetition und MetaVariableNode) und einem AST (Klassen: Token und NonTerminal) zu unterscheiden. Des Weiteren wird der Vater der Instanzen in der Variable *parent* bereitgestellt. Dieses kann eine beliebige Unterklasse von ASTElement sein, oder wenn die Instanzen die Wurzel des Baumes ist, das Symbol Top $\top$. Die Operation *equals*, welche hier nur aus einen Deklarationsteil besteht, wird in den Unterklassen genutzt, um eine Gleichheitsrelation bereitzustellen, welche einige Knotenarten ignoriert. So wird ein MetaVariableNode, durch den Wert der Belegung der dargestellten Meta-Variablen ersetzt. Die Beschränkung auf den Deklarationsteil hat den technischen Grund, dass in Object-Z die Type-Hierarchie und die Vererbungshierarchie nicht übereinander liegen, so wie es oft in Programmiersprachen der Fall ist. Damit die Operation *equals* bei der Referenzierung über die abstrakte Oberklasse genutzt werden kann, muss *equals* hier mit seinen Parametern definiert sein. Da in dieser Klasse die Gleichheit nicht

getestet wird, ist die Operation nicht eingeschränkt. Die Operation *equals* wird nach der Einführung der EAST-Klassen auf Seite 5.9.2 behandelt.

AstElement[*CH*]

\\ Abstrakt

$\upharpoonright (parent, equals)$

$parent :\downarrow AstElement \cup \{\top\}$

INIT

equals

$other? :\downarrow AstElement[MetaVariableNode \cup RepetitonNode \cup Token \cup NonTerminal \cup LocationNode \cup NoReplaceLocationNode]$

$locations? : Binding[MetaVariableNode \cup RepetitonNode \cup Token \cup NonTerminal]$

$nrlocations? : Binding[MetaVariableNode \cup RepetitonNode \cup Token \cup NonTerminal \cup LocationNode \cup NoReplaceLocationNode]$

$binding? : Binding[Token \cup NonTerminal]$

$in? : String$

NonTerminal Den Nicht-Terminalen in einer BNF steht im AST das NonTerminal gegenüber. Ein NonTerminal setzt sich aus der Reihung von anderen Elementen zusammen. Da die Reihenfolge wichtig ist, ist dies als Sequenz in der Variable *children* modelliert. Die Sequenz ist über die generische Variable getypt, was erlaubt die möglichen Klassen in der Sequenz einzugrenzen. Die Invariante der Klasse stellt sicher, dass alle Kinder einer Instanz dieser Klasse die Instanzen auch als Vater haben. Die Äquivalenz ist erfüllt, wenn die zu vergleichende Klasse auch vom Typ NonTerminal ist und die Kinder gleich sind.

In dieser Klasse werden zwei Konstrukte genutzt, die auch bei anderen Klassen im EAST genutzt werden. Zum Einen ist es die Bedingung, dass die Variable *parent* aller Kinder auf der Instanz der Klasse zeigen muss

$$\forall\, children.parent = self$$

Dabei ist *self* eine spezielle Variable [Smi00] in Object-Z, die eine Referenz auf das Objekt beinhaltet. Dies ist ähnlich wie `this` in Java. Zusammen mit der Definition der Variable *parent* stellt dies sicher, dass die Datenstruktur wirklich einen Baum ergibt. Ohne diese Bedingungen könnte z.B. eine Instanz eines Nicht-Terminal in zwei verschiedenen Knoten verwendet werden.

Das zweite Konstrukt ist die Verwendung der Schema Operation *Scope Enrichment* (•) in der Definition der Operation *equals*. Dieser Operator ermöglicht es, die Operation *equal* in den unterschiedlichen Klassen klein zu halten, d.h. das wenige rein technische Erweiterungen benötigt werden, weil sie implizit in dem Schema Enrichment hinzugefügt werden (vgl. Kapitel 3.3 und [Smi00]).

NonTerminal[*CH*]

$\upharpoonright (parent, equals, children, typename)$

inherit *AstElement*

$children : \text{seq}\ CH$

$typename : NT$

$\forall\, children.parent = self$

INIT

$equals \mathrel{\widehat=} equalPrivate \bullet \forall\, i : \mathbb{N} \mid i < \#children \bullet children(i).equals[other?/other?.children(i)]$

equalPrivate

$other? :\downarrow MetaVariableNode \cup RepetitonNode \cup Token \cup NonTerminal \cup LocationNode \cup NoReplaceLocationNode$

$locations? : Binding[MetaVariableNode \cup RepetitonNode \cup Token \cup NonTerminal]$

$nrlocations? : Binding[MetaVariableNode \cup RepetitonNode \cup Token \cup NonTerminal \cup LocationNode \cup NoReplaceLocationNode]$

$binding? : Binding[Token \cup NonTerminal]$

$\#children = \#other?.children$

$other? \in NonTerminal[CH]$

$in? : String$

Token Die festen Schlüsselwörter und Zeichenketten im Code werden als Token bezeichnet und durch eine gleichnamige Klasse im AST dargestellt. Ein Token hat ein Bild, in welchem der String des Token abgelegt ist. Zwei Token sind gleich, wenn ihr Bild übereinstimmt.

Token[*CH*]

$\upharpoonright (parent, equals, image)$
`inherit` *AstElement*

$image : String$

INIT

equals

$other? :\downarrow MetaVariableNode \cup RepetitonNode \cup Token \cup NonTerminal \cup LocationNode \cup NoReplaceLocationNode$

$other? \in Token$
$other?.image = image$

Repetition Der Knoten Repetition bildet die Wiederholungen aus den Templates ab. Die Knoten beinhalten eine Wiederholung als EAST. Bei einem Vergleich mit einem AST können die Wiederholungen mehrmals auf den AST gemachted werden. Dieses mehrfache Matchen wird durch die wiederholte Anwendung auf die Kinder des Knoten durchgeführt. Die Funktion *cardtest* stellt sicher, dass die Anzahl der Wiederholungen sich im erlaubten Bereich befinden.

Repetition[*CH*]

$\upharpoonright (parent, equals, typename, id)$

inherit *AstElement*

$cardtest : \{*, +, ?\} \times \mathbb{N} \to \mathbb{B}$

$cardtest(*, j) = true$
$cardtest(+, j) = (0 < j)$
$cardtest(?, j) = (j <= 1)$

$typename : NT$
$id : String$
$children : \text{seq}\, CH$
$card : \{*, +, ?\}$

$\forall\, children.parent = self$

INIT

$equals \mathrel{\widehat=} equalPrivate \bullet \forall\, i : \mathbb{N} \mid i < \#other?.children$
$\bullet\ children(i \bmod \#children).equals$
$[other?/other?.children(i)]$

equalPrivate

$other? :\downarrow MetaVariableNode \cup RepetitonNode$
$\cup Token$
$\cup NonTerminal$
$\cup LocationNode$
$\cup NoReplaceLocationNode$

$other?.typename = typename$
$\exists\, n : \mathbb{N} \bullet \#other?.children = \#children * n \wedge cardtest(card, n)$
$in? : String$

MetaVariableNode Die in einem Template verwendeten Meta-Variablen werden durch den MetaVariableNode dargestellt. Wenn ein Template mit einem Stück Code verglichen wird, wird der Wert der Meta-Variablen benötigt. Die Belegungen der Meta-Variablen werden beim Vergleich (Operation *equals*) in der Variablen *binding*? übergeben. Der Vergleich muss, um den Inhalt der Meta-Variablen mit dem Code vergleichen zu können, erst eine mögliche Mehrfachbelegung aufgrund von Wiederholungen auflösen. Im EAST für das Template wird deshalb, für jedes Auftreten einer Meta-Variablen, ein MetaVariableNode erzeugt, welcher einer Index-Sequenz entsprechend ihrer Position besitzt (*mult*). In der Operation *equals* wird diese Auflösung genutzt, um den Knoten für den weiteren Vergleich zu finden.

MetaVariableNode[*CH*]

$\upharpoonright (parent, equals, name, type, mult)$

`inherit` *AstElement*

$name : String$
$type : NT$
$mult : \operatorname{seq}\mathbb{N}$
$repdeep : \mathbb{N}$

$repdeep = \#mult$

INIT

$equals \mathrel{\widehat=} ((privateEquals \bullet value.equals)[in?/name]) \setminus \{name\}$

privateEquals

$value :\downarrow AstElement$
$binding? : Binding[Token \cup NonTerminal]$

$value = binding?(name)(mult)$

LocationNode Der LocationNode ist das oberste Element eines Templates. Der LocationNode trennt den Frame von dem Template und stellt den obersten Knoten dar, der ersetzt wird. Somit hat dieser Knoten eine ähnliche Funktion wie der MetaVariableNode. Die Frames werden aber nicht in den Meta-Variablen, sondern in der Struktur *locations*? gehalten. Diese ist wie die Meta-Variablen vom Typ Binding. Dieser Knoten stellt nur die Locations dar, die nicht mit `NoReplace` gekennzeichnet sind.

LocationNode[*CH*]

$\upharpoonright (parent, equals, name, type, mult)$

`inherit` *AstElement*

$name : String$
$type : NT$
$mult : \mathrm{seq}\,\mathbb{N}$
$in : String$

INIT

$equals \mathrel{\widehat=} privateEquals \bullet value.equals$

privateEquals

$other? :\downarrow AstElement$
$value :\downarrow AstElement$
$locations? : Binding[MetaVariableNode \cup RepetitonNode \cup Token \cup NonTerminal]$
$in? : String$

$value = locations?(name)(mult)$

NoReplaceLocationNode Die Locations, die mit dem Schlüsselwort `NoReplace` gekennzeichnet sind, werden nicht durch das Refactoring verändert. Da dies von den anderen Locations signifikant abweicht, ist diese Funktionalität in einer separaten Klasse gekapselt. Die Grundidee entspricht der der Locations. Funktional besteht der Unterschied darin, dass die NoReplaceLocationNodes nicht nur im BeforeTemplate eingebunden werden, sondern auch im AfterTemplate, was eine unveränderte Einsetzung bewirkt. Die Modellierung als separate Klasse hat weitere Vorteile. Das Verbot, innerhalb eines Teil-AST der von einem LocationNode ausgeht ein NoReplaceLocationNode zu verwenden, kann einfach über die generischen Parameter der Klassen gesteuert werden.

NoReplaceLocationNode[*CH*]

$\upharpoonright (parent, equals, name, type, mult)$

inherit *AstElement*

$$
\begin{array}{|l}
\hline
name : String \\
type : NT \\
mult : \operatorname{seq}\mathbb{N} \\
\hline
\#mult = 1 \\
\hline
\end{array}
$$

INIT

$equals \mathrel{\widehat=} privateEquals \bullet value.equals$

privateEquals

$$
\begin{array}{|l}
\hline
other? :\downarrow AstElement \\
value :\downarrow AstElement \\
nrlocations? : Binding[MetaVariableNode \cup RepetitonNode \\
\qquad\qquad \cup Token \\
\qquad\qquad \cup NonTerminal \\
\qquad\qquad \cup NoReplaceLocationNode \\
\qquad\qquad \cup LocationNode] \\
in? : String \\
\hline
value = nrlocations?(name)(mult) \\
\hline
\end{array}
$$

Damit sind die Knoten des (E)AST erklärt. Bevor die Refactoringsklassen beschrieben werden, soll vorher der zentrale Vergleich-Mechanismus (Operation *equals*) erklärt werden. Dieser Vergleich wird von den Templates genutzt, um beim BeforeTemplate aus dem AST des Codes die Belegung der Locations und der Meta-Variablen zu erzeugen bzw. beim AfterTemplate aus den Belegungen einen neues Code-AST. Im Prinzip werden die speziellen Knoten LocationNode, MetaVariableNode, NoReplaceLocationNode und RepetitionNode im Vergleich durchsichtig. Sie werden also übergangen und stattdessen der Vergleich mit den entsprechenden Kindern fortgesetzt. Dies ist in Abbildung 5.10 dargestellt. Auf der linken Seite ist der Code-AST dargestellt, gegenüber einem EAST, der nach der *equals* Operation äquivalent ist. Der mittlere Block zeigt dabei den EAST-Ausschnitt des Templates. Der Frame ist so gewählt, dass die LocationNodes, genau auf die Knoten des Templates passen. Ähnlich ist es beim Übergang zu den Meta-Variablen unten. Wenn die speziellen Knoten aus dem Graph entfernt wären und die Kinder zu Kindern des Vaters gemacht werden, ergibt sich der selbe Graph wie auf der linken Seite. Auf diesem Vergleich bauen die beiden Hauptoperationen der Templates auf. Bei einem BeforeTemplate ist die linke Seite (Code-AST) gegeben und es wird zu gegebenen Templates ein entsprechender EAST gesucht. Die Nutzung beim AfterTemplate ist genau entgegengesetzt. Hier ist die rechte Seite gegeben und es wird ein passender Code-AST gesucht.

Mit den eingeführten Strukturen werden die Klassen für die einzelnen Refactorings definiert.

AbstractRefactoringSemantik und allgemeine Konstrukte Die Klasse AbstractRefactoringSemantik ist die gemeinsame Oberklasse für alle Refactorings. Diese Klasse erlaubt es, bei der Verwendung von Refactorings in einem zusammengesetzten Refactoring nicht zwischen verschiedenen Refactoringklassen unterscheiden zu müssen. In dieser Klasse ist daher nur die Refactoring Operation definiert, welche genutzt wird, um das Refactoring durchzuführen.

AbstractRefactoringSemantik

$\uparrow (refactoring)$

$\backslash\backslash Abstrakt$

INIT

refactoring

$src? : AST[Token \cup NonTerminal]$

$src! : AST[Token \cup NonTerminal]$

Semantik: Simple Refactoring In diesem Abschnitt wird die Struktur der Klasse *SimpleRefactoringSemantik*, die die Semantik eines einfachen Refactorings abbildet, erläutert. Die Klasse enthält noch Lücken, die im Rahmen der Transformation in die Semantik geschlossen werden müssen. Dies wird weiter unten im Abschnitt erläutert.

Struktur der Klasse *SimpleRefactoringSemantik* In der Klasse selber werden nur die für die Refactoring-Beschreibung relevanten Daten gehalten und definiert. Daten, die

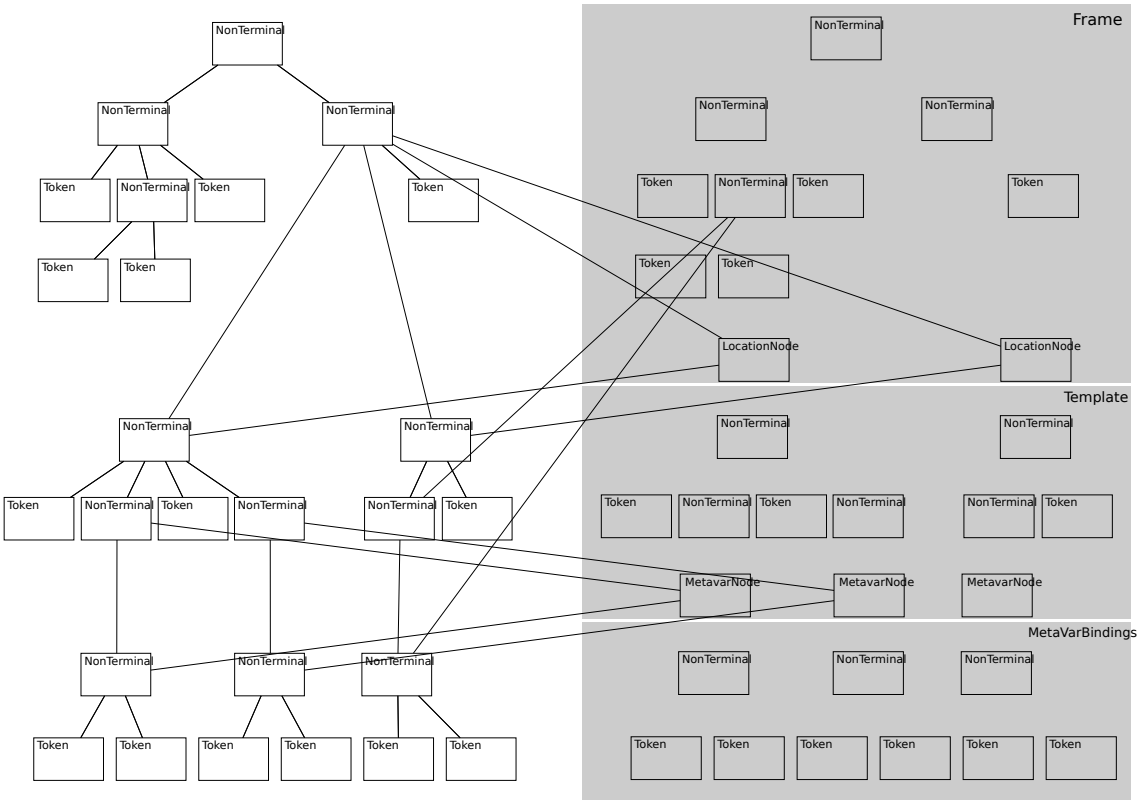


Abbildung 5.10: Gegenüberstellung eines AST und eines EAST, welche als äquivalent angesehen werden

für die Durchführung gebraucht werden, wie die Belegung der Meta-Variablen, werden als Parameter zwischen den Teilen ausgetauscht. Dadurch wird eine saubere Trennung zwischen der Semantik von Re$\mathcal{L}$ und der Ausführung des Refactorings in dieser Semantik erreicht.

Das zentrale Element der Klasse bildet die Operation *refactoring*. Diese Operation berechnet aus einem gegebenen Code in Form eines AST *src*? und den Parametern *params*? neuen Code, der wieder als AST ausgegeben wird. Das grobe Vorgehen (Abb. 5.11) in einem einfachen Refactoring besteht darin, dass sowohl das BeforeTemplate (BT) und die Precondition (*Pcon*) erfüllt sein müssen. Dabei wird eine Belegung der Meta-Variablen und des Frames des Refactorings gefunden. Der Frame stellt den unveränderten Teil des Codes dar. Die Meta-Variablen werden an die Calculation gegeben, in der weitere Meta-Variablen berechnet werden können. Mit der Meta-Variablenbelegung wird im AfterTemplate (AT) der Frame wieder mit Code gefüllt. Zum Verständnis über den Zusammenhang mit den Transformern, bzw. für die Darstellung des Datenflusses, sind diese in der Graphik angegeben, obwohl diese nicht in der Operation enthalten sind.

Von den einzelnen Bestandteilen der Operation *Refactoring* sind die Precondition *Pcon* und die Calculation die Einfachsten. Beide sind Operationen, die als einfache Operationsschemata gegeben sind. In der Precondition werden die erlaubten Meta-Variable-Belegungen festgelegt, deshalb hat die Operation *Pcon* nur einen Parameter, den Ausgabeparameter *binding*!, welcher Belegungen von Meta-Variablen enthält und nur AST-Elemente enthalten darf. Die Calculation ist ähnlich aufgebaut. Diese nimmt als Parame-

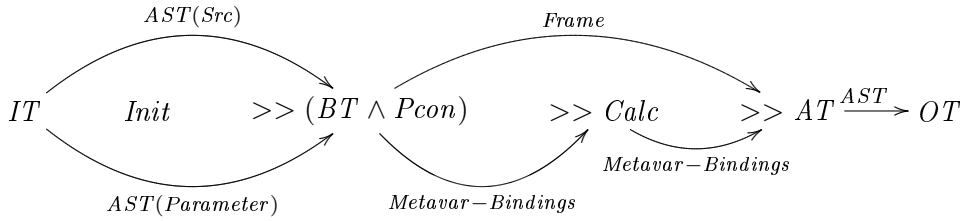


Abbildung 5.11: Übersicht über die Struktur eines einfachen Refactorings

ter zusätzlich eine existierende Belegung (*binding?*), welche unverändert in die berechnete Belegung übernommen wird (*binding?* $\subset$ *binding!*).

Das Vorgehen beim Before- bzw. AfterTemplate ist von der Idee sehr ähnlich. Es wird ein AST mit einem EAST verglichen. AST und EAST werden als gleich angesehen, wenn durch Eliminieren bzw. Einsetzen der Werte von LocationNodes, RepetionNode und MetaVariableNode ein identischer AST produziert wird.

Der hauptsächliche Unterschied zwischen der Anwendung dieser Gleichheit bei dem Before- und AfterTemplate ist, welche Seite der Äquivalenz gesucht wird. Beim BeforeTemplate wird ein EAST gesucht. Dieser EAST muss zusätzlich noch der Typ-Bedingungen, die aus der Deklaration der Meta-Variablen resultiert, genügen. Für die LocationNode besteht eine ähnliche Einschränkung über den Scope.

Beim AfterTemplate wird ein AST gesucht, der dem EAST entspricht. Dies ist ohne weitere Nebenbedingungen möglich.

SimpleRefactoringSemantik

$\upharpoonright$ (*Refactoring*)

`inherit` *AbstractRefactoringSemantik*

$$is : Value[NonTerminal \cup Token] \times NonTerminal$$

$$\forall\, a : \text{ran } Value[NonTerminal \cup Token];\ b : NonTerminal \bullet$$
$$a\ is\ b \Leftrightarrow b \in \mathbf{stype}(a.name)$$

$$invar : Varname \times AST[NonTerminal \cup Token] \rightarrow \mathbb{B}$$

Siehe Seite 127 - Muss berechnet werden

$$bts : \mathbb{P}\, Binding[MetaVariableNode \cup RepetitonNode \cup Token \cup NonTerminal]$$
$$ats : \mathbb{P}\, Binding[MetaVariableNode \cup RepetitonNode \cup Token \cup NonTerminal]$$
$$vars : \mathbb{P}\, Varname$$
$$type : Varname \nrightarrow NT$$

$$\text{dom } type = vars$$

Init

$vars = \varnothing$
$type = ...$
$bt = ...$
$at = ...$
Siehe Seite 126

Pcon

$binding! : Binding[Token \cup NonTerminal]$

Siehe Seite 127 - Muss berechnet werden
(Variablen müssen durch binding?('Varname') ersetzt werden.)

Calc

$binding!, binding? : Binding[Token \cup NonTerminal]$

$binding? \subset binding!$
$\forall m : \mathrm{dom}\, binding! \bullet \forall v : \mathrm{dom}\, m \bullet invar(v)(bel(v))$
Siehe Seite 127 - Muss berechnet werden
(Variablen müssen durch binding?('Varname') ersetzt werden.)

$Refactoring \mathrel{\widehat=} Init >> (BT \wedge Pcon) >> Cal >> AT$

$BT \mathrel{\widehat=} BTHelp \wedge f!.equals[locations!/bt;\ binding?/bindings!;\ other?/src?]$

BTHelp

$f! : AST[Token \cup NonTerminal \cup LocationNode \cup NoReplaceLocationNode]$
$binding! : Binding[Token \cup NonTerminal]$
$params? : Binding[Token \cup NonTerminal]$
$src? : AST[Token \cup NonTerminal]$
$bt : Binding[MetaVariableNode \cup RepetitonNode \cup Token \cup NonTerminal]$

$param? \subset meta!$
$bt \in bts$
$\forall m : \mathrm{dom}\, binding! \bullet \forall v : \mathrm{dom}\, m \bullet bel(v)$ is $type(v)$
$\forall m : \mathrm{dom}\, binding! \bullet \forall v : \mathrm{dom}\, m \bullet invar(v)(bel(v))$
$\mathrm{dom}\, binding! \subseteq vars$

$AT \mathrel{\widehat=} (ATHelp \wedge f?.equals[locations!/at;\ other?/result!]$

ATHelp

$f? : AST[Token \cup NonTerminal \cup LocationVars]$
$binding? : Binding[Token \cup NonTerminal]$
$result! : AST[Token \cup NonTerminal]$
$at : Binding[MetaVariableNode \cup RepetitonNode \cup Token \cup NonTerminal]$

$at \in ats$

Übertragung eines einfachen Refactorings Im letzten Abschnitt ist die allgemeine Struktur der Klasse *SimpleRefactoringSemantik* beschrieben worden. In diesem Abschnitt soll beschrieben werden, wie ein einfaches Refactoring abgebildet wird. Dazu wird als erstes eine Kopie der allgemeinen Struktur mit dem Namen des Refactorings angelegt und anschließend die Lücken in der Definition geschlossen. Zur Veranschaulichung wird diese Umsetzung wieder an dem Refactoring aus Listing 5.1 durchgeführt.

Als Erstes sollen die Lücken im INIT der Klasse geschlossen werden: Die Zuordnung der Typen zu den Variablennamen muss definiert werden. Diese werden direkt aus den Parametern und den Meta-Variablen-Definitionen bestimmt. Für jede deklarierte Meta-Variable wird eine Zuordnung $Name \mapsto Type$ erstellt. Die Menge aller dieser Zuordnungen stellt den Wert der Variable *type* dar. Auch die Struktur der Templates wird in der Initialisierung aufgebaut. Da diese durch den Text der Templates vorgegeben sind und als EAST in der Semantik genutzt werden, können zur Definition erweiterte Transformatoren (EIT) genutzt werden:

INIT

$$
\begin{aligned}
&vars = \varnothing \\
&type = \{functionname \mapsto FUNCNAME, \\
&\qquad operations \mapsto P, \\
&\qquad a \mapsto P, \\
&\qquad b \mapsto P, \\
&\qquad f \mapsto function, \\
&\qquad parameterlist \mapsto ParList, \\
&\qquad returnvar \mapsto Var\} \\
&bts = EIT(\texttt{"Scope. FWHILE ...\#z(f)*"}) \\
&ats = EIT(\texttt{"Scope. FWHILE ...endfunc"})
\end{aligned}
$$

Die erweiterten Transformatoren übertragen den Code der Templates in zwei Schritten. Im ersten Schritt wird der Code geparst. Im zweiten Schritt werden aus den geparsten Templates alle möglichen Belegungsfunktionen für die Templates gebildet. Insgesamt entspricht das Vorgehen einer Typdefinition von Bindings, die auf bestimmte Elemente eingeschränkt ist. Ein Binding ist eine Funktion von Namen in die Belegung-Elemente. In der Anwendung bei den Templates sind die Namen die Indexe der Subtemplates. Die Belegungswerte entsprechen den EAST-Bäumen der Subtemplates. Der Typausdruck der Bindings stellt die Menge aller dieser Funktionen, insbesondere auch mit allen möglichen Templates dar. Dies beinhaltet, dass alle Templateausfaltungen in dieser Menge existieren. Durch den geparsten Ausdruck wird diese Menge nun eingeschränkt, so dass nur noch Elemente enthalten sind, die dem geparsten Template entsprechen. Das heißt, dass die Menge der Namen im Binding auf die Indexnamen beschränkt sind und dass die Werte der Funktion den Subtemplate genügen.

$$
\begin{aligned}
&EIT : Code \to \mathbb{P}\, Binding[MetaVariableNode \cup RepetitonNode \\
&\qquad\qquad \cup Token \cup NonTerminal] \\
&\hline
&EIT(x) = \{b \in Binding[MetaVariableNode \cup RepetitonNode \\
&\qquad\qquad \cup Token \cup NonTerminal] \\
&\qquad\qquad \mid parsetest(x, b)\}
\end{aligned}
$$

Die Funktion *parsetest* testet, ob ein Binding wie oben beschrieben den Template genügt.

Der nächste Schritt ist die Übertragung der Precondition und der Calculation. Da die Bedingungsteile beider Refactoringskomponenten schon als Z-Prädikate gegeben sind, müssen in ihnen nur noch die Meta-Variablenbenutzungen angepasst werden. Dazu ist in den Prädikaten jedes Auftreten einer Variable durch den Verweis auf den Wert der entsprechenden Variablenbelegung zu ersetzen. Dies geschieht durch Ersetzen der Meta-Variablennutzung durch den Term $binding?('Varname')$, wobei *Varnamen* durch den entsprechenden Variablennamen ersetzt wird. Dies ist nötig, da die Meta-Variablen nicht explizit in der Refactoringklasse vorhanden sind, sondern als Binding vorliegen. Daher muss der Wert an der Stelle der Bindingfunktion statt der Meta-Variablen genutzt werden. Somit wird aus der Vorbedingung

```
Pre:
    \forall a \in \names(f) \bullet a != functionname
        | writevars(operations) \cap readvars(b) | = 1
```

das Pcon-Schema

$$
\begin{array}{|l}
\hline
Pcon \\
binding! : Binding[Token \cup NonTerminal] \\
\hline
\forall\, a \in \mathbf{names}\,(binding?('f'))\bullet \\
\qquad\qquad binding?('functionname') \neq binding?('functionname') \\
\mid writevars(binding?('operations')) \cap readvars(binding?('b')) \mid = 1 \\
\hline
\end{array}
$$

Entsprechend wird in der Calculation aus

```
Calc:
    parameterList = readvars(operations)                                    2
    returnvar \in writevars(operations) \cap readvars(b)
```

das Calc-Schema

$$
\begin{array}{|l}
\hline
Calc \\
binding!, binding? : Binding[Token \cup NonTerminal] \\
\hline
binding? \subset binding! \\
\forall\, m : \mathrm{dom}\, binding! \bullet \forall\, v : \mathrm{dom}\, m \bullet invar(v)(bel(v)) \\
binding!('parameterList') = readvars(binding!('operations')) \\
binding!('returnvar') \in writevars(binding!('operations')) \\
\qquad\qquad \cap readvars(binding!('b'))binding?('Varname') \\
\hline
\end{array}
$$

durch die Ersetzung durch $binding!('Varname')$. Im Calc-Schema kann gefragt werden, wieso alle Ersetzungen mit der Ausgabe der Belegungen durchgeführt werden und nicht die Eingaben an einigen Stellen verwendet werden. Dies ist nicht nötig, da verlangt wird, dass die Eingabebelegung vollständig in der Ausgabebelegung eingebettet ist. Dadurch vereinfacht sich der Prozess der Umsetzung in die Semantik.

Nun werden die Schemata der Semantik-Klasse ausgefüllt, somit bleibt die axiomatische Definition *invar* zu vervollständigen. Die Definition von *invar* stellt zu jeder Meta-Variablen eine Funktion, die die Invariante dieser Meta-Variablen prüft, zur Verfügung. Die Invariante wird im BeforeTemplate und der Calculation geprüft. Die Invariante der Parameter werden nicht explizit geprüft, da sie beim BeforeTemplate implizit mit geprüft werden.

Damit ist das semantische Mapping für ein einfaches Refactoring vollständig.

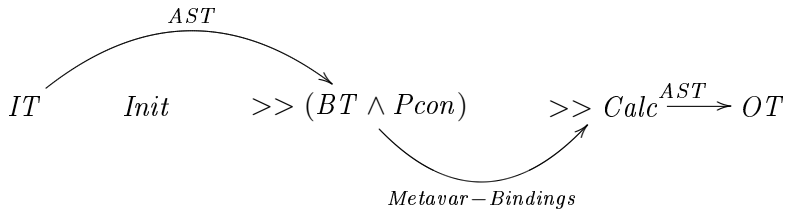


Abbildung 5.12: Übersicht über die Struktur eines zusammengesetzten Refactorings

Zusammengesetztes Refactoring Ein zusammengesetztes Refactoring unterscheidet sich von einem einfachen Refactoring darin, dass die Calculation eine Ausführung anderer Refactorings statt eine einfache Berechnung von Werten durchführt und das AfterTemplate zwingend leer sein muss. Die anderen Bestandteile (BeforeTemplate und Precondition) sind unverändert.

Der Aufbau der Klasse CompositRefactoringSemantik ist ähnlich dem Aufbau der Klasse für einfache Refactorings. Die Teile, die die Funktion des AfterTemplates abbilden, sind entfernt worden. Auch ist der Aufruf des AfterTemplates aus der Operation *Refactoring* entfallen. Die Operation *Calc* hat ein anderes Aussehen bekommen. Statt Bindings von Meta-Variablen zu berechnen, werden andere Refactorings aufgerufen.

CompositRefactoringSemantik

$\upharpoonright (Refactoring)$

inherit *AbstractRefactoringSemantik*

$is : Value[NonTerminal \cup Token] \times NonTerminal$

$\forall\, a : \text{ran}\ Value[NonTerminal \cup Token];\ b : NonTerminal \bullet$
$a\ is\ b \Leftrightarrow b \in \mathbf{stype}(a.name)$

$invar : Varname \times AST[NonTerminal \cup Token] \to \mathbb{B}$

Siehe Seite 127 - Muss berechnet werden

$bts : \mathbb{P}\, Binding[MetaVariableNode \cup RepetitonNode \cup Token \cup NonTerminal]$
$vars : \mathbb{P}\, Varname$
$type : Varname \nrightarrow NT$

$\text{dom}\ type = vars$

INIT

$vars = \varnothing$
$type = ...$
$bts = ...$
Siehe Seite 126

Pcon

$binding! : Binding[Token \cup NonTerminal]$

Siehe Seite 126
(Variablen müssen durch binding?('Varname') ersetzt werden.)

$Calc \mathrel{\widehat=} CalcHelp$ ... Siehe Seite 129 - Muss berechnet werden

CalcHelp

$binding!, binding? : Binding[Token \cup NonTerminal]$

$binding? \subset binding!$
$\forall m : \mathrm{dom}\, binding! \bullet \forall v : \mathrm{dom}\, m \bullet invar(v)(bel(v))$
Siehe Text - Muss berechnet werden
(Variablen müssen durch binding?('Varname') ersetzt werden.)

$BT \mathrel{\widehat=} BTHelp \land f!.equals[locations!/bt;\ binding?/binding!;\ other?/src?]$

BTHelp

$f! : AST[Token \cup NonTerminal \cup LocationVars]$
$binding! : Binding[Token \cup NonTerminal]$
$params? : Binding[Token \cup NonTerminal]$
$src? : AST[Token \cup NonTerminal]$
$bt : Binding[MetaVariableNode \cup RepetitonNode$
$\quad \cup Token \cup NonTerminal]$

$bt \in bts$
$param? \subset meta!$
$\forall m : \mathrm{dom}\, binding! \bullet \forall v : \mathrm{dom}\, m \bullet bel(v)$ is $type(v)$
$\forall m : \mathrm{dom}\, binding! \bullet \forall v : \mathrm{dom}\, m \bullet invar(v)(bel(v))$
$\mathrm{dom}\, binding! \subseteq vars$

$Refactoring \mathrel{\widehat=} Init >> (BT \land Pcon) >> Cal$

Übertragung eines zusammengesetzten Refactorings Die Übertragung eines zusammengesetzten Refactorings ist der Übertragung eines einfachen Refactorings ähnlich. Das BeforeTemplate, die Precondition, die Meta-Variablen und die axiomatischen Definitionen werden genauso übertragen. Die Umsetzung AfterTemplate entfällt, da es in einem zusammengesetzten Refactoring immer leer ist.

Somit bleibt noch die Übersetzung der Calculation. Die Calculation besteht im zusammengesetzten Refactoring aus zwei Teilen. Dem Berechnungsteil, wie im einfachen Refactoring und der Zusammensetzung der anderen Refactorings. Die Berechnungen werden wie beim einfachen Refactoring erstellt und befinden sich in diesem Fall in der Operation *CalcHelp*. Diese Funktion berechnet Hilfswerte und wird in die Operation *Calc* eingebaut. Die Operation *Calc* wird mittels folgender Regeln erzeugt:

C_1 `";"` $C_2 \Rightarrow (C_1) >> (C_2)$
`"IF"` P `"THEN"` C_1 `"ELSE"` $C_2 \Rightarrow ((P \land C_1) \square (\neg P \land C_2))$
`"FOR EACH"` v `"IN"` Exp `"DO"` C `"ENDFOREACH"` $\Rightarrow^{>>}_{v \in Exp}\ C$

Etwas komplizierter ist die Übertragung des Aufrufs eines anderen Refactorings. Beim Aufruf eines Refactorings muss eine initialisierte Instanz der Refactoringklasse vorliegen und anschließend die Parameter berechnet werden. Dann kann das Refactoring mittels der Operation *refactorings* aufgerufen werden. Die Umsetzung der Parameter benutzt den Umsetzungmechanismus von oben: Jedes Auftreten einer Variablen in dem Ausdruck hinter der Zuweisung wird durch $binding!(Varname)$ ersetzt. Dieser ersetzte Term wird mit $ZExpression1$ bezeichnet.
Somit wird jeder Parameter mit seiner Zuweisung in einen Term wie folgt umgewandelt:

$$Identifierer = ZExpression1$$
$$\Rightarrow newBinding('Identifierer') = ZExpression1$$

Mit Hilfe der umgewandelten Parameter wird ein Hilfsschema aufgebaut. In der Umsetzung wird dies in Form der Inline-Schreibweise durchgeführt, welches zum besseren Verständnis hier als separates Schema angegeben ist.

Hilfsschema

$newBinding =\downarrow Binding[Token \cup NonTerminal];$
$ref : Identifierer$

//Umsetzungen der Parameter wie oben beschrieben

In dem Hilfsschema wird die Variable *newBinding* definiert und kann nur einen TeilAST enthalten. Des Weiteren wird eine Variable *ref* definiert, welche die Klasse für das genutzte Refactoring enthält. Damit ergibt sich der eigentliche modellierte Aufruf:

$$((Hilfsschema \bullet (ref.Init \fatsemi ref.refactoring[binding?/newBinding]) \setminus \{newBinding, ref\})$$

Dabei werden die Variablen der technischen Hilfskonstrukte versteckt, so dass sie nicht mit anderen Variablen gleichen Namens in Konflikt geraten. Die Angabe des Hilfsschema erfolgt in der Inline-Fassung ($[\ldots \mid \ldots]$), nicht wie hier als explizites Schema.

Damit ist die Umsetzung eines zusammengesetzten Refactorings abgeschlossen. In diesem Unterkapitel sind die Hilfsstrukturen (z.B. (E)AST) und die Umsetzung der Refactorings in Object-Z-Klassen beschrieben worden. Damit ist die Transformation in die semantische Domäne abgeschlossen. In den nächsten Abschnitten werden Wohlgeformtheitseigenschaften dieser Refactorings behandelt.

5.10 Soundness

Ein Refactoring ändert ein Programm oder eine Spezifikation. Nach den Änderungen sollte das Ergebnis ein gültiges Programm oder eine gültige Spezifikation sein, welches ihr Verhalten nicht geändert hat. Ein Refactoring ist gültig bzw. Sound, wenn nach der Transformation ein syntaktisch und wohlgetyptes Programm oder Spezifikation entsteht. In den folgenden beiden Abschnitten wird die Soundness betrachtet. Die Verhaltenserhaltung ist Thema des Kapitels 7.2. In Abschnitt 5.2 wurden diese beiden Eigenschaften als wünschenswert für Refactorings herausgestellt. Im Folgenden wird dargestellt, wie diese Eigenschaften in Re$\mathcal{L}$ sichergestellt werden.

5.10.1 Syntax-Korrektheit des Refactorings

Refactorings sollen verhaltenserhaltend sein. Dafür ist es eine wichtige Voraussetzung, dass der erzeugte Code oder das erzeugte Modell syntaktisch korrekt ist, denn die Definition der Verhaltenserhaltung setzt einen syntaktisch korrekten und wohlgeformten Code voraus.

Die Prüfungen in diesem Abschnitt sind nur für ein einfaches Refactoring interessant, wie am Ende des Abschnittes beleuchtet werden soll.

Die Überprüfung der syntaktischen Korrektheit muss auf Grundlage der Transformation geprüft werden können. Diese Überprüfung kann in Re$\mathcal{L}$ mithilfe des AfterTemplates und der Deklarationen der Meta-Variablen durchgeführt werden.

Die Grundidee besteht darin, zu prüfen, ob für eine beliebig gültige Belegung der Meta-Variablen durch das AfterTemplate syntaktisch korrekter Code erzeugt wird.

Dabei müssen mehrere Sachverhalte geprüft werden. Als Erstes muss geprüft werden, ob das AfterTemplate der Syntax bzgl. dem Scope genügt. Dies kann im Falle einer technischen Umsetzung automatisch durch einen (geeigneten) Parser geprüft werden. Dazu wird ausgenutzt, das durch den Scope die Produktion bekannt ist, dem das Template genügen muss. Diese Prüfung stellt zusammen mit der inneren Wohlgeformtheit sicher, dass an der Location des Templates ein gültiger Code entsteht. Von der inneren Wohlgeformtheit wird ausgenutzt, dass die Scopes an einer ersetzten Stelle kompatibel sind. Ohne diese Kompatibilität wäre es nicht ausreichend zu prüfen, ob der Scope syntaktisch erfüllt wird.

Anschließend wird geprüft, ob die Meta-Variablen bei ihrem Auftreten vom erwarteten Typ bzw. entsprechendem Subtyp sind. Diese Überprüfung stellt sicher, dass bei jedem Auftreten der Meta-Variablen gültiger Code entsteht.

Die letzte Prüfung ist eine Prüfung, die von der Zielsprache abhängt. Es müssen alle Wohlgeformtheitsbedingungen der Zielsprache erfüllt sein. Diese Eigenschaften werden im Compilerbau oft innerhalb der *statischen Semantik* betrachtet. Hier wird dies nicht gemacht, da einige andere Teile aus der statischen Semantik für die Verhaltenserhaltung genutzt werden und die Eindeutigkeit gewahrt bleiben soll. Die äußere Wohlgeformtheit (das heißt die Wohlgeformtheit bzgl. der Zielsprache) muss somit separat geprüft werden, weil sie nicht in allen Zielsprachen gleich ist. In vielen Sprachen wird z.B. gefordert, dass alle Variablen vor ihrer Nutzung deklariert sein müssen. Diese Forderung muss dann anhand der Templates geprüft werden.

Zusammengefasst sind folgende Prüfungen für die syntaktische Korrektheit durchzuführen:

1. (a) Prüfen, ob das Template den Scope erfüllt:
   Das AfterTemplate muss ein Code-Fragment enthalten, der an dem Scope verwendet werden darf.

   (b) Korrektheit des Templates:
   Das Template muss seiner Syntax genügen.

2. Prüfen, ob Meta-Variablen korrekt sind:
   Es ist zu prüfen, ob der benutzte Nicht-Terminal-Typ der Meta-Variablen an der Verwendungsstelle erlaubt ist.

3. Prüfen der äußeren Wohlgeformtheit

Im Beispiel aus Listing 5.1 muss das AfterTemplate geprüft werden:

```
AfterTemplate:
 Scope: FWHILE                                                                    2
     main
     a;
     returnvar = functionname(parameterList);
     b
     endmain                                                                      7
     #Ifunc(f)*
     func functionname (parameterList)
     operations
     return returnvar;
     endfunc                                                                     12
```

Als Erstes muss die Syntax und der Scope des Templates geprüft werden. Der Scope ist `FWHILE`. Mit den Produktionen von FWHILE (vgl. Listing 2.1) kann die Struktur geprüft werden. Für die Meta-Variablen müssen die Deklarationen betrachtet werden:

```
parameterList : PARAM;
returnvar     : VAR;
functionname  : FNAME;                                                            3
operations    : MSTAT;
a,b           : MSTAT;
f             : FUNC;
```

Für jede Meta-Variable muss geprüft werden, ob ihr Typ an der Nutzungsstelle erlaubt ist. Die Meta-Variable `parameterList` wird an zwei Stellen genutzt. Die erste Nutzung wird durch die Produktion

```
FUSE = FNAME "(" PARAM ")"
```

beschrieben. An der Stelle an der die Meta-Variable genutzt wird, wird die Produktion `PARAM` erwartet. Die Variablennutzung ist korrekt, da die Variable vom Typ PARAM ist. Auf diese Weise müssen alle Nutzungen von Meta-Variablen geprüft werden. Diese Syntax-Prüfung ist vollständig automatisierbar und wird bei der Nutzung des ReL-Tools (vgl. Kapitel 5.12) durch das Tool geprüft. Beim Scope wird die Struktur der Grammatik von ReL ausgenutzt. Die Produktion `NTSCOPETYPE` ist aus den Namen des Scopes und der entsprechenden Produktion aufgebaut. Ein Subtemplate kann daher nur erfolgreich geparsed werden, wenn dass Subtemplate syntaktisch korrekt ist. Die Prüfung, ob die Meta-Variablen Benutzungen korrekt sind, wird mittels gesteuertem Lookahead sichergestellt.

Die hier aufgeführten Prüfungen ergeben nur für ein einfaches Refactoring Sinn, was ausreichend ist. Für ein zusammengesetztes Refactoring müssen die entsprechenden Prüfungen nicht durchgeführt werden, da in einem zusammengesetzten Refactoring andere Refactorings kombiniert werden und dort selber kein Code verändert wird. Die entsprechenden Eigenschaften sind transitiv, wenn zwei Refactorings hintereinander durchgeführt werden und jedes syntaktisch korrekten Code erzeugt. In der Folge erzeugt auch die Hintereinanderausführung syntaktisch korrekten Code.

5.10.2 Wohlgeformtheit des Refactorings

Wenn ein Code vor einem Refactoring wohlgeformt war, muss er auch nach dem Refactoring wieder wohlgeformt sein. Dieses wird nicht automatisch durch ein Refactoring sichergestellt

und muss separat überprüft werden. Für das Refactoring aus Listing 5.1 müssen

usedFunc(P) $\subset$ set(definedFunc(P)) für P ist FWHILE-Programm
set(definedFunc(P)) = definedFunc(P) für P ist FWHILE-Programm
$\forall f$: definedFunc(P) $\bullet$ calcReturn(f) $\neq$ ERR
calcReturn(`functionname`) = **int**

erfüllt sein.

Satz 30.1 *Das Refactoring aus Listing 5.1 erhält die Wohlgeformtheit des Programms.*

Beweis: Es ist zu zeigen, dass aus der Wohlgeformtheit des Programms P die Wohlgeformtheit des refaktorisierten Programms P' folgt.

- Zu zeigen:

  usedFunc(P) $\subset$ set(definedFunc(P)) $\Rightarrow$ usedFunc(P') $\subset$ set(definedFunc(P'))

  Aus der Definition von usedFunc folgt, dass für P' durch die Zeile `returnvar = functionname(parameterList);` die Funktion mit dem Namen in der Meta-Variable `functionname` genutzt wird. Da durch das AfterTemplate keine Nutzung wegfällt, gilt:

  usedFunc(P') = usedFunc(P) $\cup$ {`functionname`}

  Entsprechend kann aus

```
#Ifunc(f)*
func functionname (parameterList)
operations
return returnvar;                                                    4
endfunc
```

  gefolgert werden, dass

  definedFunc(P') = definedFunc(P) $\cup$ {`functionname`}

  ist. Wenn dies auf die Wohlgeformtheitsbedingung angewendet wird, ergibt dies:

  usedFunc(P) $\cup$ {`functionname`} $\subset$ set(definedFunc(P) $\cup$ {`functionname`})

  Da die Vereinigung bei Mengen und die Vereinigung bei Multimengen entsprechend definiert ist, kann die Mengenumwandlung aufgeteilt werden.

  usedFunc(P) $\cup$ {`functionname`} $\subset$ set(definedFunc(P)) $\cup$ {`functionname`}

  Weil das Programm vor dem Refactoring wohlgeformt ist, ist das äquivalent zu:

  {`functionname`} $\subset$ {`functionname`}

- Zu zeigen:

  $$set(\text{definedFunc}(P)) = \text{definedFunc}(P) \Rightarrow set(\text{definedFunc}(P')) = \text{definedFunc}(P')$$

  Dies gilt, weil aus

  $$\forall\, a \in \mathbf{names}(f) \bullet a \neq \mathit{functionname}$$

  folgt, dass

  $$\texttt{functionname} \notin \text{definedFunc}(P)$$

  ist.

- Zu zeigen:

  $$\forall f : \text{definedFunc}(P) \bullet \text{calcReturn}(f) \neq ERR \Rightarrow \forall f : \text{definedFunc}(P') \bullet \text{calcReturn}(f) \neq ERR$$

  Wieder wird ausgenutzt, dass

  $$\text{definedFunc}(P') = \text{definedFunc}(P) \cup \{\texttt{functionname}\}$$

  gilt. Daher muss nur gezeigt werden, dass

  $$\text{calcReturn}(\texttt{functionname}) \neq ERR$$

  gilt. Der Wert der Funktion calcReturn ist bei dieser Funktion nur von der Return-Anweisung abhängig, da die anderen Operationen in `operations` aus dem Hauptprogramm kommen, wo keine Return-Anweisung erlaubt ist.

  ```
  func functionname (parameterList)
  operations
  return returnvar;
  endfunc
  ```

  Die Return-Anweisung beinhaltet eine Integer-Variable und daher ist der Rückgabetyp **int** .

Damit sind alle Teile der Wohlgeformtheit des FWHILE-Programms nach dem Refactoring gezeigt. □□

Bei komplizierteren Wohlgeformtheitsbedingungen können Techniken genutzt werden, wie sie später für den Beweis der Verhaltenserhaltung vorgestellt werden.

5.10.3 Aufruf-Soundness bei zusammengesetzten Refactorings

Der Beweis der Aufruf-Soundness ist hinreichend für die Verhaltenserhaltung eines zusammengesetzten Refactorings, wenn die Subrefactorings echte Refactorings sind. Dies folgt direkt aus der Transitivität der Verhaltenserhaltung (vgl. Definition 28 und Kapitel 6).

5.11 Vergleich mit Ansätzen anderer Refactoring-Sprachen

In den ersten Abschnitten dieses Kapitels sind mehrere Ansätze zur Beschreibung von Refactorings vorgestellt worden. Einige werden in diesem Anschnitt mit Re$\mathcal{L}$ verglichen. Im Gegensatz zu den meisten anderen Ansätzen, liegt eine Besonderheit von Re$\mathcal{L}$ in der universellen Anwendbarkeit für BNF-basierte Sprachen. Diese Universalität umfasst nicht nur die Einsetzbarkeit für verschiedene Sprachen, sondern auch die Verwendbarkeit für verschiedene Nutzungsdomänen (Kommunikation, Implementation und formale Analyse).

5.11.1 (Pre, T) - Ansatz nach Roberts [Rob99]

Roberts [Rob99] beschreibt Refactorings als Tupel aus einer Vorbedingung (Pre) und aus der eigentlichen Transformation. Die Transformation wird von Roberts dabei als eine Art Nachbedingung für das Refactoring angegeben, so dass das Aussehen eines Refactorings dem Hoare-Kalkül [Hoa69] ähnelt, wobei im Innern der Code verändert wird.

Das Vorgehen von Roberts hat viele Arbeiten zu Refactorings beeinflusst. Die Einteilung in eine Vorbedingung und eine Transformation wird in den meisten Refactorings-Beschreibungssprachen genutzt. Im Vergleich des Ansatz von Re$\mathcal{L}$ mit dem von Roberts fällt auf, dass die Trennung von Transformation und Vorbedingung in Re$\mathcal{L}$ nicht so stark ausgeprägt ist. Dies kommt daher, dass das AfterTemplate sowohl für die Feststellung, ob ein Refactoring ausgeführt werden darf als auch für die Beschreibung der Transformation benötigt wird. Wenn Roberts's Vorbedingung Bestandteile von Re$\mathcal{L}$ gegenüberstellt werden sollen, entspricht sie einer Kombination von BeforeTemplate und Precondition. Die Transformation im Sinne von Roberts entspricht den beiden Templates zusammen mit der Calculation.

5.11.2 Graphtransformationen

Neben der Arbeit von Roberts sind die Arbeiten zu Graphtransformationen [MEDJ05, EJ04] die technisch am stärksten verwandten Arbeiten. Bei Graphtransformationen kann sogar gesagt werden, dass Re$\mathcal{L}$ Ähnlichkeiten mit einer Rückportierung der Graphenwelt auf BNF-basierte Ansätze besitzt. Der Grund, wieso nicht Graphtransformationen direkt genutzt werden, liegt grob gesagt an der speziellen Struktur der BNF-Sprache bzw. den Strukturen und deren Semantik. MOF-basierte Ansätze haben Stärken in netzartigen Sprachbeschreibungen, wie sie in visuellen (graphischen) Sprachen wie UML auftreten. Einen Syntaxtree, welches die gängige Graphbeschreibung für BNF-basierte Sprachen ist, wird dabei als allgemeiner Graph behandelt, ohne das die spezielle Struktur ausgenutzt wird. Das zeigt sich an verschiedenen Stellen. In der Folge ergeben sich daraus die in Abschnitt 5.2 genannten Probleme im Einsatz von Graphtransformationen im Umfeld von textbasierten Sprachen.

Um die Zusammenhänge zu verstehen, ist es hilfreich, sich die Parallelität in der Syntaxbeschreibungsstruktur zwischen textbasierten und visuellen Sprachen anzusehen: Beide Arten von Sprachen haben ein innere Syntaxstruktur. Bei textbasierten Sprachen ist dies eine baumartige Struktur, bei denen sich verschiedene Knotenarten wiederholen dürfen. In visuellen Sprachen ist es meist ein Netz von oft gleichrangigen Verbindungen zwischen den Knoten. Für beide Arten gibt es eine Beschreibung wie diese Struktur aussieht (vergleiche Tab. 5.7). Die jeweiligen Beschreibungssysteme bauen dabei auf die Natur der Sprache auf. Code ist immer eine Aneinanderreihung, also ein linearer Strom von Informationen.

Beschreibung der Syntaxbeschreibung	BNF der BNF	Meta-Meta-Modell
Syntax-Beschreibung	BNF	Meta-Modell
Objekt	Code	Diagramm

Tabelle 5.7: Vergleich der MOF Struktur mit dem BNF-Ansatz

Deshalb ist eine Beschreibung als Abfolge von Sytanxelementen wie sie mit einer BNF beschrieben wird, eine einfache, verständliche und angemessene Beschreibung. Dabei ist die benutzte BNF wieder eine Sprache, die auch eine textbasierte Sprache ist. Somit bietet es sich an, eine BNF wieder als BNF zu beschreiben. Diese Beschreibung verwendet nicht alle möglichen Konstrukte aus der ISO [Int96] (simple BNF).

Diese Struktur hat bemerkenswerte Ähnlichkeit mit der MOF-Struktur der OMG. Ein Diagramm wird mittels eines Meta-Modells beschrieben, welches die Syntax des Diagramms beschreibt. Auch ein Meta-Modell ist wieder ein Diagramm, welches selber wieder mittels eines Meta-Modells beschrieben werden kann. Das Meta-Meta-Modell ist dabei so gehalten, dass es durch sich selber beschrieben werden kann.

Die beschriebenen Ebenen sind auch in anderen Bereichen ähnlich, so wird z.B. auch bei der Definition einer Semantik bei beiden Formalismen auf der Ebene der Syntaxdefinition definiert, worauf nicht weiter eingegangen werden soll.

Es besteht hier eine Art Dualität zwischen den verschiedenen Sprach-Welten. Diese wirft die Frage auf, ob es zu Re$\mathcal{L}$ in der Welt der visuellen Sprache einen dualen Mechanismus gibt. Wie oben gesagt ist die positive Antwort darauf die Graphtransformationen. In der folgenden Tabelle sind die verwandten Konstrukte der beiden Formalismen gegenübergestellt.

Re$\mathcal{L}$	Graphtransformationen
BeforeTemplate	LHS
AfterTemplate	RHS
PreCondition	Nicht Anwendbarkeits Bedingung
Calculation	Small Step Rules

Die *Left Hand Side* (LHS) beschreibt bei Graphtransformationen wie der (Teil-)Graph vor einer Transformation aussieht. Wenn diese matcht und eine evtl. vorhandene *Non Application Condition* nicht erfüllt ist, dann kann die Regel ausgeführt werden. Das entsprechende Konstrukt in Re$\mathcal{L}$ ist das BeforeTemplate in Zusammenhang mit der Precondition. Im Vergleich zu den Graphtransformationen ist das BeforeTemplate, bedingt durch Eigenschaften einer BNF, etwas weniger mächtig, welches durch die stärkere Mächtigkeit der Precondition aufgefangen wird. Die Precondition ist demnach nicht nur auf die Nicht-Anwendbarkeit eingeschränkt.

Die *Right Hand Side* beschreibt wie der Graph nach der Transformation aussehen soll. Die entsprechende Rolle übernimmt bei Re$\mathcal{L}$ das AfterTemplate.

Es gibt auch prinzipielle Unterschiede. Graphtransformationsbeschreibungen sind universell gestaltet. Das heißt, sobald ein Graph eines Modells gegeben ist, können darauf die Regeln angewendet werden, ohne die Sprache an sich zu verändern. Die Voraussetzung der Graphen führt dazu, dass das Modell oder der Code als Graph dargestellt werden muss.

Eine interessante Frage ist, wieso Graphtransformationen im Vergleich von Abschnitt 5.2 so schlecht abschneiden und warum nicht die Refactorings gleich mit Graphtransformatio-

nen beschrieben wurden. Die Antwort auf beide Fragen ergibt sich aus den Unterschieden zwischen BNF und Meta-Modell. Beide Syntaxbeschreibungen bauen auf anderen Paradigmen auf. Eine BNF-basierte Sprache kann immer als Baum dargestellt werden, wogegen die visuellen Sprachen durch ein Netz beschrieben werden. Sowohl ReL als auch Graphtransformationen stellen speziell auf diese Strukturen abgestimmte Umsetzungen der Transformationen dar.

5.11.3 ReL als Domain Specific Language

Domain Specific Languages (DSL) sind Sprachen, die speziell für ein bestimmtes Problemfeld entworfen und implementiert werden. Dabei wird versucht, die Sprache des Problemfeldes aufzugreifen, so dass die Sprache schnell mit Hilfe des vorhandenen Hintergrundwissens der Domäne zu erfassen und zu verstehen ist [RGTL09]. Die Forschung über DSL hat verschiedene Richtungen, wie z.B. dem Entwickler/User eine generische Tool-Unterstützung bereit zu stellen ([msd, xte]). Für ReL ist der Forschungsbereich zu angepassten DSL interessant [MHS05]. Ziel ist es, nicht die ganze Sprache immer vom Anfang an neu zu entwickeln, sondern einen bestimmten festen Kern als Ausgangspunkt zu nehmen und dann diesen mit domainenspezifischen Bereichen anzureichern. Ein Beispiel für dieses Vorgehen ist zum Beispiel eine DSL für Workflows. Im Bereich von Workflows gibt es auf der einen Seite generelle Begriffe, die in allen Unternehmen gleich verwendet werden. Auf der anderen Seite gibt es auch spezielle Definitionen oder Prozesse, die es in anderen Firmen nicht gibt. Zusätzlich werden für allgemeine Sachverhalte dabei auch andere Begriffe verwendet. Eine DSL für diesen Zweck kann daher aus einem Kern (Core) bestehen, welcher die allgemeinen Konzepte bereitstellt. Zusätzlich gibt es definierte Erweiterungspunkte. An diesen Erweiterungspunkten werden die unternehmensspezifischen Definitionen eingefügt. Somit ergibt sich eine Klasse von Workflowbeschreibungssprachen, die einen gemeinsamen Kern haben, aber auf die einzelnen Bedürfnisse der Anwender zugeschnitten sind.

ReL nutzt die beschriebene Technik, um einen allgemeinen Sprachrahmen zu schaffen, in dem die Refactorings mit Hilfe der Zielsprache beschrieben werden. Die Zielsprache ist im Sinne von DSLs die Domäne bei ReL.

Eine Schwierigkeit von ReL ist die Abhängigkeit von der BNF der Zielsprache. Je nach Form der BNF, sind manche Refactorings sehr einfach anzugeben.

5.12 Technische Umsetzung

Die Refactoringsprache ReL ist prototypisch implementiert worden. Die Implementierung setzt sich aus mehreren Teilen zusammen. ReL-Gen erzeugt aus einer gegeben BNF eine ReL-Instanz, welche als Grundlage für das eigentliche ReL-Tool verwendet wird. Das ReL-Tool setzt sich neben der ReL-Instanz aus einem Framework und der Implementierung der Analysefunktionen zusammen.

Zusätzlich wird in diesem Kapitel das RMC-Tool betrachtet. Das RMC-Tool baut auf einem Vorläufer von ReL auf. Beim RMC-Tool sind weitergehende Konzepte zur Einbindung von Refactorings betrachtet worden. Einige dieser Ergebnisse sind direkt auf ReL übertragbar, da die Benutzungsschnittstelle sich zwischen ReL und seinem Vorläufer nicht verändert hat. Zu diesem Aspekt gehört insbesondere die Benutzerschnittstelle. ReL besitzt selber keine Interaktionen mit einem Anwender eines Refactorings. Dies ist gut für die Beweistechniken, macht aber einen zusätzlichen Schritt in der IDE-Einbindung notwendig.

Ansatz	Verständ-lichkeit	Genauig-keit	Universalität	Automatisier-barkeit	Sprach–unabhängigkeit	Syntaktische Korrektheit	Typ-Korrektheit/ Wohlgeformtheit	Beweisbarkeit
Umganssprachlich	++	-	+	– –	–	○	++	– –
Vor/Nachbed.	○	++	++	○	○	++	○	+
OCL-Script	–	+	+	+	+	+	+	– –
QVT	–	+	+	+	+	+	+	– –
Graphtranfoma.	–	+	+	+	+	+	+	– –
JunGL	–	+	+	+	+	+	+	– –
Eclipse	– –	+	+	+ +	– –	+	– –	– –
Re$\mathcal{L}$	+	++	+	+ +	+	+	++	++

Tabelle 5.8: Vergleich von Re$\mathcal{L}$ mit den existierenden Refactoringssprachen aus Tabelle 5.5

Ansatz	Verständ-lichkeit	Genauig-keit	Universalität	Automatisier-barkeit	Sprach–unabhängigkeit	Syntaktische Korrektheit	Typ-Korrektheit/ Wohlgeformtheit	Beweisbarkeit
Kommunikation	++	○	○	–	○	○	○	–
Analyse	○	++	○	–	++	++	++	+
Implementierung	○	++	○	++	++	++	++	–

Tabelle 5.9: Für den Einsatzzweck benötigte Anforderungen (Wiederholung von Tabelle 5.6)

Diese GUI-Einbindung ist bei RMC mittels einer GUI-Beschreibung realisiert worden.

5.12.1 Re$\mathcal{L}$-Gen

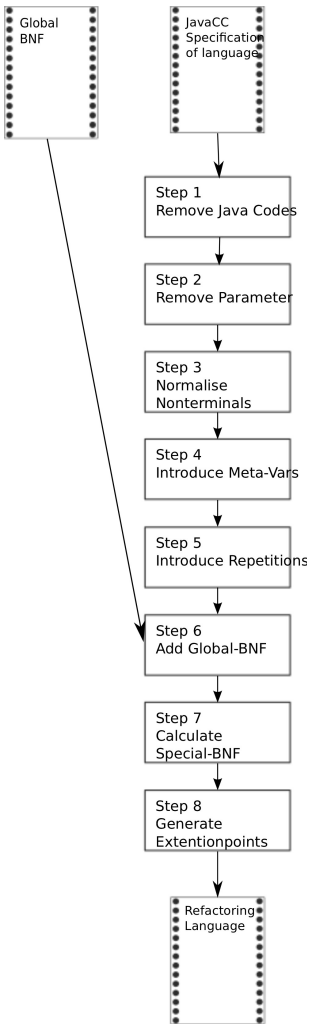


Abbildung 5.13: Übersicht über die Erstellung der Sprache

Der Refactoring-Sprachen-Generator Re$\mathcal{L}$-Gen erstellt für eine BNF einer Sprache, welche als JavaCC-Grammatik gegeben ist, eine (nach den Vorgaben aus Abschnitt 5.5) erstellte Refactoringbeschreibungssprache. Dieser Generator ist ebenfalls auf Grundlage von JavaCC entwickelt worden. Im Gegensatz zu den theoretischen Arbeiten sind hier einige zusätzliche Schritte nötig, da toolgeeignete Grammatiken häufig zusätzliche Bestandteile beinhalten.

Parser-Generatoren wie JavaCC [Cop07] oder Lex [JRL92] nutzen im Allgemeinen keine reine BNF. Um Events, bzw. die Erzeugung von Strukturen zu unterstützen, werden die Grammatiken um semantische Funktionalität erweitert. In JavaCC sind in der Produktion Java-Blöcke erlaubt, die in Abhängigkeit von den Produktionen, bzw. Teilproduktionen, ausgeführt werden. Dazu können sie auf die Tokenstruktur oder auf in Variablen abgelegte Token oder Werte zugreifen. Diese Zusatzinformationen müssen vor den eigentlichen Transformationen entfernt werden. Dies geschieht in den Stufen 1 und 2 der Sprachgenerierung.

Die zweite Änderung in der Erzeugung ist durch die Vermischung von Terminalen und Nichtterminalen in den Token-Definitionen erzwungen. Einige Token, die keinen festen Wert besitzen (z.B. Strings, Zahlen-Konstanten oder Identifierer), werden oft mit Hilfe von regulären Ausdrücken zu Token gepasst. Diese sind für Re$\mathcal{L}$ in Nichtterminale umzuwandeln. Hierzu müssen solche Nichtterminale erkannt werden und für die Einführung der Meta-Variablen vorbereitet werden. Dies wird in Step 3 durchgeführt.

Der Step 4 entspricht der in Abschnitt 5.5 beschriebenen Einführung der Meta-Variablen und stellt den ersten echten Transformationsschritt im Sinne der Umwandlung dar. Die Einführung der Meta-Variablen führt in der technischen Umsetzung zu einem Problem mit speziellen Eigenschaften der Tools. Eine toolgeeignete Grammatik muss eindeutige Entscheidungen an jeder Verzweigung ermöglichen. Wenn die Meta-Variablen, wie in Abschnitt 5.5 beschrieben, eingeführt werden, sind diese Bedingungen regelmäßig verletzt. Aus diesem Grund müssen dem Parser Hilfen gegeben werden. Im Fall von Re$\mathcal{L}$ sind dies Lookahead Informationen. Der klassische Lookahead gibt an, wie viele Token im Voraus betrachtet werden müssen, um eine Entscheidung zu treffen. JavaCC gibt zusätzlich die Möglichkeit lokale und semantische Lookaheads zu verwenden. Der lokale Lookahead erlaubt es, die zu betrachtenden Token für eine Verzweigung separat zu setzen. Dies wiederum erlaubt es, den Parser für die meisten Produktionen mit einem effizienten Lookahead von 1 zu generieren und bei Produktionen, wo dies nicht ausreicht einen höheren Lookahead zu verwenden. Der semantische Lookahead wird verwendet, um effizient die zu einer Produktion passende Meta-Variablen-Produktion aufzubauen. Sie erlaubt es, aufgrund einer

Java-Funktion die richtige Alternative zu wählen. In ReL werden dazu die Typen der Meta-Variablen ausgewertet.

Eine nützliche Besonderheit von JavaCC ist, dass zusätzliche Knoten für den AST explizit in der BNF spezifiziert werden können. Damit können AST-Knoten nicht nur mit einer Produktion verbunden, sondern flexibel eingesetzt werden. Dies ist nützlich, bringt aber keine neuen Möglichkeiten, da dies umständlicher durch eine Aufteilung von Produktionen in mehrere Produktionen erreicht werden kann.

Step 5 bis Step 7 entsprechen den in Kapitel 5.7 beschrieben Transformationen: Einbau der Wiederholungen (Step 5), Hinzufügen der Core-BNF (Step 6) und die Berechnung von `NTTYPE` und `NTSCOPETYPE` (Step 7).

Als Letztes müssen einige semantische Erweiterungen in der Sprache vorgenommen werden, damit diese zusammen mit dem Framework zu dem Refactoring-Tool ReL-Tool kombiniert werden können (Step 8).

5.12.2 ReL-Core: Ein Framework für Refactorings von verschiedenen Sprachen

Mit dem Tool ReL-Gen wird aus einer BNF für eine gegebene Sprache eine Instantiierung von ReL generiert. Da die dadurch gewonnene BNF ohne Funktion ist, das heißt keine Informationen über die Abarbeitung der Refactorings hinterlegt ist, wurde eine Framework entwickelt [Zie10], mit der aus einer ReL-BNF ein funktionierendes Refactoring-Tool generiert werden kann.

ReL-Core beinhaltet alle algorithmischen Bestandteile von ReL. Hierzu gehören unter anderem:

- Einlesen und Parsen einer Datei
- Matchen des BeforeTemplates
- Auswertung der Vorbedingung
- Durchführen der Calculation
- Zusammensetzen des refaktorisierten AST
- Ausgabe des AST in eine Datei
- Einbindung und Aufruf von Analysefunktionen

Die einzelnen Umsetzungen sind in [Zie10] beschrieben. Hier soll nur auf einige wenige interessante Details eingegangen werden.

Die schwierigste Aufgabe des ReL-Cores ist das Matchen der Templates. Dies betrifft insbesondere die effektive Berechnung der Belegungen von Wiederholungen. Dies wird an dem Beispiel in Abbildung 5.14 sichtbar. In dem Ausschnitt eines AST ist innerhalb der Produktion *NTa* viermal die Produktion *NTb* zu finden. Die Ziffern sind zur Referenzierung angegeben. Auf diesen AST soll nun ein BeforeTemplate angewandt werden, in dem die Produktionen *NTb* auf drei Variablen `MAR1`, `MAR2` und `MAR3` aufgeteilt werden:

$$\#\texttt{I1}(\texttt{MAR1}) * \texttt{MAR2}\#\texttt{I2}(\texttt{MAR3})*$$

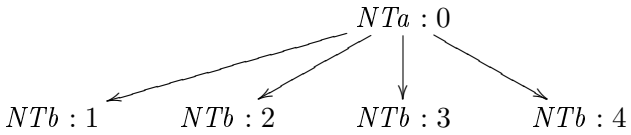


Abbildung 5.14: Beispiel für die Aufteilung von Produktionen auf Meta-Variablen

In dem Template wird gefordert, dass die Produktion *NTb* mindestens einmal auftritt. Wenn sie mehrfach auftritt, kann für `MAR2` eine beliebige gewählt werden. `MAR1` enthält alle Auftreten von *NTb* vor dem in `MAR2`; `MAR3` enthält entsprechend das Auftreten danach.

Wenn die Nummern als Referenz genommen werden, sind für Abbildung 5.14 folgende Belegungen möglich:

MAR1	MAR2	MAR3
$\{\}$	1	$\{2,3,4\}$
$\{1\}$	2	$\{3,4\}$
$\{1,2\}$	3	$\{4\}$
$\{1,2,3\}$	4	$\{\}$

Aus den Belegungen muss eine ausgewählt werden, die die Vorbedingung des Refactorings erfüllt. In [Zie10] wird dafür ein Algorithmus beschrieben, der effizient die passenden Möglichkeiten der Wiederholungen berechnet.

Eine weitere wichtige Fragestellung ist auch die der Formatierung des Codes. Ein Anwender eines Refactorings erwartet, dass zum einen der von ihm formatierte Code nicht umformatiert wird, und dass der refaktorisierte Code eine sinnvolle Formatierung erhält. Die Erhaltung der alten Formatierung ist mittels JavaCC leicht umzusetzen. Zu jedem Token werden die Whitespaces gespeichert, die den Token vom letzten Token trennen. Wenn in der Ausgabe die Whitespaces mit ausgegeben werden, bleibt die Formatierung erhalten. Somit reduziert sich die Fragestellung auf den Code, der durch das Refactoring geändert wurde.

Das Re$\mathcal{L}$-Tool wird aus der generierten Re$\mathcal{L}$-Instanz, dem Re$\mathcal{L}$-Core und den separat implementierten Analysefunktionen zusammengesetzt. Eine Einbindung in eine IDE ist mit dem Prototypen nicht durchgeführt worden, da dazu Erkenntnisse aus einer vorherigen Implementierung mit einem Vorläufer von Re$\mathcal{L}$ vorliegen. Diese wurde von der Projektgruppe RMC (Refaso Model Cockpit,[KDO+08]) durchgeführt und wird im nächsten Unterkapitel behandelt.

5.12.3 Eine Fallstudie: Re$\mathcal{L}_{Java}$

Mit den Tools aus den beiden vorherigen Abschnitten wurde eine Fallstudie mit Java durchgeführt [Zie10]. In dieser wurde gezeigt, dass die technische Umsetzung von Re$\mathcal{L}$ auch auf eine Standard-Programmiersprache anwendbar ist. Dazu wurden mehrere Refactorings in Re$\mathcal{L}$ spezifiziert und an Beispielen durchgeführt.

Die Fallstudie in Java hat mehrere Ergebnisse gebracht, die zu Verbesserungen von Re$\mathcal{L}$ geführt haben. Die wichtigste ist die Einführung von `noReplace`-Locations, die es

erlauben als Scope eines Subtemplates eine `noReplace`-Location zu verwenden. Dadurch sind Refactorings in Java einfacher zu spezifizieren. Die Lesbarkeit leidet nicht und es müssen weniger Fälle durch die Polymorphie des Repositories abgedeckt werden.

Da in den Fallstudie die Anwendbarkeit des Tools und die Mächtigkeit der Sprache Re$\mathcal{L}$ im Vordergrund stehen, sind keine formalen Beweise der Refactorings durchgeführt worden.

5.13 Refaso Model Cockpit - RMC

Das RMC-Projekt [KDO+08] ist eine prototypische Umsetzung eines Modellierungstools, welches einen Qualitätszyklus umsetzt. Bei RMC ist eine Vorversion von Re$\mathcal{L}$ genutzt worden. Die Struktur von Re$\mathcal{L}$ baut auf den Erfahrungen von RMC auf. In RMC wurden einige Themen behandelt, die bei der Re$\mathcal{L}$ Implementierung aussen vorgelassen wurden, da sie zufriedenstellend in RMC getestet wurden. Dies umfasst z.B. die Interaktion mit dem Nutzer und die Bestimmung von Anwendungsempfehlungen. Zuerst werden die Komponenten des RMC-Tools beschreiben. Im Anschluss wird genauer auf Re$\mathcal{L}$ bezogene Themen eingegangen.

Der Qualitätzyklus von RMC setzt sich aus der Bestimmung der Qualität, der Ableitung von Mängeln und deren Korrektur zusammen. RMC konzentriert sich auf Designmängel, also strukturelle Probleme. Es sind somit keine Mängel betrachtet worden, die Syntax, Wohlgeformtheit, Konsistenz o.ä. von Modellen betreffen. Bevor in den nächsten Abschnitten die Refactoringimplementierung, die Benutzerführung und die Lehren für Re$\mathcal{L}$ betrachtet werden, sollen die Bestandteile des RMC-Projekts vorgestellt werden.

Das UML/Z-Model

Das Modell auf dem RMC arbeitet ist ein UML/Z-Modell. UML/Z ist eine Zusammensetzung von Elementen der UML mit Elementen von Z. Das Ziel ist ein Modell zur Verfügung zu haben, welches eine graphische Notation nutzt, aber äquivalent in CSP-OZ abgebildet werden kann. CSP-OZ ist als semantische Domäne von UML/Z genutzt worden.

Messen von Qualität

In der Einleitung dieser Dissertation wurde die Bedeutung von guter Qualität von Modellen hervorgehoben. Im Umfeld von Refactorings ist besonders interessant, wie gut Modelle verständlich und wartbar sind. Die Frage ist, wie solche Faktoren bestimmt und beschrieben werden sollen. Im Rahmen des RMC-Projektes wurde diese Frage im Bereich der Qualitätsmodelle und Qualitätsmessung betrachtet, der zu der Dissertation von Hendrik Voigt [Voi09] führt. Es wird hier ein kleiner Überblick über diesen Bereich gegeben. Eine vollständige Darstellung findet sich in [Voi09]. Das Qualitätsmodell beschreibt, welche Informationen benötigt werden und wie diese bestimmt werden können. Die Messungen sind die Durchführung der Qualitätsbestimmung. Die Ergebnisse der Messungen können in Indikatoren zusammengefasst und bewertet werden.

Diagnose

Die Ergebnisse der Messungen und der Auswertung mittels des Qualitätsmodelles, zeigen auf, wo es Qualitätsprobleme in dem Modell gibt. Es stellt sich die Frage, wie diese zu

beheben sind. Im RMC-Tool hilft die Diagnose-Komponente dem Modellierer Optionen zu finden. Aus den Ergebnissen der Messung werden in einem zweistufigen Prozess erst Gründe für die beobachteten Mängel gesucht und darauf aufbauend Vorschläge zur Behandlung gemacht.

Dieses Vorgehen ist analog zum Vorgehen eines Arztes. Ein Patient kommt mit einer Menge von Symptomen, z.B. Husten und Fieber zum Arzt. Dieser analysiert die Symptome und stellt daraufhin eine Diagnose, welche die Symptome erklären kann. Für Husten und Fieber kann dies eine Erkältung sein. Für eine Erkältung stehen einige Standardtherapiemethoden zur Verfügung. Aus diesen wählt der Arzt Bettruhe und ein Hustenmittel aus. Somit hat der Arzt aus einer Menge von Therapiemöglichkeiten eine ausgewählt, die zur Diagnose passt.

Ähnlich ist die Diagnose im RMC aufgebaut. Denn Symptome entsprechen den Auswertungen der Indikatoren aus dem Messmodell. Wenn ein Indikator einen unguten Zustand anzeigt, ist dies ein Symptom. Zusätzlich zu den Indikatoren können normalisierte Messungen direkt verwendet werden. Die Diagnosen werden durch das Diagnosemodell beschrieben und sind in der Konfiguration vorgegeben. Diese werden mit den Therapiemöglichkeiten verknüpft. RMC gibt die Möglichkeit, Refactorings als Therapien zu verwenden. Therapien, die nicht durch ein Refactoring beschrieben werden können, können als textuelle Beschreibung, den Texttherapien, angegeben werden.

Das zentrale Element in der Modellierung der Diagnosestellung ist die Diagnose. Die Symptome und die Therapien werden in Abhängigkeit von den Diagnosen dargestellt. Eine Diagnose „unverständliche Klassennamen“ kann sich durch verschiedene Symptome ausdrücken, etwa kurze Klassennamen („x“, „In“) oder mehrere Klassen mit sehr ähnlichen Namen („ClassA“, „ClassB“).

Auch die Therapien werden einer Diagnose zugeordnet. Für die „unverständlichen Klassennamen“ kann dies das Refactoring „Rename“ sein, um einen passenden Klassennamen zu setzen. Abbildung 5.15 zeigt einen Auszug aus einer komplexen Modellierung. Es sind einige der Symptome mit mehreren Diagnosen verbunden. Auch können Therapien in mehreren Diagnosefällen Abhilfe schaffen.

Nachdem die Modellierung des Diagnose-Problems klar ist, ist die Frage, wie eine passende Diagnose gefunden und eine gute Therapie auswählt wird.

Modellverbesserung

Die Therapien dienen in RMC der Modellverbesserung. Da es sich in RMC immer um gültige Modelle handelt, bieten sich Refactorings an. Alle Verbesserungen, die sich nicht als Refactoring beschreiben lassen, liegen als textuelle Beschreibungen vor, die per Hand ausgeführt werden können.

Die Refactoringkomponente Die Beschreibung der Refactorings in RMC ähnelt in vielen Bereichen Re$\mathcal{L}$. So existieren schon die groben Bestandteile (Templates, Vorbedingung und Calculation) und die Meta-Variablen.

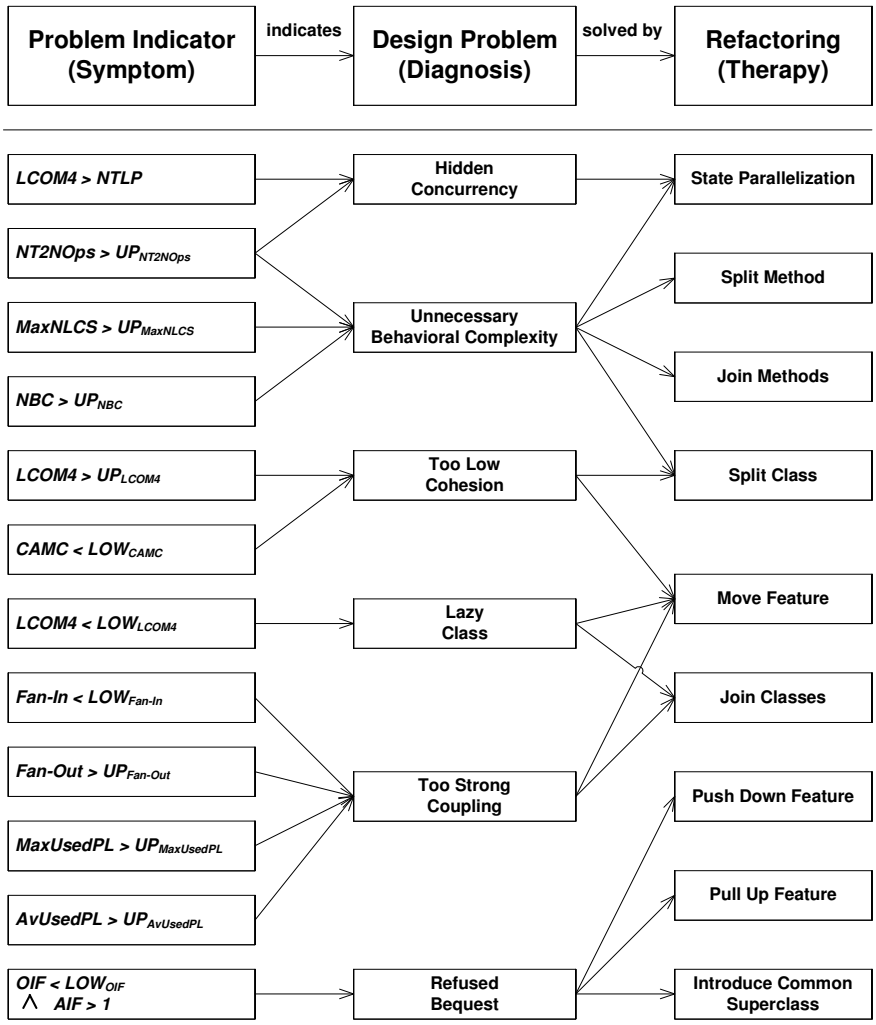


Abbildung 5.15: Beispiel einer Diagnosemodellierung nach [VR08]

Eine GUI für Refactorings In Re$\mathcal{L}$ wird ein Refactoring parametrisiert. In der Regel möchte ein Nutzer eines Refactorings die Parameter interaktiv eingeben, bzw. aus dem Kontext soviel wie möglich ableiten. In RMC ist die Beschreibung der Refactoring-GUI von der Refactoringbeschreibung getrennt worden. Das hat mehrere Vorteile. Zum Einen hängen die Dialoge von den verschiedenen Kontexten ab, in denen das Refactoring aufgerufen wird. Auf der anderen Seite ermöglicht die Trennung, dass die Refactoringdefinition frei von Zusatzinformationen ist. Dies vereinfacht die Beweise und hält die Beschreibung lesbar.

Eine Refactoring-GUI-Beschreibung gliedert sich in drei Bereiche. Der erste Bereich, die Kontextbeschreibung, stellt fest, ob die Beschreibung in dem Bearbeitungskontext eine sinnvolle Option für den Nutzer bereitstellt. Der zweite Bereich ist die Dialogbeschreibung, welche die Interaktion mit den Nutzer definiert. Der dritte Bereich beschreibt den Aufruf des Refactorings. Dabei können für jedes Refactoring mehrere GUI-Beschreibungen existieren, die in verschiedenen Bearbeitungskontexten genutzt werden.

Kontextbeschreibung Die Kontextbeschreibung stellt anhand der selektierten Elemente der GUI fest, ob dieses Refactoring sinnvoll ist. Diese Information wird genutzt, um die GUI anzupassen („ausgrauen„ von nicht sinnvollen Refactorings). Für die Bearbeitung eines Interfaces ist es nicht sinnvoll das Refactoring `ExtractMethod` zu nutzen, also sollte es auch nicht in einem Menü angeboten werden.

Dialogbeschreibung Wenn ein Refactoring ausgelöst wird, müssen meist einige Informationen vom Benutzer angefragt werden. Dazu wird zuerst versucht, aus dem Kontext möglichst viele Informationen abzuleiten. Wenn ein Variable markiert ist und `RenameVariable` durchgeführt werden soll, kann der Name der Variablen aus der Markierung abgeleitet werden. Es fehlt in diesem Beispiel die Information, wie die Variable in Zukunft heißen soll. Dazu stehen einige vorbereitete Dialoge zu Verfügung. Es bietet sich ein Dialog an, der einen Text und ein Eingabefeld besitzt. In RMC stehen neun verschiedene Dialoge zu Verfügung, die flexible parametrierbar sind. Jede Eingabe kann in der Dialogbeschreibung auf Sinnhaftigkeit geprüft werden. Wenn eine Eingabe unzulässig ist, wird dem Benutzer die Möglichkeit zur Korrektur gegeben.

Refactoringaufruf Nachdem alle benötigten Daten gesammelt wurden, wird das eigentliche Refactoring aufgerufen. Das Refactoring wird anhand der Signatur (Name und Parameter) aus dem Repository ausgewählt. Wenn mehrere Refactorings mit der gleichen Signatur zur Verfügung stehen, wird in einer gegebenen Reihenfolge geprüft, welche Definition die Vorbedingung erfüllt. Diese wird dann gewählt. Dabei ist Durchführung und Dialogsteuerung streng getrennt.

Die Schnittstelle für den Aufruf hat sich zwischen RMC und Re$\mathcal{L}$ nicht verändert, wodurch sich die Refactoring-GUI-Beschreibung auch mit Re$\mathcal{L}$ nutzen lässt. Deshalb wurde dieses System in den Nachfolge-Prototypen nicht weiterentwickelt.

Kapitel 6

Erstellung von Refactorings

In den letzten Kapiteln wurden Refactorings eingeführt und es ist auch erklärt worden, wie Refactorings beschrieben werden. Dieses Kapitel soll sich mit der Frage der Erstellung von Refactorings beschäftigen. Ähnlich wie die Benutzung einer Modellierungssprache, wie etwa UML, nicht automatisch zu guten Modellen führt, führt Re$\mathcal{L}$ nicht zwangsläufig zu gut beschriebenen Refactorings. Es sollen daher einige Probleme der Refactoringmodellierung und deren Lösungen beschrieben werden.

Bei der Modellierung von Refactorings mit Re$\mathcal{L}$ kann die Fragestellung in zwei Bereiche geteilt werden: Die *konkrete Modellierung*, d.h. die Erstellung eines Refactorings, und die Frage nach dem *Modellierungsdesign*. Die zweite Frage beinhaltet das Problem, wie die Aufgaben zwischen den Analysefunktionen und den Prädikaten in der Precondition bzw. der Calculation verteilt werden. Es soll analysiert werden, wie Analysefunktionen zur Modellierung eines Refactorings eingesetzt werden sollen. Eine zweite Fragestellung zum Modellierungsdesign ist die Aufteilung von Refactorings in einfache und zusammengesetzte Refactorings. Bevor diese Felder betrachtet werden, soll zuerst die konkrete Modellierung eines Refactorings an einem Beispiel beschrieben werden.

Das Thema dieses Kapitels ist, die angeschnittenen Probleme und Lösungen genauer zu betrachten. Dazu wird ab jetzt das Refactoring einer komplexeren Sprache betrachtet. Es werden Refactorings für die formale Sprache CSP-OZ beschrieben und die Refactorings werden entsprechend in der Re$\mathcal{L}$ Instantiierung Re$\mathcal{L}_{CSP-OZ}$ spezifiziert.

6.1 Die Modellierung eines Refactorings

Dieser Abschnitt beschäftigt sich mit dem Prozess der Erstellung eines Refactorings. Anhand eines einfachen Refactorings soll gezeigt werden, wie mit Re$\mathcal{L}$ ein Refactoring erstellt wird. Als Beispiel soll das Refactoring `SplitOperationOZ` dienen. Das OZ im Namen deutet an, dass dieses Refactoring auf einer Object-Z-Klasse einer CSP-OZ Spezifikation arbeitet. `SplitOperationOZ` zerlegt das Schema einer Operation in zwei Schemata, welche als private Operationen der Klasse hinzugefügt werden. Die alte Operation wird dann aus den beiden Teiloperationen zusammen gebaut.

Der Ablauf der Modellierung eines Refactoring wie `SplitOperationOZ`gliedert sich in mehrere Schritte:

Vorläufige Templates erstellen Hierbei werden die für das Refactoring wichtigen Teile beschrieben. Dabei wird sich auf die wesentlichen Elemente konzentriert, die das Re-

factoring ausmachen. Ziel ist es, die Idee des Refactorings als unvollständiges Templates zu beschreiben, ohne sich in Details der Zielsprache bzw. von ReL zu verstricken.

Variablen Definitionen ergänzen Der erste Schritt der Vervollständigung des Refactorings ist die Ergänzung der Meta-Variablen-Deklarationen und Berechnungen. Dabei wird festgelegt, welche Meta-Variablen Parameter sind und welche durch das BeforeTemplate gematcht werden.

Vorbedingungen erstellen Wenn die Variablen festgelegt wurden, kann die Precondition des Refactorings erstellt werden. Dabei wird zuerst auf die Durchführbarkeit Wert gelegt. Ergänzungen für die Verhaltenserhaltung können in dem letzten Erstellungsschritt hinzugefügt werden.

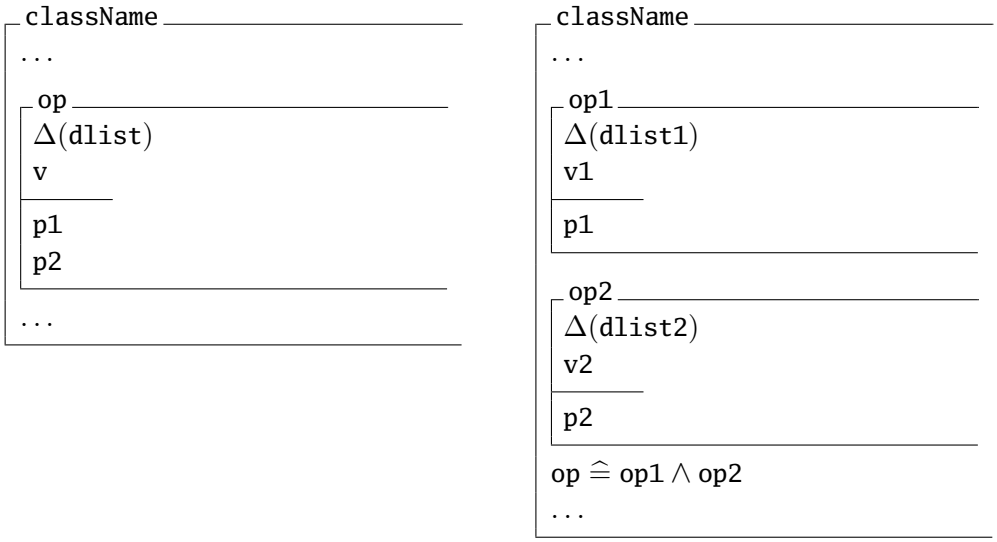
Templates vervollständigen Die Templates besitzen meist Bereiche, die bisher nicht spezifiziert sind („Lücken“) und sich durch das Refactoring nicht ändern. Diese Lücken werden jetzt gefüllt. Dazu müssen neue Meta-Variablen eingeführt werden.

Refactoring anpassen, bis die Refactoring-Eigenschaften erfüllt sind Ein Refactoring in der Entwicklung erfüllt meist nicht alle benötigten Eigenschaften (Syntaxkorrektheit, Soundness, Verhaltenserhaltung). Das Refactoring ist solange anzupassen, bis alle Eigenschaften erfüllt sind. Dabei ist es hilfreich die Eigenschaften immer in der gleichen Reihenfolge (Syntaxkorrektheit, Soundness, Verhaltenserhaltung) zu prüfen.

Dieses Vorgehen soll anhand des Refactorings `SplitOperationOZ` dargestellt werden. Da Object-Z und CSP-OZ halbgraphische Sprachen (Boxen) sind, wird hier eine teilweise kompilierte Version von ReL genutzt. Das heißt, das in der Quelle die originale ReL Spezifikation genutzt wird, in der Darstellung die LaTeX-Befehle aber gerendert werden, so dass eine von Object-Z gewohnte Darstellung erreicht wird.

Vorläufige Templates erstellen `SplitOperationOZ` teilt eine Operation in zwei Teiloperationen und baut aus diesen zwei Teilen die alte Operation wieder auf. Die beiden Teiloperationen werden für äußere Nutzer nicht sichtbar gemacht, da dies eine Erweiterung der Schnittstelle ist, die in diesem Refactoring nicht gewollt ist. Die Modellierung eines solchen Refactorings fängt immer mit den zentralen Elementen von ReL an: den Templates. Als Erstes wird die zentrale Idee des Refactorings in Templates ausgedrückt.

```
op, op1, op2            : OperationName;
className               : Classname;
dlist, dlist1, dlist2   : DeltaList;
v, v1, v2               : Declaration;
p1, p2                  : Predicatelist;
```



Auf der linken Seite befindet sich die Klasse, wie sie vor dem Refactoring aussehen soll. Zur Zeit sind nur die Bestandteile der Klasse betrachtet, die sich zentral ändern sollen. Die Lücken, die in den nächsten Schritten gefüllt werden müssen, sind mit Punkten (...) gekennzeichnet. Die Operation mit Namen `op` soll geteilt werden. Die Operation `op` wird anschließend aus den Operationen `op1` und `op2` wieder zusammengesetzt. Die Prädikate der Operation sind schon durch zwei Meta-Variablen ausgedrückt, wobei eine davon als Parameter genutzt werden soll. Mit diesem ersten Schritt ist schon die grundlegende Idee des Refactorings beschrieben. Schritt für Schritt werden die beiden Template-Rümpfe zu einem vollständigen Refactoring ausgebaut.

Variablen Definitionen ergänzen Dazu sollen die Meta-Variablen in den Refactorings betrachtet werden. In der Beschreibung fällt auf, das im Rumpf des AfterTemplates Variablen genutzt werden, die nicht im Rumpf des BeforeTemplates genutzt werden. Für diese Variablen muss entschieden werden, wie sie ihre Werte erhalten. Dazu gibt es zwei Möglichkeiten: Ein Wert einer Variable kann durch einen Parameter vom Anwender vorgegeben oder berechnet werden. In diesem Beispiel sollen die Operationsnamen `op1` und `op2` vom Benutzer vorgegeben werden und die anderen Variablen berechnet sein. Für die Berechnung muss eine Calculation hinzugefügt werden. Bei der Berechnung der Variablen wird bei unserer Modellierung keine sichtbare Rücksicht auf die Typen der Variablen gelegt. Dies ist wegen der Definition der Analysefunktion (siehe unten) nicht nötig, weil diese eine Menge von Tupeln zurück gibt. In einem Tupel ist neben dem Namen der Variable auch der Typ gegeben. Somit sind die Variablenberechnungen in der Calculation implizit auch typgleich (vgl. Kapitel 6.3 und Anhang B).

$$\mathbf{vars}(\texttt{dlist1}) \cup \mathbf{vars}(\texttt{dlist2}) = \mathbf{vars}(\texttt{dlist})$$
$$\mathbf{vars}(\texttt{p1}) \subset \mathbf{vars}(\texttt{dlist1}) \cup \mathbf{vars}(\texttt{v1})$$
$$\mathbf{vars}(\texttt{p2}) \subset \mathbf{vars}(\texttt{dlist2}) \cup \mathbf{vars}(\texttt{v2})$$
$$\mathbf{vars}(\texttt{v1}) \cup \mathbf{vars}(\texttt{v2}) = \mathbf{vars}(\texttt{v})$$

Als nächstes wird festgelegt, welche Meta-Variablen als Parameter übergeben werden sollen. Da der Benutzer die Operation aus einer Klasse auswählen können soll, wird auf jeden Fall der Klassenname (`className`) und der Name der Operation (`op`) in den Parametern

benötigt. Des Weiteren sollen die Namen der neuen Operationen und die Prädikate, die in die erste Operation ausgelagert werden sollen, angegeben werden. Somit ergibt sich als Parameterliste:

```
op, op1, op2  : OperationName;
className     : Classname;
p1            : Predicatelist;
```

Die anderen Variablendefinitionen werden auf das BeforeTemplate und die Calculation verteilt, wobei die Definition so spät wie möglich erfolgen soll. Das Ergebnis ist in Abbildung 6.1 dargestellt.

Erstellen der Vorbedingungen In diesem Schritt wird eine vorläufige Precondition festgelegt. Da meist noch nicht bekannt ist, welche Vorbedingungen für die Verhaltenserhaltung benötigt werden, empfiehlt es sich, dabei auf die Durchführbarkeit des Refactorings zu achten. In dem Fall von `SplitOperationOZ` fällt keine Vorbedingung auf Anhieb auf. Somit wird eine leere Vorbedingung erstellt, was bedeutet, dass sie immer erfüllt ist.

Vervollständigen der Templates Bis zu diesem Schritt ist die Idee der Refactorings in der Notation von Re$\mathcal{L}$ ausgedrückt worden. Dieses Refactoring ist noch kein gültiges Re$\mathcal{L}$ Refactoring. Dazu müssen die ausgelassenen Bereiche (diese Lücken wurden mit Pünktchen gekennzeichnet) ausgefüllt und die Soundness hergestellt werden. Auch muss sichergestellt werden, dass das Refactoring verhaltenserhaltend ist.
Die Lücken werden anhand der Grammatik von CSP-OZ gefüllt. Dieses Auffüllen soll am Beispiel des BeforeTemplate demonstriert werden. Das AfterTemplate wird entsprechend angepasst. Das Template soll der Produktion `Class0` genügen.

```
Class0 := "\\begin{class}" "\{" ClassName (FormalParameters)? (CSPParameters)? "\}"
          ( VisibilityList | Interface )?
          ( InheritedClass )* ( LocalDefinition )*                                    3
          ( "\\begin{cps}" CSPZ "\\end{csp}" )?
          ( State0 )?
          ( InitialState )?
          ( Operation0 )*
          "\\end{class}"                                                              8
```

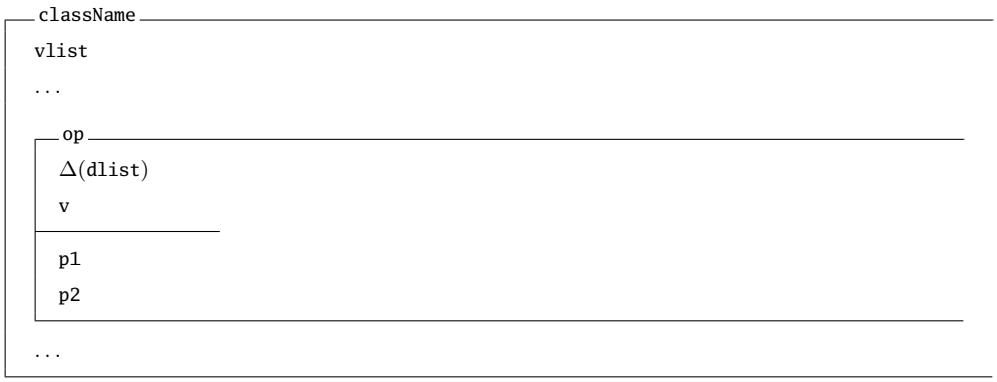
bzw. als gerenderte Version:

Class0 :=

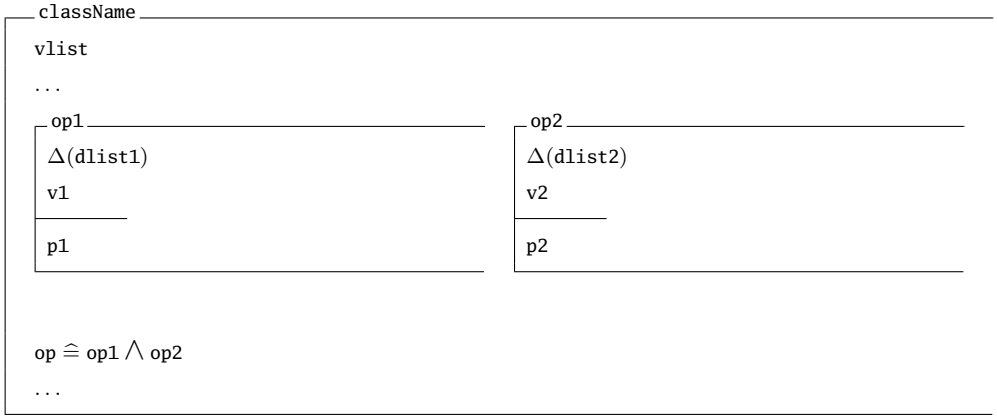
ClassName(*FormalParameters*)?(*CSPParameters*)?
(*VisibilityList* | *Interface*)?
(*InheritedClass*)∗
(*LocalDefinition*)∗
(*CSPZ*)?
(*State*0)?(*InitialState*)?
(*Operation*0)∗

Aus dem Vergleich mit dem Template aus Abbildung 6.1 geht hervor, dass z.B. die Parameter der Klasse ((FormalParameters)? (CSPParameters)?) und andere Operationen bisher nicht berücksichtigt wurden. Für die im Template nicht spezifizierten Teile werden Meta-Variablen deklariert und an die entsprechenden Stellen in das Template eingefügt. Dabei

Refactoring: SplitOperationOZ
Parameter:
op, op1, op2 : OperationName;
className : ClassName;
p1 : PredicateList;
BeforeTemplate:
Define:
v : Declaration;
p2 : PredicateList;
dlist : DeltaList;
vlist : VisibilityList;
Index: Class1 **Scope:** Class0



Precondition:
true
Calculation:
Define:
dlist1, dlist2 : DeltaList;
v1,v2 : Declaration;
SCHEMA:
vars(dlist1) ∪ **vars**(dlist2) = **vars**(dlist)
vars(p1) ⊂ **vars**(dlist1) ∪ **vars**(v1)
vars(p2) ⊂ **vars**(dlist2) ∪ **vars**(v2)
vars(v1) ∪ **vars**(v2) = **vars**(v)
DONE
AfterTemplate:
Index: Class1 **Scope:** Class0



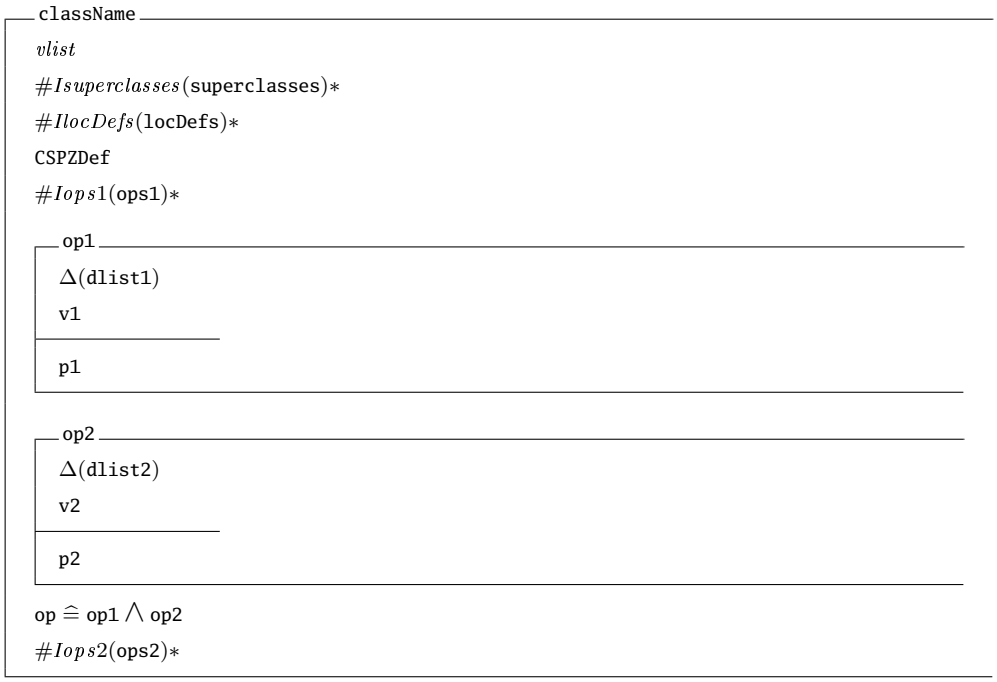
Listing 6.1: Zwischenergebnis der Modellierung eines Refactorings. Die aktiven Teile der Templates sind definiert und die Meta-Variablen deklariert.

muss auf die Art der Klasse geachtet werden. Das Refactoring ist für Object-Z-Klassen vorgesehen, die Produktion `Class0` deckt sowohl CSP-OZ-Klassen als auch Object-Z-Klassen ab. Die Unterscheidung zwischen den Klassen findet durch die Inhalte statt, daher werden hier nur Elemente, die in Object-Z auftreten können, eingesetzt. Das heißt z.B., dass die Produktion Interface nicht genutzt wird, da eine Object-Z-Klasse nur eine Visilbility-Liste hat.

```
BeforeTemplate:
        Define:                                                          2
        v            : Declaration;
        p2           : PredicateList;
        dlist        : DeltaList;
        vlist        : VisibilityList;
        superclasses : InheritedClass;                                   7
        locDefs      : LocalDefinition;
        CSPZDef      : CSPZ;
        classState   : State0;
        init         : InitialState;
        ops1, ops2   : Operation0;                                       12
Index: Class1 Scope: Class0
```

Da die entsprechenden Elemente nach dem Refactoring nicht verändert sein sollen, werden diese genau wieder im AfterTemplate übernommen.

```
AfterTemplate:
Index: Class1   Scope: Class0                                            2
```



Damit ist eine vorläufige Re$\mathcal{L}$-Spezifikation fertig gestellt. Der nächste Schritt ist die Überprüfung der Soundness des Refactorings.

Refactoring anpassen, bis alle wichtigen Eigenschaften erfüllt sind Die erstellte Refactoringbeschreibung muss noch die Syntaxkorrektheit, Soundness und Verhaltenserhaltung erfüllen. Dazu werden die im vorherigen Kapitel bzw. für die Verhaltenserhaltung im nächsten Kapitel beschriebenen Beweistechniken genutzt. Die Prüfung der Eigenschaften liefert häufig nicht auf Anhieb ein positives Ergebnis. Meist lässt sich das anhand eines an das Refactoring angepassten Gegenbeispiels sehen. In dem laufendem Beispiel schlägt der Beweis der Verhaltenserhaltung beim aktuellen Stand fehl. Ein Problem tritt auf, wenn einer der neuen Operationsnamen schon der Name einer existierenden Operation ist. Dieses muss behoben werden. Da die Namen der neuen Operationen als Parameter übergeben werden, kann das in dem Beispiel durch Anpassung der Precondition erreicht werden. Das Refactoring ist nicht durchführbar, wenn für die neuen Operationen Namen angegeben werden, die schon in der Klasse verwendet werden.

Precondition:

$$\forall\, operation : \mathsf{ops1} \cup \mathsf{ops2} \bullet \mathbf{name}(operation) \neq \mathbf{name}(\mathsf{op1}) \land \mathbf{name}(operation) \neq \mathbf{name}(\mathsf{op1})$$

6.1.1 Fertiges Refactoring

Nachdem alle Eigenschaften erfüllt wurden, ist die Erstellung des Refactorings beendet. Es kann einem Repository hinzugefügt werden. Das fertige Refactoring ist in den Abbildungen 6.2 und 6.3 angegeben.

Damit ist die Erstellung eines einfachen Refactoring erklärt. Auf eine detaillierte Erklärung beim Vorgehen für ein zusammengesetztes Refactorings soll verzichtet werden, da sich die Vorgehensweisen ähneln. Als Ausgangspunkt für die Erstellung wird die Kombination (sequentielle Ausführung, Fallunterscheidungen und Schleifen) von Refactorings, statt

Refactoring: SplitOperationOZ
Parameter:
op, op1, op2 : OperationName;
className : ClassName;
p1 : PredicateList;
BeforeTemplate:
Define:
v : Declaration;
p2 : PredicateList;
dlist : DeltaList;
vlist : VisibilityList;
superclasses : InheritedClass;
locDefs : LocalDefinition;
CSPZDef : CSPZ;
classState : State0;
init : InitialState;
ops1, ops2 : Operation0;
genParam : FormalParameters;
Index: Class1 **Scope:** Class0

className(genParam)?

vlist

$\#Isuperclasses$(superclasses)$*$

$\#IlocDefs$(locDefs)$*$

CSPZDef

classState

init

$\#Iops1$(ops1)$*$

op

Δ(dlist)

v

---

p1

p2

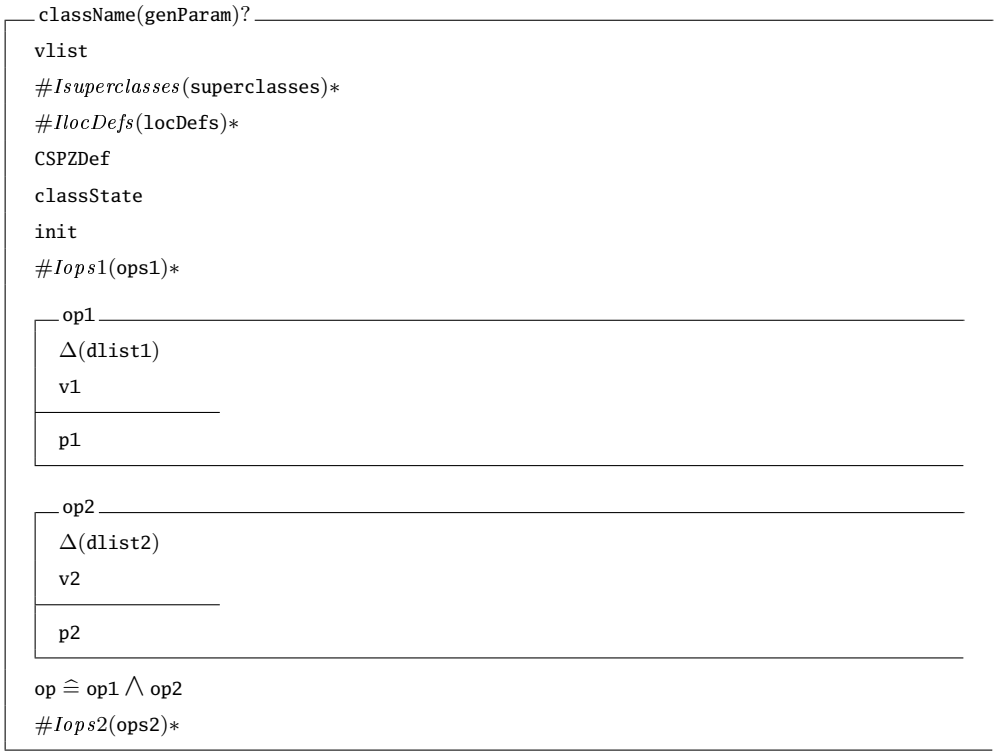
$\#Iops2$(ops2)$*$

Precondition:
$\forall operation : \ \text{ops1} \cup \ \text{ops2} \ \bullet \mathbf{name}(operation) \neq \mathbf{name}(\text{op1}) \wedge \mathbf{name}(operation) \neq \mathbf{name}(\text{op1})$
Calculation:
Define:
dlist1, dlist2 : DeltaList;
v1,v2 : Declaration;
SCHEMA:
$\mathbf{vars}(\text{dlist1}) \cup \mathbf{vars}(\text{dlist2}) = \mathbf{vars}(\text{dlist})$
$\mathbf{vars}(\text{p1}) \subset \mathbf{vars}(\text{dlist1}) \cup \mathbf{vars}(\text{v1})$
$\mathbf{vars}(\text{p2}) \subset \mathbf{vars}(\text{dlist2}) \cup \mathbf{vars}(\text{v2})$
$\mathbf{vars}(\text{v1}) \cup \mathbf{vars}(\text{v2}) = \mathbf{vars}(\text{v})$

DONE

Listing 6.2: Das fertige Refactoring

AfterTemplate: 1
Index: Class1 **Scope:** Class0



Listing 6.3: Fortsetzung: Das fertige Refactoring

eines AfterTemplates, genutzt. Das BeforeTemplate wird wie bei einem einfachen Refactoring erstellt und später ergänzt. Insgesamt ist die Erstellung eines zusammengesetzten Refactorings, wenn eine brauchbare Idee existiert, wie sich das Refactoring zusammensetzt, einfacher als die Erstellung eines einfachen Refactorings.

6.2 Designvorgaben bei der Modellierung von Refactorings

Nachdem die Vorgehensweise bei der Erstellung eines Refactorings vorgestellt wurde, sollen Schwierigkeiten bei der Erstellung von Refactorings betrachtet werden. Die meisten Schwierigkeiten sind allgemeine Designprobleme von Refactorings. Das heißt, dass es dabei um Entscheidungen geht, die die Erstellung aller Refactorings beeinflussen. Dabei sollen nicht die Probleme in den Vordergrund gerückt werden, sondern die Lösungen. In den nächsten Abschnitten werden einige Designregeln angegeben, die aus den dazugehörigen Problemstellungen abgeleitet werden.

6.2.1 Teilrefactorings sind Refactorings

`MoveOperation` ist ein zusammengesetztes Refactoring für CSP-OZ: Dieses Refactoring soll eine Operation aus einer Object-Z-Klasse in eine andere Object-Z-Klasse verschieben. Die meisten Beschreibungen für dieses Refactoring (vgl. [Rob99, Fow04]) teilen die Durchführung in zwei Schritte:

1. Verschieben der Operation: In diesem Schritt wird die Operation aus der alten Klasse entfernt und dann der neuen Klasse hinzugefügt. Die innere Struktur des Programms ist somit zerstört. Das Programm oder Modell ist damit nicht mehr gültig.

2. Alle Verweise auf die Operation werden angepasst. Somit wird in diesem Schritt die innere Struktur wieder hergestellt. Anschließend ist das Refactoring korrekt und verhaltenserhaltend durchgeführt.

Ein Vorteil bei der manuellen Anwendung von Refactorings auf Code ist, dass der Compiler die noch nicht im zweiten Schritt behandelten Stellen als Fehler aufzählt. Die Probleme zeigen sich, wenn versucht wird zu argumentieren, dass das Refactoring verhaltenserhaltend ist. Nach dem ersten Schritt ist die Verhaltenserhaltung zerstört, es ist im Regelfall sogar das Modell bzw. das Programm ungültig. Erst im zweiten Schritt wird wieder ein gültiger Zustand hergestellt. Dieser Bruch während des Refactorings sorgt für mehrere Probleme:

- Die Verallgemeinerung eines Teilrefactorings ist schwierig. Für die Modellierung eines Refactorings `RenameOperation` wird wie in `MoveOperation` ein Teilrefactoring benötigt, welches die Operationsaufrufe ändert. Es könnte versucht werden die beiden Anpassungen zusammenzuführen, da sie ähnlich sind. Da beide Refactorings für die Verhaltenserhaltung von anderen Gegebenheiten ausgehen, ist diese Vereinheitlichung schwierig. Das vereinheitlichte Teilrefactoring sollte mit allen zu nutzenden Refactorings zusammenarbeiten.

- Die Beweise eines Refactorings werden umfangreicher. Entweder wird jedes Refactoring (auch zusammengesetzte Refactorings) separat bewiesen, oder es existieren für manche Teilrefactorings unnatürlich generalisierte Beweise. Weiter muss es für manche Teilrefactorings verschiedene Beweise geben, die für die verschiedenen Refactorings genutzt werden.

- Änderungen an einem Teilrefactoring sind nicht isoliert. Es besteht die Gefahr, dass alle benutzten Refactorings angefasst werden müssen. Dies gilt sowohl für die Durchführung als auch für die Beweistechniken.

Die beschriebenen Probleme lassen sich verhindern, wenn alle Teilrefactorings selber wieder *Refactorings* sind. Das heißt, es werden keine Teilschritte in zusammengesetzten Refactorings genutzt die keine Refactorings sind. Diese Forderung löst auf der einen Seite die meisten der beschriebenen Probleme, zwingt auf der anderen Seite aber zu geänderten Entwurfstrategien von Refactorings. Im Folgenden wird die *echte Refactoring*-Forderung aus den zu den Problemen korrespondierenden Forderungen hergeleitet.

Für die Schwierigkeiten aus dem vorigen Abschnitt können jeweils einige Eigenschaften aufgezählt werden die, wenn sie erfüllt sind, das Problem nicht auftreten lassen.

- Die Änderung eines Teilrefactorings soll möglichst wenig Einfluss auf die Beweise der Soundness, Syntaxkorrektheit und Verhaltenserhaltung eines benutzenden Refactorings haben.
- Die Beweise der Soundness, Syntaxkorrektheit und Verhaltenserhaltung eines zusammengesetzten Refactorings sollen sich einfach aus den genutzten Teilrefactorings zusammensetzen.

Die Forderung, dass Teilrefactorings echte Refactorings sein sollen, vereinfacht die Modellierung von zusammengesetzten Refactorings ungemein. Dieser positive Effekt resultiert aus der Transitivität der Syntaxkorrektheit, der Wohlgeformtheit und der Verhaltenserhaltung. Ein zusammengesetztes Refactoring ist eine Hintereinanderreihung anderer Refactorings. Die Syntaxkorrektheit, Wohlgeformtheit und Verhaltenserhaltung folgt aus den entsprechenden Eigenschaften der Teilrefactorings. Wenn diese Eigenschaften erfüllt sein sollen, müssen die Teilschritte echte Refactorings sein. Nur die Soundness ist nicht transitiv und muss separat gezeigt werden.

6.2.2 Einführung und Eliminierung von Redundanzen

Aus der Forderung, dass alle Teilschritte eines Refactorings wieder Refactorings sind, ergibt sich scheinbar das Problem, dass sich einige Refactorings nicht beschreiben lassen. Ein Beispiel ist das oben beschriebene Refactoring `MoveOperation`. Dort soll die Operation verschoben werden. Mit einer geänderten Denkweise ist dieses Refactoring gut zu spezifizieren. Einführung und Elimination von Redundanzen können das Verschieben der Operation beschreiben. Dazu wird das Refactoring vereinfacht in drei Teile aufgespalten:

- Kopieren der Operation in die neue Klasse (Einführen der Redundanz);
- Anpassen der Operationsbenutzung;
- Entfernen der nicht mehr genutzten Operation (Eliminierung der Redundanz).

6.2.3 Kleine allgemeine Analysefunktionen

Die Analysefunktionen spielen in der Spezifikation der Refactorings eine wichtige Rolle. Sie erlauben es, den Code zu analysieren und dann die Calculation bzw. die Precondition zu

formulieren. In diesem Zusammenhang sind zwei Fragen interessant: Wie detailliert sollen die Funktionen sein und wie sollen sie nutzbringend eingesetzt werden? Diese Frage ist im Allgemeinen nicht zu beantworten. Deshalb sollen einige Hinweise gegeben werden, die zu einer geeigneten Auswahl der Analysefunktionen führen.

Vor der Betrachtung dieser Hinweise sollen einige Beobachtungen analysiert werden, die zu diesen Tipps führen. Die Analysefunktionen werden in Re$\mathcal{L}$ nur in der Precondition und in der Calculation genutzt. Gleichzeitig sind in diesen beiden Teilen eines Refactorings viele Sprachkonstrukte aus der Logik und Mengenlehre erlaubt. Deshalb gibt es verschiedene Möglichkeiten die Analysefunktionen zu nutzen, wobei ein Unterscheidungsmerkmal die Granularität ist. Es gibt zwei zu betrachtende Extremfälle. Der Erste ist, dass alle Bedingungen in die Analysefunktion gepackt werden und nur das Nötigste in der Precondition bzw. der Calculation umgesetzt wird. In dem Refactoring `SplitOperationCSPOZ` könnte dann die Precondition bzw. die Calculation wie folgt formuliert werden:

Precondition:
$preSplitOperation(\texttt{ops1}, \texttt{ops2}, \texttt{operation})$
Calculation:
Define:
```
dlist1, dlist2 : DeltaList;
v1,v2          : Declaration;
```
SCHEMA:
$\texttt{dlist1} = calcSplitOperationDlist1(\texttt{dlist})$
$\texttt{dlist2} = calcSplitOperationDlist2(\texttt{dlist})$
$\texttt{p1} = calcSplitOperationP1(\texttt{dlist1}, \texttt{v1})$
$\texttt{p2} = calcSplitOperationP2(\texttt{dlist2}, \texttt{v2})$
DONE

Zwei Dinge können beobachtet werden. Zum Einen ist nicht zu erkennen, worin die Bedingungen bestehen. Es müssen die Analysefunktionen betrachtet werden, um die Bedingungen und Berechnungen zu verstehen. Auch ist durch die Einschränkung, dass die Funktionen nur einen Rückgabewert haben, die Calculation auf mehrere Analysefunktionen aufgebaut. Die zweite Beobachtung besteht darin, dass sehr viele Analysefunktionen gebraucht werden. Diese müssen alle spezifiziert, programmiert und verifiziert werden.

Ein zweites Extrembeispiel ist die Verwendung von generischen Zugriffsfunktionen auf den AST. Als kleinste notwendige Menge werden „getChilds“, „getSlibing“, „getParent“, „getContent“ und „getProductionName“ benötigt. Es können alle atomaren Prädikate in der Precondition und Calculation damit ausgedrückt werden. Der Vorteil besteht in einer kleinen vollständige Menge von Analysefunktionen, die bereitgestellt und verifiziert werden müssen. Das Problem ergibt sich, wenn eine Precondition oder Calculation beschrieben werden soll. Es müssen dort aufwändig die benötigten Werte beschrieben werden. Die Prädikate werden sehr lang, unverständlich und schlecht wartbar.

Die beiden Extrembeispiele hängen direkt mit der Entwurfsstrategie für die Analysefunktionen zusammen. Wenn spezielle Funktionen für jedes Refactoring entwickelt werden, ist die Erstellung der Analysefunktionen ein Bestandteil der Refactoringserstellung. Wenn generische AST-Funktionen definiert werden, werden die Analysefunktionen zum Erstellungszeitpunkt der Re$\mathcal{L}$ Umgebung erstellt und sind dann ein Teil der Toolentwicklung.

Es hat sich gezeigt, dass die Definition und die Erstellung der Analysefunktionen weder der Refactoringerstellung noch der Toolerstellung zugeordnet sein sollte. Die Definition und Erstellung der Analysefunktionen stellt einen eigenen Schritt in der Umsetzung der Refactorings dar. Nachdem das Tool erstellt wurde, sollte eine Grundmenge an Analyse-

funktionen erstellt werden. Wenn sich später herausstellt, dass die erstellte Menge nicht ausreichend ist, kann diese erweitert werden.
Nachdem nun einige Designprinzipien für die Funktionen eingegrenzt wurden, sollen einige Hinweise auf eine gute Auswahl der Analysefunktionen gegeben werden.

Einfache sprachbezogende Analysefunktionen Die Auswahl der Funktionen soll die Eigenschaften der Sprache berücksichtigen. Bei CSP-OZ gibt es z.B. verschiedene Typen von Variablen (Input, Output, Simple, Delta, Secondary, State). Da häufig verschiedene Mengen von Variablen benötigt werden, sollen für diese verschiedenen Arten auch Funktionen zur Verfügung stehen, die diese aus einer Klasse oder aus einem Schema extrahieren können.

Überladen von Analysefunktionen Funktionen, die auf verschiedenen Sprachbestandteilen arbeiten, aber die gleiche Information berechnen, sollten mit dem gleichen Namen versehen werden. Wenn eine Funktion benötigt wird, die den Namen einer Klasse bestimmt und eine Funktion, die den Namen eines Schemas bestimmt, dann sollte nur eine Funktion *name* erstellt werden. Diese gibt in Abhängigkeit vom Typ des Parameters die gesuchte Information zurück.

Keine Spezialfunktionen für Refactorings Diese Forderung entspricht der Begründung vom Anfang dieses Abschnitts. Eine Funktion soll allgemein verwendbar sein und nicht nur für ein bestimmtes Refactoring. Jede Analysefunktion muss implementiert und auch auf Korrektheit geprüft werden.

Sinnvolle Rückgabewerte Die Verwendbarkeit der Analysefunktionen hängt stark von den Rückgabewerten ab. Eine Funktion die aus einem Z-Schema alle Variablen zurückgeben soll, kann so definiert sein, dass das Ergebnis eine Menge von Variablennamen (Strings) ist. Diese Möglichkeit stellt sich nach der Bearbeitung von mehreren Refactorings als umständlich heraus. So müssen ständig die Typen der Variablen geprüft werden, da diese in vielen Prädikaten übereinstimmen müssen. Es ist an dieser Stelle einfacher, als Rückgabewert eine Struktur mit Namen „Variable“ zu verwenden. Man kann dafür ein Tupel wählen, welches aus den Typen und den Namen besteht. Mit dieser Struktur kann implizit die Typgleichheit formuliert werden.

6.3 Analysefunktionen für CSP-OZ

Die Analysefunktionen spielen bei der Modellierung mit Re$\mathcal{L}$ eine entscheidende Rolle. Bevor im nächsten Kapitel die Verhaltenserhaltung von Refactorings betrachtet wird, sollen in diesem Abschnitt die für CSP-OZ benötigten Analysefunktionen eingeführt werden. Dies ist gleichzeitig auch ein Beispiel für die Auswahl der Analysefunktionen, wie sie im letzten Abschnitt beschrieben wurden. Zuerst wird untersucht, für welche Anwendungszwecke Analysefunktionen in CSP-OZ benötigt werden. Anschließend soll informal eine Aufzählung der Funktionen gegeben werden. Exemplarisch wird auch die genaue Definition gezeigt. Die restlichen Definitionen sind im Anhang zu finden.

Analysefunktionen werden meist genutzt, um Informationen, die nicht direkt aus dem Code mittels der Templates gewonnen werden können, zu bestimmen. In jeder bisher untersuch-

ten Sprache warfen Variablen ein solches Problem auf. Um die Menge der genutzten, bzw. der geschriebenen Variablen mit ihrem Typ zu gewinnen, müssten mehrere Subrefactorings genutzt werden. Da bietet es sich an, die Analyse von Variablen mit Analysefunktionen zu beschreiben. Dies ist besonders bei CSP-OZ nützlich, da CSP-OZ viele verschiedene Arten von Variablen (Input, Output, Simple, Secundary, State) besitzt. Von Variablen wird meist nicht nur der Name der Variable benötigt, sondern oft auch der Typ der Variable. In CSP-OZ ist auch die Art der Variablen wichtig. Die Art beschreibt, ob es sich um eine State-Variable, eine Eingabe-Variable oder um eine Ausgabe-Variable handelt. Daher soll mit den Analysefunktionen für die Variablen begonnen werden. Wie im vorherigen Abschnitt erläutert, spielen geeignete Typen für die Funktionen eine wichtige Rolle. Deshalb soll für die Darstellung der Variablen in den Prädikaten von Re$\mathcal{L}$ ein geeigneter Typ gefunden werden. Bei den Variablen in CSP-OZ sind der Name, der Typ und die Art der Variablen entscheidend. Deshalb werden hier die Variablen als ein Paar von Namen und Typ modelliert. Die Unterscheidung nach der Art wird später durch spezielle Analysefunktionen vorgenommen. Seien die Mengen $\mathcal{T}$ und $\mathcal{NAME}$ die Mengen aller Typen und aller Variablennamen, dann ist eine Variable ein Element des Kreuzproduktes:

$$\mathit{Variable} \in \mathcal{NAME} \times \mathcal{T}$$

Die Paare erlauben es, Variablen gleichzeitig nach dem Typ und dem Namen zu vergleichen. Dieses ist besonders nützlich, wenn Operationen auf Variablen-Mengen arbeiten, wie dieses im Abschnitt 7.2 geschehen wird. Wenn die Einzelinformationen des Namens bzw. des Types benötigt werden, kann dieses mit Hilfe der Ausdrücke in der Precondition oder Calculation selektiert werden. Der Lesbarkeit halber werden dafür separate Funktionen definieren. Eine weitere Funktion wird den Vergleich von Typ-kompatiblen Variablen erlauben.

$$\mathit{name} : \mathit{Variable} \rightarrow \mathcal{NAME}$$

$$\forall\, \mathit{vname} : \mathcal{NAME};\ \mathit{vtyp} : \mathcal{T} \bullet \mathit{name}((\mathit{vname}, \mathit{vtyp})) = \mathit{vname}$$

$$\mathit{type} : \mathit{Variable} \rightarrow \mathcal{T}$$

$$\forall\, \mathit{vname} : \mathcal{NAME};\ \mathit{vtyp} : \mathcal{T} \bullet \mathit{vtyp}((\mathit{vname}, \mathit{vtyp})) = \mathit{vtyp}$$

$$\mathit{compatible} : \mathit{Variable} \times \mathit{Variable} \rightarrow \mathbb{B}$$

$$\forall\, \mathit{vname}1, \mathit{vname}2 : \mathcal{NAME};\ \mathit{vtyp}1, \mathit{vtyp}2, : \mathcal{T} \bullet \mathit{compatible}((\mathit{vname}1, \mathit{vtyp}1), (\mathit{vname}2, \mathit{vtyp}2)) \leftrightarrow (\mathit{vname}1 = \mathit{vname}2 \wedge \mathit{vtyp}1 \in \mathbf{stype}(\mathit{vtyp}2))$$

Nachdem die Variablen und einige zugehörige Definitionen vorhanden sind, sollen die Funktionen betrachtet werden, die die verschiedenen Variablenmengen berechnen. Es gibt für jede Art von Variablen eine Funktion und zusätzlich eine, die alle Variablen unabhängig von der Art sammelt (**vars**). Diese Funktionen lassen sich auf verschiedene Einheiten einer Spezifikation anwenden. Wenn eine Funktion auf ein Schema angewandt wird, so werden Variablen des Schemas berechnet. Im Falle einer Operation werden entsprechend die Variablen der Operation zurückgegeben.

Funktionsname	Beschreibung
vars	Gibt die Menge aller Variablen zurück.
delta	Gibt die Menge aller Delta-Variablen zurück.
input	Gibt die Menge aller Input-Variablen zurück.
output	Gibt die Menge aller Output-Variablen zurück.

Die oben schon für Variablen definierte Funktion **name** wird auch für andere Objekte benötigt. Wenn eine Operation in einer Meta-Variablen gespeichert ist, und deren Namen bestimmt werden soll, ist die Funktion **name** sehr hilfreich. Ähnlich ist es mit Klassen, Operation u.a.

Name	Parameter	Funktion
name	c : class0	Gibt den Klassennamen von c zurück.
	s : schema	Gibt den Schemanamen von s zurück.
	o : Operation	Gibt den Operationsnamen von o zurück.
	v : Variable	Gibt den Variablennamen von v zurück.

Häufig wird nicht nur der Name eines einzelnen Konstruktes benötigt. Dabei hilft die Funktion **names**, welche für eine Menge von Klassen, Operatoren oder Variablen die Menge der Namen zurück gibt.

In CSP und CSP-OZ spielen Kanäle und Events bei der Synchronisation eine wichtige Rolle. Daher werden Funktionen benötigt, die die Mengen der Kanäle bzw. Events berechnen. Zusätzlich sind die Traces eines CSP-Prozesses interessant.

Funktionsname	Beschreibung
channels	Gibt die Menge der Kanäle zurück.
events	Gibt die Menge aller Events zurück.
traces	$= \mathcal{T}(P)$ Gibt die Menge der Traces eines Prozesses zurück.

Damit sind die zentralen Analysefunktionen für CSP-OZ aufgezählt. Es fällt auf, dass es nur eine kleine Menge von Funktionen ist, die ausreicht, um alle in dieser Arbeit behandelten Refactorings in CSP-OZ zu beschreiben. Formale Definitionen der Funktionen finden sich im Anhang B. Im nächsten Kapitel werden diese Refactorings genauer auf die Verhaltenserhaltung untersucht.

Kapitel 7

Korrektheit von Refactorings

Refactorings sollen das Verhalten des Modells oder des Programms erhalten. Die Sicherstellung der Verhaltenserhaltung ist eine zentrale Aufgabenstellung bei der Ausführung von Refactorings. In diesem Kapitel werden Techniken für die Prävalidierung von Refactorings betrachtet. Ziel ist es, *vor* der Ausführung sicher zu sein, dass das Verhalten erhalten wird. Um dies feststellen zu können, muss festgelegt werden, wie sich das Verhalten ausdrückt, und was es heißt, dass es sich nicht verändert. Nachdem die Verhaltenserhaltung beschrieben ist, werden anschließend verschiedene Typen von Refactorings betrachtet. Diese Typen unterscheiden sich durch ihre innere Struktur. Diese zeigt sich besonders im Vorgehen bei den Korrektheitsbeweisen.

7.1 Das Problem der Korrektheit von Refactorings

Ein Refactoring soll nach seiner Definition das Verhalten eines Programms oder eines Modells nicht verändern. Diese Eigenschaft eines Refactorings hört sich einfach an, birgt aber eine Menge Probleme und Fragen. Das fängt bei der Grundlage, der Verhaltenserhaltung, an. Was bedeutet es, dass ein Programm oder ein Modell sein Verhalten behält? Die Antwort auf diese Frage muss das Problem von unzureichenden Semantik-Definitionen genauso lösen wie das Problem von verschiedenen Detaillierungsgraden von Spezifikationen. Als weiterer Punkt spielt die Wechselwirkung der Verhaltenserhaltung mit den Techniken für ihre Validierung eine besondere Rolle. Die Validierung der Verhaltenserhaltung wird der zweite Themenblock in diesem Kapitel sein. Ein entscheidender Punkt ist, dass die Beschreibung eines Refactorings für eine Validierung geeignet ist. Bei der Entwicklung von Re$\mathcal{L}$ (Kapitel 5) wurde auf eine für Beweise geeignete Beschreibung geachtet. Daher findet sich in den Beweisen die Terminologie von Re$\mathcal{L}$ wieder. Der Fokus in der Validierung der Verhaltenserhaltung wird auf die Prävalidierung mittels Beweisen gelegt werden. Ein zweiter Schwerpunkt wird sein, die Techniken auf Modelle mit mehreren Sichten anzuwenden. Dabei werden verschiedene Techniken von Refactorings auf mehreren Sichten betrachtet, deren Modellierung und passende Beweistechniken diskutiert werden. Bevor diese Themen in den nächsten Unterkapiteln im Detail behandelt werden, sollen im Rest dieses Unterkapitels die einzelnen Bereiche kurz vorgestellt werden und die Wechselwirkung vorweggenommen werden, soweit sie für das Verständnis notwendig sind.

Als Erstes wird die Verhaltenserhaltung betrachtet. Auffallend ist, dass nicht von Semantikerhaltung gesprochen wird. Warum dies so ist, wird aus der Geschichte der Entwicklung von

Refactorings ersichtlich. Refactorings wurden zuerst als Strukturveränderungen in der objektorientierten Entwicklung mittels SmallTalk eingeführt [Opd92]. Die Entwicklung geht von der Objekt-Orientierung aus. Ein zentraler Begriff in der Objekt-Orientierung ist das Subtyping von Klassen. Mit diesem Begriff wird beschrieben, dass ein Subtyp ein Verhalten zeigt, so dass ein Nutzer, der den Typ erwartet, nicht merkt, dass er mit dem Subtyp arbeitet. Diese Sichtweise beinhaltet insbesondere eine Sicht der Nutzung der Klasse, die in erster Approximation vollständig unabhängig von ihrem inneren Zustand ist. Opdyke [Opd92] baut auf diesem Begriff des kompatiblen Verhaltens den Begriff der Verhaltenserhaltung auf. Dabei geht es um das beobachtbare Verhalten und nicht um eine strikte Erhaltung der Semantik. Als noch keine umfassende Toolunterstützung für Refactorings zur Verfügung stand, wurde dies mit dem Testen der Programme vor und nach dem Refactoring im Vordergrund gleichgesetzt. Die Testfälle, die das Programm oder ein Teil davon von außen betrachten, sollen kein anderes Verhalten zeigen. Das Vorgehen zeigt, dass sich die Verhaltenserhaltung vom Subtyping unterscheidet. Es soll nicht nur kompatibles Verhalten gezeigt werden, sondern es soll auch kein geändertes Verhalten, insbesondere kein neues Verhalten, auftreten. Dies ist besonders wichtig, wenn sich Nutzer einer Bibliothek auf ein bestimmtes Verhalten verlassen. Wenn sie mit einer refaktorisierten Implementierung der Bibliothek konfrontiert werden, wollen diese nicht deswegen ihre Programme verändern.

Auch wenn die Verhaltenserhaltung schwächer als eine Semantikerhaltung ist, muss dieser Begriff auf der Semantik aufbauen. Die Semantik beschreibt dabei in der Regel mehr als das sichtbare Verhalten. Mit einer Semantik soll das Verhalten universell und vollständig beschrieben werden. Das schließt dabei nicht aus, dass bestimmte Bereiche des Gesamtverhaltens bewusst offen gelassen werden. Der Verhaltenserhaltungsbegriff muss daher eine Relation zwischen zwei Semantiken herstellen, wenn diese beiden Semantiken das gleiche beobachtbare Verhalten zeigen. In der Definition von Refactoring wurde daher die Verhaltenserhaltung als Relation $\preceq$ auf den semantischen Domänen eingeführt. Die Schnittmenge aller möglichen Verhaltenserhaltungsbegriffe ist gerade die Semantikerhaltung, das heißt, die Semantikerhaltung ist die stärkste mögliche Art der Verhaltenserhaltung.

Da die Verhaltenserhaltung eine Relation auf der Domäne der Semantik ist, ist die Art und Struktur der Semantik für die Verhaltenserhaltung entscheidend. Leider wird das Verhalten von Programmen häufig nicht in der Form einer formalen Semantik angegeben. Z.B. ist die „Semantik“ von Java entweder als natürlichsprachliche Beschreibung [Javc] oder als Referenzimplementation [Java] gegeben. Beide Arten erlauben keine formale Analyse der Verhaltenserhaltung. Bei diesen Arten von Semantiken kann die Verhaltenserhaltung nur durch die Beobachtung des Verhaltens definiert werden. Die Verhaltenserhaltung kann (bis auf pathologisch kleine Fälle) nicht im Ganzen geprüft werden. Formale Semantiken erlauben eine formale Definition der Verhaltenserhaltung. Der Vorteil dieser formalen Definition ist, dass die formalen Techniken für die Validierung der Verhaltenserhaltung genutzt werden können. Seit einiger Zeit wird versucht für wichtige Sprachen brauchbare formale Semantiken zu schaffen. Interessant sind dabei die Arbeiten zur Semantik von Java [Zam03] bzw. für die Semantik von UML [EW01, FS07, KGKK02, Szl06]. Im weiteren dieses Kapitels soll daher davon ausgegangen werden, dass eine Sprache eine formale Semantik besitzt.

Ein weiteres Problem tritt auf, wenn Refactorings von Programmiersprachen auf Modellsprachen übertragen werden. In vielen Modellsprachen wird nicht gefordert, dass alle Informationen auf dem gleichen Detaillierungslevel spezifiziert sind. So erlaubt z.B. die UML, dass die Operationen einer Klasse verschieden stark spezifiziert sind. Dazu soll eine

Klasse *Rechne* mit drei Operationen „add“, „sub“ und „calc“ betrachtet werden.

1. int add (int a, int b)
2. int sub (int a, int b)
   - pre: true
   - post: result = a - b
3. int calc(int a) uses vars x,y ops add, sub

Die verschiedenen Operationen sind dabei unterschiedlich stark spezifiziert. Die Operation „add“ hat nur die Signatur gegeben. Die Operation „sub“ ist neben der Signatur mittels einer Vorbedingung und einer Nachbedingung vollständig spezifiziert. Bei der Operation „calc“ ist gegeben, welche anderen Operationen bzw. Klassenvariablen von ihr genutzt werden. Wenn eine Verhaltenserhaltung auf diesem Modell definiert werden soll, ist dies schwierig. Ein erster Ansatz ist es, die stärkste Verhaltenserhaltung zu wählen, die von allen Operationen erfüllt werden kann. Es wird also von der am schwächsten definierten Operation ausgegangen. Wegen der Spezifikation von „add“ wird hier die Beibehaltung der Signatur als Ausgangspunkt genommen. Dadurch ergibt sich das Problem, dass die Operation „sub“ mit der folgenden Definition verhaltenserhaltend zu „sub“ von oben ist:

1. int sub (int a, int b)
   - pre: true
   - post: result = a + b

Der Erhaltungsbegriff erhält nicht das, was naiverweise erwartet würde. Eine einzelne Art von Verhaltenserhaltung kann in diesem Fall nicht helfen. Die Lösung, die in dieser Arbeit genutzt wird (vgl. Kapitel 7.2), ist die Verwendung von verschiedenen Erhaltungsbegriffen, die eine Hierarchie bilden. In diesem Beispiel könnte für die Operation „sub“ ein stärkerer Erhaltungsbegriff verwendet werden, der zusätzlich auch die Signatur erhält. An den verschiedenen Stellen in der Spezifikation kann dann ein adäquater Begriff verwendet werden.

Die Hierachien der Verhaltenserhaltung sind besonders hilfreich, wenn gleichzeitig Beobachtungspunkte eingeführt werden. Ein Beobachtungspunkt ist grob gesagt ein Ort innerhalb des Codes, an dem die Verhaltenserhaltung geprüft wird (beobachtet), und von dem ausgehend eine Verhaltenserhaltung im Gesamtsystem impliziert wird. Dabei stehen die Verhaltenserhaltung am Beobachtungspunkt und die Verhaltenserhaltung des Gesamtsystems in keiner Relation. Eine Verhaltenserhaltung am Beobachtungspunkt kann eine stärkere oder eine schwächere Verhaltenserhaltung im Gesamtsystem hervorrufen. Besonders interessant ist der Fall, in dem eine schwache Verhaltenserhaltung an einem Beobachtungspunkt eine starke Verhaltenserhaltung des Gesamtsystems impliziert. Beispielsweise kann aus der Failures-Äquivalenz des Z-Teils bei CSP-OZ auf die Failures-Divergences-Äquivalenz der Gesamtklasse geschlossen werden.

Der zweite Block in diesem Kapitel behandelt die Sicherstellung der Verhaltenserhaltung. Es werden zwei Gruppen von Verfahren unterschieden. Die erste Gruppe stellt nach der Durchführung des Refactorings fest, ob das Verhalten erfolgreich erhalten wurde, ansonsten wird das Ergebnis verworfen. Diese *Postvalidierung* [Fow04] wird häufig in der Softwareentwicklung genutzt, indem geprüft wird, ob die Testfälle für den Code vor und nach dem

Refactoring die gleichen Ergebnisse liefern. Die zweite Gruppe ist die *Prävalidierung*, bei der vor der Durchführung für die Transformation im allgemeinen die Verhaltenserhaltung analysiert wird. Dabei kann in der Prävalidierung durch Testen des Refactorings ein großes Vertrauen in das Refactoring gewonnen werden. Wünschenswert ist es, dass die Prävalidierung mittels formaler Methoden, z.B. durch Beweise verifiziert wird. Die Postvalidierung ist ein gut verstandener Bereich, der durch Tests bzw. durch direkte Beweise (z.B. Refinement [Cor04]) verstanden ist. Daher soll in dieser Arbeit nur die Prävalidierung betrachtet werden. Ziel der Prävalidierung ist es, ein Refactoring so durchzuführen, dass nach der Anwendung darauf vertraut werden kann, dass das Verhalten erhalten wurde.

Die Refactoringssprache Re$\mathcal{L}$ liefert einige Voraussetzungen für die formale Analyse von Refactorings. Eine besondere Rolle spielen dabei die Meta-Variablen. Meta-Variablen können Code-Stücke enthalten, die sich aus einem Nicht-Terminal erzeugen lassen. In vielen Semantikdefinitionen von BNF-basierten Sprachen wird die Semantik gerade auf Ebene der Nicht-Terminale definiert, d.h. die Definition einer operationalen Semantik baut genau die Produktionsschritte nach. Diese Beobachtungen zeigen den Weg für die formale Analyse auf. In den Meta-Variablen sind Code-Stücke enthalten, denen eine (Teil-)Semantik zugeordnet ist. Da sich der Inhalt einer Meta-Variable in Re$\mathcal{L}$ nicht verändern kann, bleibt die Semantik des in der Meta-Variablen gespeicherten Codes bestehen. Die Idee soll an einem einfachen mathematischen Beispiel erläutert werden.

Das Verhalten der Meta-Variablen ähnelt funktionswertigen Variablen in der Mathematik: Seien drei Funktionsvariablen f_1, f_2 und f_3 gegeben. Für reellwertige Funktionen gilt die Distributivität $f_1 * (f_2 + f_3) = f_1 + f_2 * f_1 + f_3$. Die Distribution spielt in diesem Beispiel die Rolle des Refactorings. Die linke Seite spielt dabei die Rolle des BeforeTemplates und die rechte Seite fungiert als AfterTemplate. Die Funktionen selber verändern sich während der Anwendung nicht. Dass das Ergebnis für die Gleichung erfüllt ist, ist dadurch gegeben, dass die Variablen immer die gleiche Semantik haben. Diesen Effekt werden wir bei den Beweisen der Refactorings ausnutzen.

Der dritte und letzte Block in diesem Kapitel wird sich mit Refactorings mehrerer Sichten befassen. Viele Modelle werden nicht innerhalb einer homogenen, geschlossenen Art wie Programmen beschrieben. Es werden verschiedene Aspekte oft in verschiedenen Arten von Diagrammen oder mittels kombinierter Formalismen zusammengestellt. Der Vorteil dieser Fenster auf das Modell ist, dass sich der Entwickler bei der Modellierung auf die jeweils wichtigen Teilaspekte konzentrieren kann. Die Aufteilung erschwert aber die Durchführung von Refactorings, weil alle Sichten beachtet werden müssen. Es wird sich zeigen, dass die Verwendung von Verhaltenserhaltungshierarchien zusammen mit den Beobachtungspunkten genutzt werden kann, um die Verhaltenserhaltung zu analysieren. Dabei werden die im vorherigen Abschnitt angesprochenen Beweistechniken genutzt, um die Verhaltenserhaltung der Änderungen zu beweisen. Die Beobachtungspunkte helfen die verschiedenen Sichten mit einem zur Sicht passenden Verhaltenserhaltungsbegriff zu analysieren. Anschließend wird mit Hilfe der Hierarchie der Erhaltungsbegriffe die Verhaltenserhaltung für das Gesamtsystem hergeleitet.

7.2 Die Verhaltenserhaltung

Refactorings werden als verhaltenserhaltende Veränderungen von Programmen oder Spezifikationen definiert. Eine zentrale Frage bei dieser Definition ist, was Verhaltenserhaltung heißt. Diese Frage ist besonders bei Modellen und Spezifikationen schwer zu beantworten, weil diese kein ausführbares System bilden, welches beobachtet werden könnte. Am Anfang wird anhand der Verhaltenserhaltung in Programmen ein Einblick in die Problematik der Verhaltenserhaltung gegeben.

7.2.1 Verhaltenserhaltung bei Programmen

Verhaltenserhaltung bei Programmen wird seit Odyke [Opd92] als das beobachtbare Verhalten eines Programms betrachtet. Ein Programm soll nach einem Refactoring unter den gleichen Umständen auf die gleichen Eingaben die gleichen Ausgaben liefern wie zuvor. In der Programmentwicklung werden daher häufig Tests durchgeführt, um festzustellen, ob sich das Verhalten geändert hat. Besonders die Verwendung von automatisierten Tests, wie mittels JUnit [jun], helfen dem Entwickler, weil er aufwandssparend und regelmäßig das Programm prüfen kann. Ein Problem dieser Tests ist, dass sie oft nicht vollständig sind. Wenn ein Ablauf des Programms sich verändert, welcher nicht durch einen Test abgedeckt wird, bleibt dies unbemerkt. Ein anderer Ansatz in der Programmentwicklung ist, sich auf Tools zu verlassen, die Refactorings durchführen. Viele Tools wie Eclipse [Fou], Netbeans [Sun] oder auch VisualStudio [Cor] unterstützen Refactorings. Der Entwickler verlässt sich in diesem Fall auf die korrekte Durchführung durch das Tool. Dieses Vertrauen ruht auf der Erfahrung vieler anderer Entwickler mit dem Tool nach dem Motto: „Wenn bei einer großen Anwendergemeinde nichts schief geht, wird auch hier nichts schief gehen."

Ein Problem ist dabei, dass sich die Entwickler ganz auf das Tool verlassen, obwohl in Sonderfällen nicht klar ist, ob das Refactoring korrekt ausgeführt werden kann. In Eclipse treten regelmäßig solche Fälle auf. Die Lösung innerhalb von Eclipse sieht deshalb vor, dass in solchen Fällen die Code-Änderungen dem Entwickler zur Begutachtung vorgelegt werden. (vgl. Abbildung 5.2) Der Entwickler kann dann interaktiv entscheiden, ob die Änderungen in Ordnung sind oder sie entsprechend verändern.

7.2.2 Verhaltenserhaltung bei Modellen

Modelle abstrahieren Eigenschaften, d.h. sie sind eine Vereinfachung und Abstraktion der zu erstellenden Software. Erst durch diese Vereinfachung in den Modellen werden die eigentlichen Stärken der modellgetriebenen Entwicklung deutlich. Diese Eigenschaft erschwert die Verhaltenserhaltung. Ein Modell enthält evtl. nicht wie ein Programm alle Informationen. Eine Eigenschaft, die in einem Modell nicht modelliert wird, kann auch nicht erhalten werden. Daher werden für verschiedene Modelle verschiedene Ansätze der Verhaltenserhaltung definiert (siehe Tabelle 7.1). Das Finden einer geeigneten Verhaltenserhaltung wird weiter erschwert, da Modelle es erlauben Dinge auszulassen, obwohl diese modelliert werden könnten. Besonders schwierig wird es, wenn diese Eigenschaften an einigen Stellen im Modell modelliert sind und an anderen nicht. So ist es möglich in UML für einige, aber nicht alle, Operationen Vor- und Nachbedingungen via OCL anzugeben.

Eine weitere Schwierigkeit bei der Definition der Verhaltenserhaltung ist, dass nicht alles erhalten bleibt, sondern (manchmal) zwingend Änderungen vorgenommen werden müssen. Ein (pathologischer) Extremfall ist, wenn ein Programm textuell unverändert bleiben

access preserving Jede Methodenimplementation greift auf die gleichen Variablen zu wie vor dem Refactoring. Diese Zugriffe können auch transitiv durchgeführt werden, indem andere Methoden (direkt oder indirekt) aufgerufen werden, die die Variablen benutzen. [MDJ02]

update preserving Jede Methodenimplementation verändert auf die gleichen Variablen zu wie vor den Refactorings. [MDJ02]

call preserving Von jeder Methodenimplementation werden intern die gleichen Methoden wie vor dem Refactoring genutzt. [MDJ02].

object-preserving Die Objektstruktur einschließlich der Hierarchie darf nicht verändert werden. Die Objektstrukur beinhaltet dabei nur die Referenzstruktur und die Operationenstruktur. Namen sind z.B. nicht enthalten. [Ber91]

Tabelle 7.1: Verhaltenserhaltungsbegriffe nach [MT04] für Modelle

soll. Ohne eine textuelle Änderung sind überhaupt keine Veränderungen möglich. Dieses Verständnis der Verhaltenserhaltung ist zu stark. Ein erster Ansatz dies zu lösen ist, die Semantikerhaltung zu fordern. Auch dies ist ein häufig zu starker Erhaltungsbegriff. In Object-Z betrachtet die Z-Semantik auch den State einer Klasse. Somit sind manche Änderungen innerhalb des State-Schemas nicht möglich, die zwar das Innere der Klasse verändern, aber sich nicht auf das Verhalten der Klasse auswirken. In diesem Fall ist die Semantikerhaltung zu stark.

Das Wort *Verhaltenserhaltung* drückt aus, dass es um das Verhalten des Modells geht. Wie bei Programmen ist damit das beobachtbare Verhalten gemeint. Dieses Verhalten kann aus verschiedenen Blickwinkeln betrachtet werden, aber von irgendeiner Schnittstelle ausgesehen, sei es API oder GUI, ist das Verhalten gleich. Dabei spielt auch die Vergangenheit eine Rolle. Ein Begriff aus den formalen Techniken, der die geforderten Eigenschaften aufweist, ist das *Refinement* (vgl. auch Abschnitt 3.4).

Refinement beschreibt den Aspekt der Beobachtung sehr gut. Dieser Begriff besitzt noch einige Schwächen, die aufbauend auf das Refinement behoben werden können.

- Bei Modellen wird das Refinement zur schrittweisen Verfeinerung des Modells bis zum fertigen Programm verwendet. Ein Refactoring in einem solchen Kontext sollte dieses nicht zerstören. Dies wird nicht durch den Verhaltenserhaltungsbegriff des einfachen Refinements sichergestellt. Die Verwendung einer Refinement-Äquivalenz hilft den Kontext zu bewahren.

- Die Verhaltenserhaltung eines Teiles des Modells soll sich auf die Verhaltenserhaltung des Gesamtmodells übertragen. So ist das Data Refinement von Object-Z nicht kompositionell, dies gilt ebenso wenig für die Refinement-Äquivalenz. Deshalb ist wegen einer fehlenden Kongruenz die Nutzung dieser Begriffe für reine Object-Z-Klassen ungeeignet. Daher wird für reine Object-Z-Klassen in dieser Arbeit die Semantikerhaltung genutzt.

Zusammenfassend gibt es verschiedene Anforderungen an die Verhaltenserhaltung, die von den benötigten Anforderungen abhängen. Einige Refactorings sind davon abhängig, dass

der Verhaltenserhaltungsbegriff schwach genug ist, so dass die Änderung verhaltenserhaltend ist. Auf der anderen Seite muss für andere Eigenschaften (z.B. Kompositionalität) die Verhaltenserhaltung stark genug sein.

Bei Modellsprachen gibt es nicht *den* Verhaltenserhaltungsbegriff. Hilfreich ist, die Verhaltenserhaltungsbegriffe so zu wählen, dass sie eine Hierarchie ergeben, die beschreibt, aus welchem Verhaltenserhaltungsbegriff ein anderer folgt. Ein Vorteil liegt darin, dass ein Refactoring bzgl. eines Verhaltenserhaltungsbegriff auch ein Refactoring gegenüber den darunterliegenden Begriffen ist. Ein Beispiel für eine Verhaltenserhaltungshierarchien findet sich im Kapitel 7.2.4 über die Verhaltenserhaltungsbegriffe von CSP-OZ.

7.2.3 Der Beobachtungspunkt

Wenn die Verhaltenserhaltung geklärt ist, kann für ein Refactoring gezeigt werden, dass es verhaltenserhaltend ist. Wenn kleine Änderungen tief im Code durch die Transformation durchgeführt werden, ist es aufwändig, die Verhaltenserhaltung für das Gesamtsystem zu zeigen. Eine Abhilfe bieten die Beobachtungspunkte, die es erlauben nur einen Ausschnitt des Codes zu betrachten.

Ein einfaches Beispiel sind Java-Programme. Wenn in einer Klasse eine private Instanzvariable umbenannt wird, dann müssen zwar die Methoden in der Klasse angepasst werden, aber die Änderung ist nicht von außerhalb der Klasse zu beobachten. In diesem Fall reicht es die Verhaltenserhaltung der Klasse sicherzustellen. Aus dieser Verhaltenserhaltung der Klasse folgt die Verhaltenserhaltung des Gesamtsystems.

Bei Modellsprachen funktioniert das Vorgehen ähnlich, nur im Detail muss auf mehr Aspekte geachtet werden, da Modelle z.B. mehrere Sichten besitzen können.

Definition 31 (Beobachtungspunkte in BNF-basierten Sprachen) *Sei eine Sprache $P = (A, L, D, [\![.]\!])$ mit kompositioneller Semantik, sei $\simeq_g$: $D \times D$ ein Verhaltenserhaltungsbegriff, und sei $G = (NTerm, Term, Prod, S_0)$ eine BNF für L. Ein Paar $(pov, \simeq_l)$ ist ein* Beobachtungspunkt bzgl. $\simeq_l$ *mit $pov \in NTerm$ und $\simeq_l$: $D \times D$, wenn*

$$
\begin{array}{l}
\forall\, \lambda_I \in (Term)^* pov (Term)^*;\ \lambda_x, \lambda_y \in L;\ x, y \in (Term)^* \mid \\
\quad \exists\, s_x, s_y : seq\, NTerm \mid \\
\qquad\qquad S_0 \rightsquigarrow_G \lambda_I \overset{s_x}{\rightsquigarrow}_G \lambda_x \wedge \\
\qquad\qquad S_0 \rightsquigarrow_G \lambda_I \overset{s_y}{\rightsquigarrow}_G \lambda_y \wedge \\
\qquad\qquad pov \overset{s_x}{\rightsquigarrow}_G x \wedge \\
\qquad\qquad pov \overset{s_y}{\rightsquigarrow}_G y \\
\qquad\qquad\qquad\qquad \bullet\ [\![x]\!] \simeq_l [\![y]\!] \Rightarrow [\![\lambda_x]\!] \simeq_g [\![\lambda_y]\!].
\end{array}
$$

Die Relation $\simeq_l$ wird dann lokale Verhaltenserhaltung und die Relation $\simeq_g$ globale Verhaltenserhaltung genannt.

Bemerkung 6 *Der Ausdruck $(Term)^* pov (Term)^*$ soll die Menge aller Wörter, die als einziges Nichtterminal das Nichtterminal pov enthalten, beschreiben (vgl. Definition 49 Anhang A).*

Die Variable λ_I beschreibt Code, der vollständig produziert ist, bis auf die Stelle, an dem der Beobachtungspunkt sitzt. In diese „Lücke“ wird einmal der Code aus x und einmal der Code aus y eingesetzt. y und x sind dabei aus dem Nichtterminal *pov* generiert worden.

Der vollständige Code wird mit λ_x und λ_y bezeichnet. Die Traces der Produktionen stellen entsprechende Einsetzungen sicher. Die Kernaussage der Definition ist die Ableitung der Verhaltenserhaltung des Gesamtcodes aus dem der Teile x und y.

Definition 32 (Passender Beobachtungspunkt) *Seien die Bezeichnungen wie in Definition 31. Dann ist* $(pov, \simeq_l)$ *ein zum Refactoring* $Re = (pre, T, \backsimeq)_P$ passender Beobachtungspunkt*, wenn*

$$
\begin{array}{l}
\forall\, \lambda_I \in (Term)^* pov (Term)^*;\ \lambda_y \in L \mid \\
\quad \exists\, \lambda_x \in L; \mid \\
\qquad S_0 \rightsquigarrow_G \lambda_I \rightsquigarrow_G \lambda_x \wedge \\
\qquad S_0 \rightsquigarrow_G \lambda_I \rightsquigarrow_G \lambda_y \wedge \\
\qquad \bullet\ \lambda_x = T(\lambda_y).
\end{array}
$$

Für jedes Refactoring kann es mehrere passende Beobachtungspunkte geben. Das Gesamtsystem zusammen mit der Verhaltenserhaltung des Systems ist für jedes Refactoring ein Beobachtungspunkt. Wenn ein Refactoring mehrere Beobachtungspunkte besitzt, kann der Beweis der Verhaltenserhaltung auf den Beobachtungspunkt beschränkt werden.

Den Beobachtungspunkten in Definition 31 ist ein lokaler Verhaltenserhaltungsbegriff zugeordnet, welcher sich vom globalen Verhaltenserhaltungsbegriff des Gesamtsystems unterscheiden kann. Dies ist in zwei Fällen nützlich. Zum Einen kann für den betrachteten Teil des Codes ein anderer Verhaltenserhaltungsbegriff genutzt werden. Zum Anderen erlaubt dies für mehrsichtige Sprachen die lokale Verhaltenserhaltung zu verwenden.

Die Beobachtungspunkte sollen im Zusammenhang mit den Verhaltenserhaltungsbeweisen genutzt werden. Dazu ist es nötig einen zum Refactoring passenden Beobachtungspunkt zu wählen. Bei den Analysen von Refactorings von CSP-OZ hat sich gezeigt, dass dies, wie die Beweise, ein kreativer Prozess ist. Generell soll ein passender Beobachtungspunkt gewählt werden, der eine möglichst kleine Menge Code beschreibt, die durch das Refactoring geändert wird.

Während die Beweise erstellt werden, wird versucht, diese in Bezug auf einen Beobachtungspunkt zu führen. Teilweise stellt sich bei der Suche nach einem Beweis heraus, dass ein Refactoring an dem angenommenen Beobachtungspunkt gar nicht verhaltenserhaltend ist. Dann müsste die Suche nach einem Beweis an einem anderem Beobachtungspunkt fortgesetzt werden. Wenn auf der anderen Seite einen Beobachtungspunkt gewählt wird, der zu weit von der Änderung entfernt ist, dann ist beim Beweis zu sehen, dass viele nicht relevante semantische Regeln anwendet werden müssen. In der Regel wird mit einem gutem Beweis auch ein geeigneter Beobachtungspunkt gefunden.

7.2.4 Verhaltenserhaltungsbegriffe und Beobachtungspunkte für CSP-OZ

Die integrierte formale Methode CSP-OZ bietet mit ihren verschiedenen Sichten und lokalen Semantiken ein schönes Beispiel für die Definition der Verhaltenserhaltung und der Beobachtungspunkte. Als Erstes werden die Verhaltenserhaltungsbegriffe für CSP-OZ betrachtet.

Wie in Kapitel 7.2.2 diskutiert, ist die Auswahl der Verhaltenserhaltungsbegriffe ein Prozess, der von den Anforderungen, die erfüllt werden müssen, abhängt. Diese Abhängigkeiten führen dazu, dass es sinnvoll ist, nicht nur einen Erhaltungsbegriff zu betrachten

Verhaltenserhaltung	Definition
Verhaltenserhaltung auf einer CSP-Semantik bzw. Transitionssystemen	
$\sqsubseteq_{\mathcal{T}}$	Trace-Refinement nach Definition 21
$\sqsubseteq_{\mathcal{F}}$	Failure-Refinement nach Definition 24
$\sqsubseteq_{\mathcal{FD}}$	Failure-Divergences-Refinement nach Definition 25
$\equiv_{\mathcal{T}}$	$A \equiv_{\mathcal{T}} B := A \sqsubseteq_{\mathcal{T}} B \wedge B \sqsubseteq_{\mathcal{T}} A$
$\equiv_{\mathcal{F}}$	$A \equiv_{\mathcal{F}} B := A \sqsubseteq_{\mathcal{F}} B \wedge B \sqsubseteq_{\mathcal{F}} A$
$\equiv_{\mathcal{FD}}$	$A \equiv_{\mathcal{FD}} B := A \sqsubseteq_{\mathcal{FD}} B \wedge B \sqsubseteq_{\mathcal{FD}} A$
$\approx$	Schwache Bisimulation nach Definition 27
$\sim$	Starke Bisimulation nach Definition 26
Verhaltenserhaltung auf einer Z-Semantik	
$\sqsubseteq_{DS}$	Downwardsimulation nach Definition 18
$\equiv_{DS}$	$A \equiv_{DS} B := A \sqsubseteq_{DS} B \wedge B \sqsubseteq_{DS} A$
$\sqsubseteq_{iDS}$	Downwardsimulation nach Definition 18 mit einer injektiven Funktion als Retrieve-Relation
$\equiv_{iDS}$	$A \equiv_{iDS} B := A \sqsubseteq_{iDS} B \wedge B \sqsubseteq_{DS} A$
$\sqsubseteq$	Refinement
$\equiv_{\sqsubseteq}$	Es gilt: $A \sqsubseteq C$ und $C \sqsubseteq A$
$\equiv_{[\![]\!]}$	Semantikerhaltung

Tabelle 7.2: Verhaltenserhaltungsbegriffe für Object-Z und CSP-OZ

sondern mehrere. Hierbei bietet sich der Begriff des Refinements als Ausgangspunkt an. Für CSP-OZ können dies das Trace-, Failure- und Failure-Divergences-Refinement sein. Da diese Refinementbegriffe nicht kompatibel zu einer bestehenden Refinement-Struktur sind, bietet es sich an auch die entsprechenden Refinement-Äquivalenzen mit in die Verhaltenserhaltungsbegriffe aufzunehmen (vgl. Tabelle 7.3).

Für CSP-OZ sind zwei Eigenschaften, die eine Verhaltenserhaltung implizieren können, für diese Arbeit interessant:

Kongruenz Die Kongruenz beschreibt die Verträglichkeit einer Äquivalenzrelation mit den Operatoren eines Formalismus und kommt aus der Algebra.

Definition 33 (Kongruenz (vgl. [Sch99]))
Sei $\simeq$ eine Äquivalenzrelation auf einer Menge D. Die Äquivalenzrelation $\simeq$ ist eine Kongruenz *bezüglich eines n-stelligen Operators op auf M gdw. gilt:*

$$\forall\, i = 1 \ldots n : \; P_i \simeq P_i' \;\Rightarrow\; op(P_1, \ldots, P_n) \simeq op(P_1', \ldots, P_n')$$

Die Kongruenz beschreibt somit eine Eigenschaft, die für die Übertragung der Verhaltenserhaltung an einem Beobachtungspunkt auf das Gesamtsystem genutzt werden kann. Daher werden für eine lokale Verhaltenserhaltung in Beobachtungspunkten im weiteren nur Verhaltenserhaltungsbegriffe genutzt, die eine Kongruenz bezüglich der in der Semantik verwendeten Operatoren ist.

Verhaltenserhaltung	Äquivalenz	Kongruenz	Refinement-Erhaltend
Verhaltenserhaltung für CSP, CSP-OZ und CSP-Teil			
$\sqsubseteq_{\mathcal{T}}$	$-$	$+$ [1]	$-$
$\sqsubseteq_{\mathcal{F}}$	$-$	$+$ [1]	$-$
$\sqsubseteq_{\mathcal{FD}}$	$-$	$+$ [1]	$-$
$\equiv_{\mathcal{T}}$	$+$	$+$ [2]	$(\sqsubseteq_{\mathcal{T}})$
$\equiv_{\mathcal{F}}$	$+$	$+$ [2]	$(\sqsubseteq_{\mathcal{T}}, \sqsubseteq_{\mathcal{F}})$
$\equiv_{\mathcal{FD}}$	$+$	$+$ [2]	$(\sqsubseteq_{\mathcal{T}}, \sqsubseteq_{\mathcal{F}}, \sqsubseteq_{\mathcal{DF}})$
$\approx$	$+$	$+$	$(\sqsubseteq_{\mathcal{T}})$
$\sim$	$+$	$+$	$(\sqsubseteq_{\mathcal{T}})$
Verhaltenserhaltung für Object-Z, Z und Z-Teil			
$\sqsubseteq_{DS}$	$-$	$-$	$-$
$\equiv_{DS}$	$+$	$-$	$(\sqsubseteq_{DS}, \sqsubseteq)$
$\sqsubseteq_{iDS}$	$-$	$-$	$-$
$\equiv_{iDS}$	$+$	$-$	$(\equiv_{iDS}, \sqsubseteq_{DS}, \sqsubseteq)$
$\sqsubseteq$	$-$	$-$	$-$
$\equiv_{\sqsubseteq}$	$+$	$-$	$(\sqsubseteq)$
$\equiv_{[\![]\!]}$	$+$	$+$	$(\equiv_{iDS}, \sqsubseteq_{DS}, \sqsubseteq)$

[1] [Ros97]
[2] Folgt aus der Kongruenz der entsprechenden Refinements.

Tabelle 7.3: Übersicht über einige Eigenschaften verschiedener Verhaltenserhaltungsbegriffe

Refinement-Erhaltung Die formale Methode CSP-OZ wird häufig im Zusammenhang mit der Technik des Refinements verwendet. Ein Refactoring sollte daher eine Refinementbeziehung zwischen zwei Modellen nicht zerstören.

Definition 34 (Refinement-Erhaltung) *Eine Verhaltenserhaltung* $\simeq$ *ist* refinementerhaltend *für ein Refinement* $\sqsubseteq_X$*, wenn für Modelle A und B sowie deren Refinements* A' *und* B' *gilt:*

$$A \sqsubseteq_X B \land A \simeq A' \land B \simeq B' \Rightarrow A' \sqsubseteq_X B'$$

Eine von einem Refinement abgeleitete Äquivalenz ist verhaltenserhaltend für das zugrunde liegende Refinement:

Satz 34.1 *Sei* $\sqsubseteq_X$ *ein Refinement und* $\equiv_X$ *eine Äquivalenz, die vom Refinement* $\sqsubseteq_X$ *abgeleitet ist und es gelte:*

$$A \equiv_X B := A \sqsubseteq_X B \land B \sqsubseteq_X A.$$

Dann ist $\equiv_X$ *als Verhaltenserhaltung refinementerhaltend für das Refinement* $\sqsubseteq_X$*.*

Beweis:

$$\begin{aligned} & A \sqsubseteq_X B \land A \equiv_X A' \land B \equiv_X B' \\ \Rightarrow\ & A' \equiv_X A \sqsubseteq_X B \equiv_X B' \\ \Rightarrow\ & A' \sqsubseteq_X A \sqsubseteq_X B \sqsubseteq_X B' \\ \Rightarrow\ & A' \sqsubseteq_X B' \end{aligned}$$

□□

Im Zusammenhang mit CSP-OZ werden häufig drei Refinementbegriffe für das Gesamtsystem genutzt, die sich aus der gemeinsamen $\mathrm{CSP_Z}$ Semantik ergeben: Trace-, Failure und Failure-Divergences-Refinement. Dabei bildet das Gesamtsystem den äußersten möglichen Beobachtungspunkt. Das heißt, dass für alle Refactorings gelten muss, dass sich das Gesamtsystem unter Beachtung des Verhaltenserhaltungsbegriffes nicht verändern darf.

Im Folgenden werden Beobachtungspunkte für CSP-OZ analysiert. Dabei werden die Abhängigkeiten von den Verhaltenserhaltungsbegriffen betrachtet und festgelegt, welche Begriffe an welchem Punkt genutzt werden müssen.

CSP-OZ hat mehrere Sichten, die eine eigene lokale Semantik haben. Somit kann die lokale Semantik als Anknüpfung benutzt werden und man erhält als Beobachtungspunkte die CSP-Sicht und die Z-Sicht. Ein weiterer Punkt ergibt sich aus der Art und Weise wie die Object-Z-Klassen in CSP-OZ eingebettet sind. Somit bilden die CSP-OZ-Klassen einen weiteren Beobachtungspunkt. Innerhalb der Object-Z-Klassen und dem Z-Teil der CSP-OZ-Klassen sind die Operationen ebenfalls Beobachtungspunkte. Einen Überblick über die Beobachtungspunkte gibt Tabelle 7.4.

Nachdem wichtige Beobachtungspunkte für CSP-OZ bekannt sind, stellt sich die Frage nach den Verhaltenserhaltungsbegriffen für die Punkte. Da zwei verschiedene semantische Domänen verwendet werden ($\mathrm{CSP_Z}$ und Z), werden hier die Begriffe für beide semantische Domänen betrachtet. Die Auswahl der Verhaltenserhaltung ist von der Anwendungsdomäne abhängig. Innerhalb von CSP-OZ sind deshalb verschiedene Verhaltenserhaltungsbegriffe interessant, die alle auf die üblichen Refinementbegriffe $\sqsubseteq_{\mathcal{T}}$, $\sqsubseteq_{\mathcal{F}}$, und $\sqsubseteq_{\mathcal{FD}}$ für

Beobachtungs-punkt	Beschreibung	lokale Semantik	$\backsimeq_l$ für $\backsimeq_g$ [1]					
			$\sqsubseteq_{\mathcal{T}}$	$\sqsubseteq_{\mathcal{F}}$	$\sqsubseteq_{\mathcal{FD}}$	$\equiv_{\mathcal{T}}$	$\equiv_{\mathcal{F}}$	$\equiv_{\mathcal{FD}}$
Gesamtsystem	Oberste Ebene der Beobachtungspunkte. Hier sollte sich die Verhaltenserhaltung des Gesamtsystems beobachten lassen	$\mathrm{CSP_Z}$	$\sqsubseteq_{\mathcal{T}}$ [3]	$\sqsubseteq_{\mathcal{F}}$ [3]	$\sqsubseteq_{\mathcal{FD}}$ [3]	$\equiv_{\mathcal{T}}$ [3]	$\equiv_{\mathcal{F}}$ [3]	$\equiv_{\mathcal{FD}}$ [3]
Klassensystem	Eine Menge von CSP-OZ-Klassen, die über das System-CSP direkt miteinander interagieren. Kann im Gegensatz zum Gesamtsystem ein Teilausdruck des System-CSP sein.	$\mathrm{CSP_Z}$	$\sqsubseteq_{\mathcal{T}}$ [4]	$\sqsubseteq_{\mathcal{F}}$ [4]	$\sqsubseteq_{\mathcal{FD}}$ [4]	$\equiv_{\mathcal{T}}$ [4]	$\equiv_{\mathcal{F}}$ [4]	$\equiv_{\mathcal{FD}}$ [4]
CSP-OZ-Klasse	eine CSP-OZ-Klasse	CSP_Z	$\sqsubseteq_{\mathcal{T}}$ [4]	$\sqsubseteq_{\mathcal{F}}$ [4]	$\sqsubseteq_{\mathcal{FD}}$ [4]	$\equiv_{\mathcal{T}}$ [4]	$\equiv_{\mathcal{F}}$ [4]	$\equiv_{\mathcal{FD}}$ [4]
Object-Z-Klasse	eine Object-Z-Klasse	Z	$\equiv_{[\![]\!]}$ [5]	$\equiv_{[\![]\!]}$ [5]	$\equiv_{[\![]\!]}$ [5]	$\equiv_{[\![]\!]}$ [5]	$\equiv_{[\![]\!]}$ [5]	$\equiv_{[\![]\!]}$ [5]
CSP-Teil	Der CSP-Teil einer Klasse. Dabei wird nur der CSP-Teil einer Klasse betrachtet	CSP_Z	$\sqsubseteq_{\mathcal{T}}$ [4]	$\sqsubseteq_{\mathcal{F}}$ [4]	$\sqsubseteq_{\mathcal{FD}}$ [4]	$\equiv_{\mathcal{T}}$ [4]	$\equiv_{\mathcal{F}}$ [4]	$\equiv_{\mathcal{FD}}$ [4]
Z-Teil	Der Z-Teil einer CSP-OZ-Klasse. Auch hier wird nur der Z-Teil einer Klasse betrachtet	CSP_Z	$\sqsubseteq_{\mathcal{T}}$ [4]	$\sqsubseteq_{\mathcal{F}}$ [4]	$\sqsubseteq_{\mathcal{F}}$ [4]	$\equiv_{\mathcal{T}}$ [4]	$\equiv_{\mathcal{F}}$ [4]	$\equiv_{\mathcal{F}}$ [4]
Z-Teil	Der Z-Teil einer CSP-OZ-Klasse. Auch hier wird nur der Z-Teil einer Klasse betrachtet	Z	$\sqsubseteq_{iDS}$ [6]	$\sqsubseteq_{iDS}$ [6]	$\sqsubseteq_{iDS}$ [6]	$\equiv_{\sqsubseteq_{iDS}}$ [6]	$\equiv_{\sqsubseteq_{iDS}}$ [6]	$\equiv_{\sqsubseteq_{iDS}}$ [6]
Operation[2]	Eine einzelne Operation in einer CSP-OZ-Klasse	Z	$\sqsubseteq_{iDS}$ [6]	$\sqsubseteq_{iDS}$ [6]	$\sqsubseteq_{iDS}$ [6]	$\equiv_{\sqsubseteq_{iDS}}$ [6]	$\equiv_{\sqsubseteq_{iDS}}$ [6]	$\equiv_{\sqsubseteq_{iDS}}$ [6]

[1] Die in der Tabelle angegeben Verhaltenserhaltung stellt die im Header angegebene Verhaltenserhaltung sicher.
[2] Beim Beobachtungspunkt der Operationen muss wegen einiger Eigenschaften (fehlende Kompositionalität) des Refinement von Object-Z auch alle Operationen in geprüft werden, die auf die geänderte Operation aufbauen.
[3] Der Beobachtungspunkt ist das Gesamtsystem. Deshalb folgt die globale Verhaltenserhaltung trivialerweise aus der lokalen Verhaltenserhaltung.
[4] Die globale Verhaltenserhaltung folgt direkt aus der Kongruenz von CSP bzgl. aller CSP-Operatoren.
[5] Die CSP-Semantik von Object-Z benutzt die Semantikfunktionen von Object-Z. Wenn sich diese nicht verändern, was die Aussage der Semantikerhaltung ist, verändert sich die CSP-Semantik nicht. Mit der Kongruenz von CSP folgt die globale Verhaltenserhaltung.
[6] Obwohl $\sqsubseteq_{iDS}$ keine Kongruenz bzgl. der Z-Operatoren ist, folgt wegen Satz 34.2 zusammen mit der Kongruenz von CSP die Beobachtungspunkteigenschaft.

Tabelle 7.4: Beobachtungspunkte in CSP-OZ

CSP aufbauen. Für die Beobachtungspunkte werden für die verschiedenen Verhaltenserhaltungen von CSP-OZ passende lokale Verhaltenserhaltungen benötigt. In Tabelle 7.4 sind einige Möglichkeiten angegeben. Dabei ist in der Tabellenüberschrift angegeben, welche Verhaltenserhaltung für CSP-OZ erreicht werden soll ($\backsimeq_g$). In der Tabelle sind die schwächsten Verhaltenserhaltungen dieser Arbeit angegeben, die $\backsimeq_g$ implizieren. Die Hierarchie der Begriffe ist in Abbildung 7.1 dargestellt. Eine weiter oben stehende Verhaltenserhaltung impliziert die darunter mit einem Pfeil verbundenen.

Für jeden der einzelnen Beobachtungspunkte müssen die Beobachtungspunktseigenschaften gezeigt werden. In den meisten Fällen der lokalen Semantik $\mathrm{CSP_Z}$ ergibt sich dies als direkte Folgerung aus der Kongruenz-Eigenschaft. Dabei gibt es zwei Ausnahmen bei der CSP-Semantik des Z-Teils. Hier reicht eine Betrachtung der Failures, weil die CSP-Semantik des Z-Teils immer divergenzlos ist. Daher ist dort durch ein Failure-Refinement immer auch ein Failure-Divergences-Refinement gegeben.

Für die Betrachtung der Semantiken, die in Z gegeben sind, wird ein kleiner Trick angewandt. Wenn sich für einen Beobachtungspunkt zeigen lässt, dass dieser ein Beobachtungspunkt in Bezug auf einen anderen Beobachtungspunkt ist, so ist dieser auch ein Beobachtungspunkt für das Gesamtsystem. Es reicht daher nur einen Fall aus der Tabelle zu analysieren. Wenn für den CSP-Teil aus dem injektiven Data Refinement das Failure Refinement folgt, können alle anderen Fälle auf diesen zurückgeführt werden.

Satz 34.2 *(Beobachtungspunkte Z-Teil) Aus dem Beobachtungspunkt Z-Teil zusammen mit der injektiven downward Simulation $\sqsubseteq_{iDS}$ folgt der Beobachtungspunkt Z-Teil zusammen mit dem Failure Refinement $\sqsubseteq_{\mathcal{F}}$.*

Beweis: Der Beweis baut auf den Semantikfunktionen *procS* und *procO* auf, weil diese die Umsetzung von Z nach CSP vornehmen. Dazu seien zwei Z-Teile A und C, wobei C ein Refinement von A ist. Weil die Retrieve-Relation eine Injektion ist, ist die Menge der durch eine Operation erreichbaren Zustände in C kleiner oder gleich der von A. Des Weiteren, sind die Operationen in A und C in den gleichen Fällen ausführbar (Applicability). Da die Ausgaben bzw. Eingaben durch ein Data Refinement nicht verändert werden, können die Prozesse $procS_A$ und $procO_A$ immer die gleichen sichtbaren Events ausführen wie $procS_C$ und $procO_C$. Daher sind die Failuremengen immer gleich. Es muss nur noch die Initialisierung betrachtet werden. Da die Retrieve-Relation alle drei Bedingungen der Downward Simulation erfüllen muss, können nur Paare in ihr enthalten sein, bei denen alle Operationen in pre übereinstimmen. Zusammen mit dem Init ergibt dies, dass nach dem Init immer die gleichen Operationen nutzbar sind. Da dies zusammen mit *procS* und *procO* die Failures nach dem Init bestimmt, stimmen diese überein.

Insgesamt sind die Failuremengen an den Knoten gleich, was dann das Refinement impliziert. □□

Für CSP-OZ werden drei verschiedene Refinement Begriffe verwendet. Diese induzieren auch drei verschiedene Verhaltenserhaltungsbegriffe. Daher ist es interessant zu wissen, für welche Verhaltenserhaltung das jeweilige Refactoring gültig ist. Da die Verhaltenserhaltung und Refinements sich teilweise bedingen, wird aus der Erfüllung einiger Begriffe auf die Erfüllung anderer Begriffe gefolgert. Wenn z.B. ein Refactoring auf der System-Ebene eine $\mathcal{FD}$-Äquivalenz bildet, dann ist dieses Refactoring auch eine $\mathcal{F}$- und eine $\mathcal{T}$-Äquivalenz. Der Zusammenhang der Begriffe wird in Tabelle 7.2 gezeigt. Dabei schließen höher stehende Begriffe die unter ihnen angeordneten Begriffe mit ein. Daher werden Refactorings im

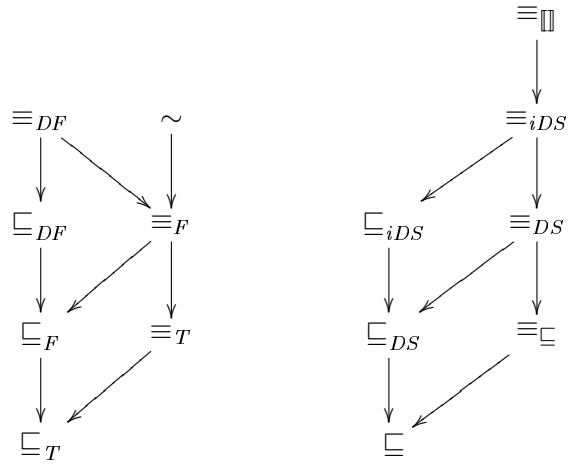


Abbildung 7.1: Hierarchie der Verhaltenserhaltungsbegriffe für CSP-OZ und Object-Z

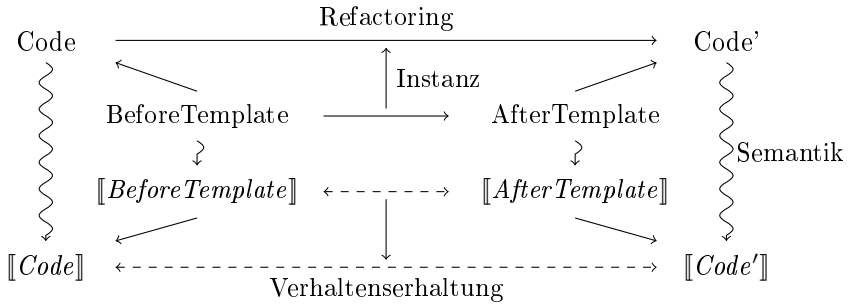


Abbildung 7.2: Zusammenhänge der Verhaltenserhaltung von Code und der Verhaltenserhaltung der Templates

weiteren auf der allgemeinsten Ebene, die möglich ist, bewiesen werden. Dies ist in der Regel die $\mathcal{FD}$-Äquivalenz.

7.3 Beweisbar korrekte Refactorings einer Sicht

Re$\mathcal{L}$ unterstützt durch sein Design die formale Analyse. Bevor in einem späteren Unterkapitel Beweise für mehrere Sichten betrachtet werden, sollen in diesem Unterkapitel die grundlegenden Beweistechniken anhand von Refactorings einer Sicht eingeführt werden. Dies entspricht auch dem, was für die meisten Sourcecode Refactorings in Tools benötigt wird, da Sourcecode normalerweise nur eine Sicht hat.

Die vorgestellten Beweistechniken sind von dem Erhaltungsbegriff unabhängig, solange sie und die Semantik einige Bedingungen erfüllen. Die Semantik muss kompositionell sein (vgl. Kapitel 2), d.h. dass sich alle (oder zumindest die im Beweis benötigten) semantischen Terme durch einen Ausdruck über das semantische Mapping beschreiben lassen. Als Beispiel dient wieder die Mathematik: Für zwei reelle Funktionen $f_1(x)$ und $f_2(x)$ lässt sich sicher die Semantik der Summe an einem Punkt $[\![f_1(x) + f_2(x)]\!]$ als Summe der Funktionssemantiken schreiben $[\![f_1(x)]\!] + [\![f_2(x)]\!]$.

Diese Eigenschaft erlaubt es, dass mittels der Semantik der Zielsprache den Templates die Menge der Semantiken der matchenden Codestücke gegeben werden kann. Ziel ist es, dann zu zeigen, dass die Semantik der Codestücke gleich ist. Einen Überblick über die Idee beinhaltet die Abbildung 7.2. Das äußere Quadrat stellt die Verhaltenserhaltung des Codes dar, das Innere die Verhaltenserhaltung der Templates. Der *Code* in der oberen linken Ecke wird durch ein Refactoring zu dem geänderten *Code′*. Da die Verhaltenserhaltung für das Refactoring als Relation auf der semantischen Domäne des Codes definiert ist, wird sowohl der *Code* als auch der geänderte *Code′* auf ihre Semantik abgebildet. Die Verhaltenserhaltung ist unten zwischen den Elementen der semantischen Domäne definiert. Das äußere Quadrat entspricht dem Vorgehen eines Beweises in der Postvalidierung. Das innere Quadrat beschreibt, wie die Beweisführung mit den Templates funktioniert. Im Grunde ist dieses Vorgehen das Gleiche wie bei Code. Die Templates werden auf die semantische Domäne abgebildet und anschließend wird gezeigt, dass das Verhalten erhalten wird. Der entscheidende Unterschied besteht darin, dass die Templates, da sie unvollständiger Code sind, nur partiell in die semantische Domäne überführt werden. Dies ist möglich, da gefordert wurde, dass die Semantik kompositionell ist. Das heißt, dass Teile des Codes in Semantik-Klammern stehen gelassen werden können. Die Verhaltenserhaltung wird dann über den ausgewerteten Teil der Semantik geführt. Die Pfeile zwischen den beiden Quadraten zeigen, wie aus dem inneren Quadrat die Gültigkeit des äußeren folgt. Die Templates können mit ihren Meta-Variablen als eine Menge von Code verstanden werden. Die Menge ist bestimmt durch alle gültigen Einsetzungen von Werten für die Meta-Variablen. Im Grunde gibt somit eine Before-/AfterTemplate Kombination alle möglichen Paare von Code an, welche mit dem Refactoring durchgeführt werden können. Somit überträgt sich die Refactoringrelation der Templates auf die Refactoringrelation des Codes, welches ein Element der ausgefalteten Relationsmenge der Templates ist. Genauso verhält es sich im unteren Teil der Abbildung. Mit dem Beweis in der nicht vollständig ausgewerteten Semantik werden Mengen der semantischen Domäne in Relation gesetzt. Die spezielle Semantik des refaktorisierten Codes ist somit eine Instanz dieser Relationsmenge.

Wie leicht zu sehen ist, wird es durch die Templates ermöglicht, Refactorings allgemein zu beweisen, so dass alle möglichen Instanzen eines Code-Refactorings durch den Beweis instantiiert werden können.

Im Folgenden soll ein Korrektheitsbeweis eines Refactorings im Detail besprochen werden. Als Beispiel soll das Refactoring genutzt werden, welches im letzten Kapitel erstellt wurde: `SplitOperationOZ`.

Satz 34.3 (`SplitOperationOZ`) *`SplitOperationOZ` ist ein Refactoring für Object-Z unter der Verhaltenserhaltung* $\equiv_{[\![]\!]}$.

Beweis: Bei diesem Refactoring kann die stärkste Art der Verhaltenserhaltung gezeigt werden: Die Gleichheit der Semantik des Codes vor und nach dem Refactoring. Das Semantische Mapping bildet die Klasse vor und nach dem Refactoring auf das gleiche Element der semantischen Domäne ab. Als Beobachtungspunkt wird daher die Object-Z-Klasse unter der Verhaltenserhaltung $\equiv_{[\![]\!]}$ genutzt.

$$
\begin{aligned}
[\![\texttt{className}_{BT}]\!]_{Object-Z} = \; & (\textbf{visible}(\texttt{className}_{BT}),\\
& \textbf{inherited}(\texttt{className}_{BT}),\\
& \textbf{state}(\texttt{className}_{BT}),\\
& \textbf{init}(\texttt{className}_{BT}),\\
& \{Op_k(\texttt{className}_{BT}\}_{k\in\{Op1,\ldots,Opm\}})\\
= \; & (\textbf{visible}(Class1_{AT}),\\
& \textbf{inherited}(\texttt{className}_{AT}),\\
& \textbf{state}(\texttt{className}_{AT}),\\
& \textbf{init}(\texttt{className}_{AT}),\\
& \{Op_k(\texttt{className}_{AT}\}_{k\in\{Op1,\ldots,Opm\}})\\
= [\![\texttt{className}_{AT}]\!]_{Object-Z}
\end{aligned}
$$

Dazu werden in dem folgenden Beweis die einzelnen Gleichheiten der einzelnen Hilfsfunktionen der Semantik gezeigt. In den Beweisen werden für die Konstrukte aus den Templates Meta-Variablen genutzt, die dem Index des Subtemplates entsprechen. Mit einem Suffix wird angegeben, ob es sich dabei um das Subtemplate aus dem Before- oder AfterTemplate handelt. So werden im folgenden die Meta-Variablen $\texttt{className}_{BT}$ für die Klasse aus dem BeforeTemplate und $\texttt{className}_{AT}$ für die Klasse aus dem AfterTemplate genutzt. Der Typ der Variablen entspricht dabei dem Scope des Subtemplates, hier ist dies `Class0`. Zur Verdeutlichung wird auch an einige Meta-Variablen ein BT bzw. AT annotiert. Dies hat dabei dokumentarische Gründe, da sich der Wert einer Meta-Variable nicht ändern kann. Die Annotation macht aber in einigen Fällen deutlich, wann die Variable im Kontext des Before- bzw. AfterTemplates betrachtet wird. Dies wird sofort bei der Äquivalenz für die ersten Hilfsfunktionen deutlich. Hier muss gezeigt werden, dass sich die Menge der sichtbaren Elemente der Klasse nicht verändert:

$$\textbf{visible}(\texttt{className}_{BT}) = \textbf{visible}(\texttt{className}_{AT})$$

Diese Äquivalenz gilt, wegen:

$$
\begin{aligned}
\textbf{visible}(\texttt{className}_{BT}) &= \textbf{visible}(\texttt{vlist}_{BT})\\
&= \textbf{visible}(\texttt{vlist}_{AT}) = \textbf{visible}(\texttt{className}_{AT})
\end{aligned}
$$

Die mittlere Gleichheit ist im Prinzip überflüssig, da es nur eine Meta-Variable *vlist* gibt, macht aber den Übergang zwischen den Templates innerhalb eines Beweises deutlich. In so einfachen Fällen wie hier wird meist auf diese zusätzliche Annotation verzichtet.

Hier wird auch der Nutzen der kompositionellen Auswertung der Semantik deutlich. Die Funktion **visible** muss nicht bis zu einem Wert ausgerechnet werden. Es reicht die Funktion soweit auszuwerten, dass ein Ausdruck vorliegt, in dem nur noch eine Meta-Variable auftaucht.

Damit ist auch die Äquivalenz für die erste semantische Hilfsfunktion gezeigt. Die Gleichheit beruht dabei darauf, dass sich die für die Funktion interessanten Teile der Spezifikation (hier die Visibility-Liste) nicht geändert haben. Der Beweisteil ist sehr einfach, weil die, von der Definition benötigten Konstrukte (hier Visibility-Liste) durch die gleiche Meta-Variable in den beiden Templates ausgedrückt wurde. Beim Beweis der nächsten Gleichheit ist dies nicht der Fall. Zu zeigen ist, dass die Menge der Superklassen gleich ist:

$$\textbf{inherited}(\texttt{className}_{BT}) = \textbf{inherited}(\texttt{className}_{AT})$$

Die Definition von **inherited** für Klassen benötigt die einzelnen Elemente der Auflistung der Oberklassen:

$$\mathbf{inherited}(A[X_1, \ldots, X_n]) = \bigcup_{i \in \{A_1, \ldots, A_n\}} \mathbf{inherited}(i) \cup \{i\}$$

An dieser Stelle werden die Analysefunktionen aus Re$\mathcal{L}$ eingesetzt. Diese müssen für die Beweise in einer formalen Form vorliegen, was für die Analysefunktionen von CSP-OZ und Object-Z gegeben ist (siehe Abschnitt 6.3). In der Formel oben stehen die A_x für die einzelnen aufgezählten Oberklassen in der Spezifikation.
Der Beweis ergibt sich dann als:

$$\begin{aligned}
&\mathbf{inherited}(\texttt{className}_{BT}) \\
&\qquad = \bigcup_{i \in \texttt{superclasses}_{BT}} \mathbf{inherited}(i) \cup \{i\} \qquad (7.1) \\
&\qquad = \bigcup_{i \in \texttt{superclasses}_{AT}} \mathbf{inherited}(i) \cup \{i\} \qquad (7.2) \\
&\qquad = \mathbf{inherited}(\texttt{className}_{AT})
\end{aligned}$$

In dem Beweis muss noch begründet werden, warum die Zeilen 7.1 und 7.2 wirklich äquivalent sind, wo eine Funktion (**inherited**) auf anderen Klassen aufgerufen wird. Da eine Vererbungsstruktur in CSP-OZ nicht zirkulär sein darf (eine Klasse kann nicht Oberklasse von sich selber sein), kann **inherited** nur auf anderen Klassen als `className` aufgerufen werden. Aus dem gleichen Grund ist es nicht möglich, dass **inherited** auf einer Unterklasse von `className` aufgerufen wird. Diese beiden Folgerungen stellen sicher, dass in dem Schritt nur unveränderte Klassen betrachtet werden. Diese Argumentation wird auch bei anderen Schritten an diesem Beweis benötigt. Sie wird dann als *SClass*-Argument referenziert.

Damit ist die zweite Gleichheit gezeigt. Bei dem Beweis ist zu sehen, dass es wichtig sein kann die Meta-Variablen Templates zuzuordnen. Wenn dies nicht geschieht, besteht die Gefahr, dass wichtige Argumentationsschritte übersehen werden.

Die Beweise der semantischen Funktionen **state** und **init** folgen den gleichen Prinzipien. Deshalb werden sie ohne weitere Erläuterung im Block aufgeführt. Dabei wird im Schritt von BT nach AT immer auch das Argument für die Ober- und Unterklassen angewendet.

$$\begin{aligned}
\mathbf{state}(\texttt{className}_{BT}) &= [\mathit{self} : \texttt{className}_{BT}[\texttt{genParam}_{BT}]] \\
&\quad \wedge \bigwedge_{i \in \mathbf{inherited}(\texttt{superClasses}_{BT})} (\mathbf{state}(i) \setminus \{\mathit{self}\}) \bullet (\bullet_{i \in \texttt{locDefs}_{BT}} \mathbf{state}(i)) \\
&\quad \bullet \mathbf{state}(\texttt{classState}_{BT}) \\
&= [\mathit{self} : \texttt{className}_{AT}[\texttt{genParam}_{AT}]] \\
&\quad \wedge \bigwedge_{i \in \mathbf{inherited}(\texttt{superClasses}_{AT})} (\mathbf{state}(i) \setminus \{\mathit{self}\}) \bullet (\bullet_{i \in \texttt{locDefs}_{AT}} \mathbf{state}(i)) \\
&\quad \bullet \mathbf{state}(\texttt{classState}_{AT}) \\
&= \mathbf{state}(\mathit{Class1}_{AT})
\end{aligned}$$

$$
\begin{aligned}
\mathbf{init}(\mathtt{className}_{BT}) &= \bigwedge_{i \in \mathbf{inherited}(\mathtt{superClasses}_{BT})} \mathbf{init}(i) \bullet \mathbf{init}(\mathtt{init}_{BT}) \\
&= \bigwedge_{i \in \mathbf{inherited}(\mathtt{superClasses}_{AT})} \mathbf{init}(i) \bullet \mathbf{init}(\mathtt{init}_{AT}) \\
&= \mathbf{init}(\mathtt{className}_{BT})
\end{aligned}
$$

Als letzten Schritt in dem Verhaltenserhaltungsbeweis kommt der Beweis der Gleichheit der Operationen. Im Rahmen des Gesamtbeweises ist dies der interessante Teil, weil sich die Operationen verändern. Auch aus Sicht der Beweistechniken ist dieser Teil bemerkenswert. In den Templates wird eine Menge von Operationen in verschiedene Teilmengen zerlegt, die dann getrennt behandelt werden. Dies erfordert für jede dieser Teilmengen von Operationen einen separaten Beweis. In diesem Refactoring gibt es die Operationen, die vor der Operation `op`, die refaktorisiert werden soll, stehen, die Operation `op` und die Operationen nachher. Die Operationen vor und nach der Operation können vom Beweis zusammengefasst werden, da diese sich nicht verändern.
Sei k eine Operation aus $\mathbf{names}(\mathtt{ops1}) \cap \mathbf{names}(\mathtt{ops2})$, dann gilt folgendes:

$$
\begin{aligned}
\mathtt{Op}_k(\mathtt{className}_{BT}) &= (\bigcup_{i \in B_{BT}} \mathit{first}(\mathtt{Op}_k(i)) \cup \mathbf{delta}(k), \bigwedge_{i \in B_{BT}} \mathit{second}(\mathtt{Op}_k(i)) \bullet \mathbf{schema}(k)) \\
&\quad \text{mit } \mathtt{superclasses}_{BT} \cap \mathrm{classdom}\mathtt{Op}_k = B_{BT} \\
&= (\bigcup_{i \in B_{AT}} \mathit{first}(\mathtt{Op}_k(i)) \cup \mathbf{delta}(k), \bigwedge_{i \in B_{AT}} \mathit{second}(\mathtt{Op}_k(i)) \bullet \mathbf{schema}(k)) \\
&\quad \text{mit } \mathtt{superclasses}_{AT} \cap \mathrm{classdom}\mathtt{Op}_k = B_{AT} \\
&= \mathtt{Op}_k(\mathtt{className}_{AT})
\end{aligned}
$$

Grob ist die Menge B als die Menge aller Oberklassen, die auch die Operation `op` definieren, vorstellbar. Auch hier werden wieder Definitionen der Oberklassen benutzt. Diese sind wieder mit dem Argument *SClass* zu begründen.

Bleibt der Fall der geänderten Operation:

$$
\begin{aligned}
\mathtt{Op}_{\mathtt{op}_{BT}}(\mathtt{className}_{BT}) = (&\bigcup_{i \in B} \mathit{first}(\mathtt{Op}_{\mathtt{op}_{BT}}(i)) \cup \mathbf{delta}(\mathtt{op}_{BT} \mathrel{\widehat{=}} [\Delta(\mathtt{dlist})\mathtt{v} \mid \mathtt{p1};\ \mathtt{p2}]), \\
&\bigwedge_{i \in B} \mathit{second}(\mathtt{Op}_{\mathtt{op}_{BT}}(i)) \bullet \mathbf{schema}(\mathtt{op}_{BT} \mathrel{\widehat{=}} [\Delta(\mathtt{dlist})\mathtt{v} \mid \mathtt{p1};\ \mathtt{p2}])) \\
&\text{mit } \mathtt{superclasses}_{BT} \cap \mathrm{classdom}\mathtt{Op}_{\mathtt{op}_{BT}} = B_{BT}
\end{aligned}
$$

Da sich die Operation hier verändert, müssen die semantischen Funktionen weiter ausgewertet werden.

$$
\begin{aligned}
= \ & (\bigcup_{i \in B} \mathit{first}(\mathtt{Op}_{\mathtt{op}_{BT}}(i)) \cup \mathbf{delta}(\mathtt{dlist}), \\
& \bigwedge_{i \in B} \mathit{second}(\mathtt{Op}_{\mathtt{op}_{BT}}(i)) \bullet \mathbf{state}(\mathtt{className}_{BT})) \bullet \mathbf{schema}([\Delta(\mathtt{dlist})\mathtt{v} \mid \mathtt{p1};\ \mathtt{p2}]) \\
& \text{mit } \mathtt{superclasses}_{BT} \cap \mathrm{classdom}\mathtt{Op}_{\mathtt{op}_{BT}} = B
\end{aligned}
$$

Der nächste Schritt scheint auf den ersten Blick Informationen zu verlieren, da die Delta-Liste aus dem Schema eliminiert wird. Da aber eine Operation in der Semantik als Tupel einer erweiterten Delta-Liste und dem Schema dargestellt wird, ist die Delta-Liste in gewisser Weise doppelt vorhanden.

$$
\begin{aligned}
= \quad & \Bigg(\bigcup_{i \in B} \mathit{first}(\mathsf{Op}_{\mathsf{op}_{BT}}(i)) \cup \mathbf{delta}(\mathtt{dlist}), \\
& \bigwedge_{i \in B} \mathit{second}(\mathsf{Op}_{\mathsf{op}_{BT}}(i)) \bullet \mathbf{state}(\mathtt{className}_{BT}) \bullet [\mathtt{v} \mid \mathtt{p1};\ \mathtt{p2}]) \\
& \text{mit } \mathtt{superclasses}_{BT} \cap \mathrm{classdom}\mathsf{Op}_{\mathsf{op}_{BT}} = B \\
= \quad & (\bigcup_{i \in B} \mathit{first}(\mathsf{Op}_{\mathsf{op}_{BT}}(i)) \cup \mathbf{delta}(\mathtt{dlist1}) \cup \mathbf{delta}(\mathtt{dlist2}), \\
& \bigwedge_{i \in B} \mathit{second}(\mathsf{Op}_{\mathsf{op}_{BT}}(i)) \bullet \mathbf{state}(\mathtt{className}_{BT}) \bullet [\mathtt{v1} \mid \mathtt{p1}] \\
& \qquad\qquad \wedge \mathbf{state}(\mathtt{className}_{BT}) \bullet [\mathtt{v2} \mid \mathtt{p2}] \Bigg) \\
& \text{mit } \mathtt{superclasses}_{BT} \cap \mathrm{classdom}\mathsf{Op}_{\mathsf{op}_{BT}} = B \\
= \quad & \Bigg(\bigcup_{i \in B} \mathit{first}(\mathsf{Op}_{\mathsf{op}_{BT}}(i)) \cup (\mathbf{delta}(\mathtt{dlist1}) \cup \mathbf{delta}(\mathtt{dlist2})), \\
& \bigwedge_{i \in B} \mathit{second}(\mathsf{Op}_{\mathsf{op}_{BT}}(i)) \bullet \mathbf{schema}(\mathtt{op1}_{BT} \mathrel{\widehat{=}} [\Delta(\mathtt{dlist1})\ \ \mathtt{v1} \mid \mathtt{p1}])) \\
& \qquad\qquad \wedge \mathbf{schema}(\mathtt{op2}_{BT} \mathrel{\widehat{=}} [\Delta(\mathtt{dlist2})\ \ \mathtt{v2} \mid \mathtt{p2}])) \Bigg) \\
& \text{mit } \mathtt{superclasses}_{BT} \cap \mathrm{classdom}\mathsf{Op}_{\mathsf{op}_{BT}} = B
\end{aligned}
$$

Es erfolgt der Übergang zum AfterTemplate und die semantischen Hilfsfunktionen werden nach außen gezogen:

$$
\begin{aligned}
= \quad & \Bigg(\bigcup_{i \in B} \mathit{first}(\mathsf{Op}_{\mathsf{op}_{AT}}(i)) \cup (\mathbf{delta}(\mathtt{dlist1}) \cup \mathbf{delta}(\mathtt{dlist2})), \\
& \bigwedge_{i \in B} \mathit{second}(\mathsf{Op}_{\mathsf{op}_{AT}}(i)) \bullet \mathbf{schema}(\mathtt{op1}_{AT} \mathrel{\widehat{=}} [\Delta(\mathtt{dlist1})\ \ \mathtt{v1} \mid \mathtt{p1}])) \\
& \qquad\qquad \wedge \mathbf{schema}(\mathtt{op2}_{AT} \mathrel{\widehat{=}} [\Delta(\mathtt{dlist2})\ \ \mathtt{v2} \mid \mathtt{p2}])) \Bigg) \\
& \text{mit } \mathtt{superclasses}_{AT} \cap \mathrm{classdom}\mathsf{Op}_{\mathsf{op}_{AT}} = B \\
= \quad & \mathsf{Op}_{\mathsf{op}_{AT}}(\mathtt{className}_{AT})
\end{aligned}
$$

Damit ist die Verhaltenserhaltung der Operation in der Klasse gezeigt. Wieder muss, wie bei der Funktion **inherited**, für die Ober- und Unterklassen argumentiert werden, dass auch diese unverändert sind (*SClass*). □□

In dem Beweis der Verhaltenserhaltung des Refactorings `SplitOperationOZ` sind die wichtigen Techniken für einen Verhaltenserhaltungsbeweis eines in Re$\mathcal{L}$ beschriebenen Refac-

torings für eine BNF-Sprache, die eine kompositionellen Semantik besitzt, beschrieben worden. Bevor Refactorings mehrerer Sichten und deren Korrektheitsbeweise betrachtet werden, sollen diese Techniken nochmals zusammengefasst werden.

Partielle Auswertung von kompositionellen Semantiken In den Beweisen wird ausgenutzt, dass die Semantik kompositionell ist. Somit ist es möglich, vom Inhalt der Meta-Variablen zu abstrahieren, in dem der Inhalt der Meta-Variablen auf deren Semantik reduziert wird. Da sich der Wert einer Meta-Variablen in Re$\mathcal{L}$ nicht ändern kann, ist damit auch die Verhaltenserhaltung innerhalb der Variablen gegeben.

Klare Unterscheidung von BeforeTemplate und AfterTemplate Durch die explizite Angabe, ob das Auftreten der Meta-Variable dem Before- bzw. AfterTemplate zuzuordnen ist, wird der Übergang zwischen den Templates klar.

Betrachtung nicht sichtbarer (indirekter) Änderungen In dem Beweis wurde regelmäßig argumentiert, dass sich Klassenelemente nicht in den Ober- bzw. Unterklassen ändern. Das ist ein typischer Bestandteil von Beweisen von Refactorings. In der Objekt-Orientierung ist durch die Klassenhierarchie eine implizite Änderung von Unterklassen üblich, wenn sich eine Klasse verändert. In CSP-OZ tritt zusätzlich der Effekt auf, dass sich durch die Trennung der Typ- und Vererbungsstruktur Typdefinitionen bilden lassen, von denen die Typen von Unterklassen abhängen können. Somit ist es in dem Beweis wichtig die Oberklassen zu betrachten.

7.4 Refactorings mehrerer Sichten am Bsp. CSP-OZ

Ein zentrales Thema dieser Dissertation sind Refactorings von formalen Modellen mit mehreren Sichten. Bisher wurde in dieser Arbeit die Beschreibung eines Refactorings und einiger Beweistechniken im Rahmen der Beschreibungssprache Re$\mathcal{L}$ betrachtet. Wie im vorherigen Unterkapitel zu sehen war, sind Beweise von Refactorings, wenn eine geeignete Beschreibung vorliegt, nicht allzu kompliziert. In den nächsten Abschnitten sollen formale Refactorings mit mehreren Sichten einschließlich dafür notwendiger Beweistechniken betrachtet werden. Anders als bei Refactorings einer Sicht sind die hier betrachteten Refactorings nicht direkt von Programm- bzw. Code-Refactorings hergeleitet. Programmiersprachen beinhalten implizit alle benötigten Informationen in einem geschlossenen Code. Ein Beispiel für eine Modell-Sprache mit mehreren Sichten ist UML. Es gibt verschiedene Diagramme, die das Modell aus unterschiedlichen Blickwinkeln betrachten. So ist in einem Klassendiagramm die Struktur der Daten gut zu sehen. Ein Sequenzdiagramm dagegen zeigt die Interaktion von Objekten in isolierten Abläufen und Statediagramme beschreiben die innere Zustandsabfolge in einer Klasse. Die Herausforderung ist Refactorings zu definieren, die mehrsichtige Modelle beweisbar verhaltenserhaltend transformieren. Im folgenden wird wieder CSP-OZ als Beispiel-Zielsprache genutzt. CSP-OZ hat drei Sichten (System, Klassenzustand und Klassenablauf). Durch die Beschränkung auf drei Sichten werden die Beispiele einfacher, weil nicht wie in der UML elf verschiedene Diagrammtypen, die hier den Sichten entsprechen, betrachtet werden müssen. Auch hat CSP-OZ eine durchgehende formale Semantik, die sich in der UML noch in der Entstehung befindet [EW01, FS07, KGKK02, Szl06].

7.4.1 Beschreibung mehrsichtiger Refactorings mit ReℒL

Für Modelle mit mehreren Sichten wird eine geeignete Beschreibung benötigt, die die verschiedenen Sichten erfassen und deren Veränderungen beschreiben kann. Im Weiteren soll dafür weiter ReℒL benutzt werden. In diesem Abschnitt werden die Beschreibungstechnik von ReℒL aus Sicht von mehrsichtigen Refactorings betrachtet und gezeigt, wie ReℒL mehrsichtige Refactorings unterstützt.

Das Problem einer Refactoringbeschreibung für mehrere Sichten ist, dass die parallele Veränderung von mehreren Sichten beschrieben werden können muss. Durch diese Eigenschaft ist abgedeckt, dass Bedingungen an Sichten gestellt werden können. In ReℒL ist dies möglich, da verschiedene Subtemplates für die verschiedenen Sichten nutzbar sind. Für jede das Refactoring betreffende Sicht können ein oder mehrere Subtemplates definiert werden. Damit diese Subtemplates gewinnbringend genutzt werden können, muss darauf geachtet werden, dass das Gesamtmodell alle Sichten in einer BNF (also dem Meta-Modell) beschreibt.

Ein Beispiel für eine solche Beschreibung ist das Refactoring `SplitClassCSPOZ`. In diesem Refactoring wird eine Klasse, die inhaltlich nicht zusammenhängend ist, in zwei Klassen geteilt, die dann in der Systemsicht als Kombination die alte CSP-OZ-Klasse ersetzen. In Listing 7.1 ist die Umsetzung zu sehen. Bevor das Refactoring in der ReℒL-Beschreibung betrachtet wird, soll es kurz anhand eines Beispiels erläutert werden. Die folgende Klasse soll in zwei Klassen gespalten werden. Die Klasse *ClassToSplit* beinhaltet zwei Variablen a und b vom Typ $\mathbb{N}$. Diese können durch entsprechende Operationen auf einen größeren Wert (aUp, bUp) bzw. auf kleinere Werte ($aDown, bDown$) gesetzt werden. Der CSP-Teil erlaubt für die Operation auf der Variablen B eine beliebige Reihenfolge, wogegen die Variable A erst zweimal erhöht werden muss, bevor sie erniedrigt werden kann.

ClassToSplit

$\mathsf{chan}\ aUp, bUp, aDown, bDown$

$$\begin{aligned} main &= procA \,|||\, procB \\ procA &= aUp \rightarrow aUp \rightarrow aDown \rightarrow procA \\ procB &= bUp \rightarrow procB \sqcap bDown \rightarrow procB \end{aligned}$$

$a, b : \mathbb{N}$

INIT

$a < 10$

$b = 0$

aUp

$\Delta(a)$

$a' > a$

aDown

$\Delta(a)$

$a' < a$

bUp

$\Delta(b)$

$b' = b + 1$

bDown

$\Delta(b)$

$b' = b - 1$

Das zugehörige System-CSP ist einfach der Aufruf der Klassen:

$$System \stackrel{c}{=} ClassToSplit$$

Der Inhalt der Klasse zerfällt in zwei unabhängige Bereiche. Diese sollten getrennt wer-

den, um die Trennung im Design deutlich zu machen. Dazu kann das Refactoring `SplitClassCSPOZ` (Listing 7.1) genutzt werden. Dieses Refactoring besitzt vier Parameter:

- `class` : Name der Klasse, die gespalten werden soll,
- `classNeuA`, `classNeuB` : Namen der neuen Klassen,
- `operationsA`: Liste der Operationen, die in die Klasse mit Namen `classNeuA` verlagert werden sollen.

Die Parameter sollen zu

$$\begin{array}{ll} \texttt{class} & = ClassToSplit \\ \texttt{classNeuA} & = ClassA \\ \texttt{classNeuB} & = ClassB \\ \texttt{operationsA} & = \{aUp, aDown\} \end{array}$$

gewählt werden. Das Ergebnis des Refactorings ist:

$$System \stackrel{c}{=} ClassA \underset{\{init\}}{\parallel} ClassB$$

ClassA

$\textbf{chan}\, aUp, bUp, aDown, bDown$

$$\begin{array}{lcl} main & = & procA \\ procA & = & aUp \rightarrow aUp \rightarrow aDown \rightarrow procA \end{array}$$

	Init
$a : \mathbb{N}$	$a < 10$

aUp	*aDown*
$\Delta(a)$	$\Delta(a)$
$a' > a$	$a' < a$

ClassB

$\textbf{chan}\, aUp, bUp$

$$\begin{array}{lcl} main & = & procB \\ procB & = & bUp \rightarrow procB \sqcap bDown \rightarrow procB \end{array}$$

	Init
$b : \mathbb{N}$	$b = 0$

bUp	*bDown*
$\Delta(b)$	$\Delta(b)$
$b' = b + 1$	$b' = b - 1$

Bei diesem Refactoring werden alle drei Sichten von CSP-OZ verändert. Für die verschiedenen Sichten sind im Refactoring Listing 7.1 zwei verschiedene Vorgehensweisen zu finden.

Da das System-CSP nicht in die Klasse integriert ist, kann für das System-CSP ein eigenes Subtemplate (Index:`classInSys`) verwendet werden. Ein bisschen schwieriger ist der Umgang mit dem CSP-Teil bzw. dem Z-Teil in der Klasse, da diese in einem Klassenschema verbunden sind. In diesem Fall wird keine Trennung durch die Subtemplates durchgeführt, sondern beide Sichten zusammen abgehandelt, als wenn es sich um eine Sicht handeln würde. Die Trennung der Sichten spielt erst beim Beweis der Verhaltenserhaltung eine Rolle.

Die Precondition stellt sicher, dass die Klasse eine innere Trennung besitzt, auf die das Refactoring aufbaut. Das Beispiel zeigt, dass die Operationen in zwei Gruppen aufteilbar sind, wenn sie disjunkte Variablenmengen nutzen. Die Calculation berechnet die Trennung, insbesondere die Bestandteile der neuen Interfaces, des States und der Inits.

7.4.2 Einteilung von mehrsichtigen Refactorings

Nachdem Refactorings mehrerer Sichten mit Re$\mathcal{L}$ beschreibbar sind, sollen in den nächsten Abschnitten Techniken für die Beweisführung der Verhaltenserhaltung betrachtet werden. Dabei werden verschiedene Refactorings betrachtet, die in verschiedene Gruppen eingeteilt werden. Diese Gruppen entsprechen den Grundtypen von Wechselwirkungen von Sichten, die in einem Refactoring auftreten können. Wenn zwei Sichten betrachtet werden, können dabei drei verschiedenen Arten auftreten:

Unabhängiges Refactoring einer Sicht Das Refactoring verändert *eine* Sicht und diese Veränderungen sind ohne eine Betrachtung anderer Sichten gültige Refactorings.

Refactoring *einer* Sicht mit Bedingung an die andere Sicht Das Refactoring verändert nur eine Sicht. Die Verhaltenserhaltung ergibt sich nicht aus der Sicht alleine. Nur wenn die andere Sicht einige Eigenschaften erfüllt, wird das Verhalten erhalten. Dies wird in Re$\mathcal{L}$ durch eine Bedingung an die zweite Sicht beschrieben, die erfüllt sein muss.

Refactoring beider Sichten zusammen sind Refactorings, bei denen beide Sichten simultan verändert werden.

Diese Einteilung spiegelt sich in den Beweisen wider. Die Beweise der ersten beiden Art, können in der Regel auf der lokalen Semantik der Sicht bzw. einer gemeinsamen Semantik der Sichten durchgeführt werden. Die „Refactorings beider Sichten zusammen“ müssen immer auf der globalen Semantik betrachtet werden.

Wenn mehr als zwei Sichten vorhanden sind, können sich die Fälle der zwei Sichten überlagern. Das heißt, dass Refactorings existieren, bei denen mehrere Sichten gleichzeitig geändert werden, Bedingungen an andere Sichten gestellt werden und wieder andere ohne Beeinflussung daneben stehen.

Diese Fälle lassen sich leicht auf den Fall von zwei Sichten zurückführen. In den nächsten Abschnitten sollen deshalb diese drei Grundfälle genauer betrachtet werden. Dabei wird der Einfluss der Gruppen auf die Beweise betrachtet werden.

```
Refactoring: SplitCSPOZClass
Parameter:
        class                : classname;                                          3
        classNeuA, classNeuB : classname;
        opa                  : Operations;
        statea               : State0
BeforeTemplate:
        formalPara           : FormalParameters;                                   8
        cspParams            : CSPParameters;
        interface            : Interface;
        inheritedClass       : InheritedClass;
        P,Q                  : ProcExpr;
        CSPDEfs              : CSPZ;                                               13
        state                : State0;
        ini                  : InitialState;
        op                   : Operation0;
        X                    : Set;
Index: classInSys Scope: ProcExpr1                                                 18
        class
Index: Class1 Scope: Class0
```

class(formalPara)?

interface

(inheritedClass)∗

$main \stackrel{c}{=} \mathtt{P} \underset{\mathtt{X}}{\|} \mathtt{Q}$

CSPdefs

(state)?

(init)?

(op)∗

```
Precondition;
        ∀ i ∈ op | name(i) ∈ names(opa) • (vars(i) ∩ vars(state)) ⊂ vars(statea)   23
        ∀ i ∈ op | name(i) ∉ names(opa) • (vars(i) ∩ vars(state)) ⊂ vars(state) \ vars(statea)
        ∃ inita, initb : InitialState |
                inita ⋀ initb = init •
                        vars(inita) ⊂ vars(statea)
                        ∧ vars(inita) ⊂ vars(state) \ vars(statea)                 28
        ∃ interfacea, interfaceb : Interface |
                channels(interfacea) ∪ channels(interfaceb) = channels(interface) •
                        names(channels(interfaceb)) ⊂ names(channels(opb) ∪ channels(P))
                        ∧ names(channels(interfacea)) ⊂ names(channels(opa) ∪ channels(Q))
                        ∧ channels(interfacea) ∩ channels(interfaceb) = ∅          33
```

Listing 7.1: Refactoring

Calculation:
Define:
inita, initb : init ;
interfacea, interfaceb : Interface;
stateb : State0;
opb : Operations;
SCHEMA:
inita $\bigwedge$ initb = init
vars(inita) $\subset$ **vars**(statea)
vars(initb) $\subset$ **vars**(state) $\setminus$ **vars**(statea)
channels(interfacea) $\cup$ **channels**(interfaceb) = **channels**(interface)
channels(interfacea) $\cap$ **channels**(interfaceb) = $\varnothing$
names(**channels**(interfaceb)) $\subset$ **names**(**channels**(opb) $\cup$ **channels**(P))
names(**channels**(interfacea)) $\subset$ **names**(**channels**(opa) $\cup$ **channels**(Q))
opa $\cup$ opb = op
opa $\cap$ opb = $\varnothing$
statea $\bigwedge$ stateb = state
vars(statea) $\cap$ **vars**(stateb) = $\varnothing$
DONE
AfterTemplate:
Index: classInSys
(classNeuA $\underset{X \cup \{init\}}{\|}$ classNeuB)
Index: Class1 **Scope:** Class0

classNeuA(formalPara)?
interfaceA
(inheritedClass)$*$
$main \stackrel{c}{=}$ P
CSPdefs
(statea)?
(inita)?
(opa)$*$

classNeuB(formalPara)?
interfaceB
(inheritedClass)$*$
$main \stackrel{c}{=}$ Q
CSPdefs
(stateb)?
(initb)?
(opb)$*$

Listing 7.2: Fortsetzung: Refactoring

7.4.3 Unabhängige Refactorings einer Sicht

Die erste Art von Refactorings in einer kombinierten formalen Methode ist das unabhängige Refactoring einer Sicht. Dies ist die einfachste Gruppe, weil die Refactorings nur Teile der Spezifikation ändern, die in einer Sicht definiert sind und keine Auswirkungen auf andere Sichten haben. Das Hauptaugenmerk in diesem Kapitel wird deshalb auf die Definition von *Unabhängigen Refactorings einer Sicht* fallen, die Ergebnisse für die Refactorings sind dann direkte Folgen dieser Definition.

Oberflächlich betrachtet gesagt ist ein Refactoring unabhängig, wenn sich ein Refactoring auf eine Sicht beschränkt. Die Änderung in einer Sicht führt zu keiner Änderung in einer anderen Sicht.

Für Refactorings von mehrsichtigen Modellen wird ein konsistentes Modell vorausgesetzt (vgl. Kapitel 3). Ein Refactoring ist nicht geeignet Fehler in einem Modell zu korrigieren. Deshalb ist die Voraussetzung eines konsistenten Modells keine Einschränkung.

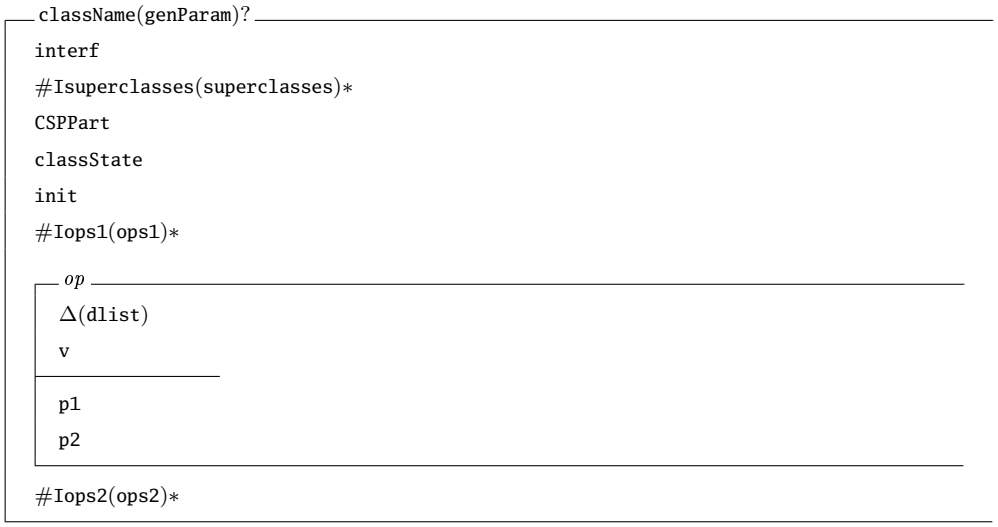
Definition 35 (unabhängiges Refactoring einer Sicht)
Sei $P = (P_1, \dots P_{imax})$ *eine Sprache* P *mit mehreren Sichten* $P_i = (A_k, L_k, D_k, [\![.]\!]_k)$. *Des Weiteren sei* $\bar{p} = (\bar{p_1}, \dots \bar{p}_{imax})$ *eine Instanz der Sprache. Ein Refactoring* $(pcon, T, \backsimeq)_P$ *heißt* unabhängiges Refactoring einer Sicht p_i, *wenn*

1. $\text{impact}(pcon) \subset \{P_i\}$ *(Vorbedingung stellt nur Bedingungen an die Sicht* P_i*),*
2. $\text{impact}(T) \subset \{P_i\}$ *(Nur die Sicht* P_i *wird verändert),*
3. $pcon \wedge [\![\bar{p}_i]\!]_i \backsimeq_l [\![T(\bar{p}_i)]\!]_i \Rightarrow [\![\bar{p}]\!] \backsimeq [\![(\bar{p}_1, \dots, t(\bar{p}_i), \dots, \bar{p}_{imax}]\!]$ *(Die Verhaltenserhaltung des Gesamtmodells folgt aus der Verhaltenserhaltung der Sicht) für einen lokalen Verhaltenserhaltungsbegriff* $\backsimeq_l$: $D_i \times D_i$.

Wie schon aus der Definition erahnt werden kann, ist es möglich die Beweise der unabhängigen Refactorings einer Sicht auf die lokale Semantik zu beschränken. Dies vereinfacht die Beweise oft, da nur die lokale Semantik der Sichten geprüft werden muss. Die Betrachtung der anderen Sichten kann dann entfallen. Als Beispiel für ein solches Refactoring soll `SplitOperationCSPOZ` dienen. Das Refactoring ist ähnlich dem Refactoring `SplitOperationOZ` aus Abschnitt 7.3 aufgebaut. Anders als beim Object-Z Refactoring wird das Split hier nicht in zwei Operationen sondern durch zwei Inline-Schema durchgeführt. Dieses vereinfacht das Refactoring, da die Schnittstelle nicht angepasst werden muss, um das Verhalten zu erhalten. Dadurch ist es möglich sich auf die interessanten Teile zu beschränken.

```
Refactoring: SplitOperationCSPOZ
Parameter:
        op, op1, op2 : Word;
        className    : ClassName;                                          4
        p1           : PredicateList;
BeforeTemplate:
        Define:
        v            : Declaration;
        p2           : PredicateList;                                      9
        dlist        : DeltaList;
        interf       : Interface;
        superclasses : InheritedClass;
        classState   : State0;
        init         : InitialState;                                       14
        ops1, ops2   : Operations;
        genParam     : FormalParameters;
        cspPart      : CSPZ;
```

Index: Class1 **Scope:** Class0



Precondition:

$\forall\, operation : \mathtt{ops1} \cup \mathtt{ops2} \bullet \mathbf{name}(operation) \neq \mathbf{name}(\mathtt{op1}) \land \mathbf{name}(operation) \neq \mathbf{name}(\mathtt{op1})$

Calculation:

Define:

```
dlist1, dlist2 : DeltaList;
v1,v2          : Declaration;
```

SCHEMA:

$\mathbf{vars}(\mathtt{dlist1}) \cup \mathbf{vars}(\mathtt{dlist2}) = \mathbf{vars}(\mathtt{dlist})$

$\mathbf{vars}(\mathtt{p1}) \subset \mathbf{vars}(\mathtt{dlist1}) \cup \mathbf{vars}(\mathtt{v1})$

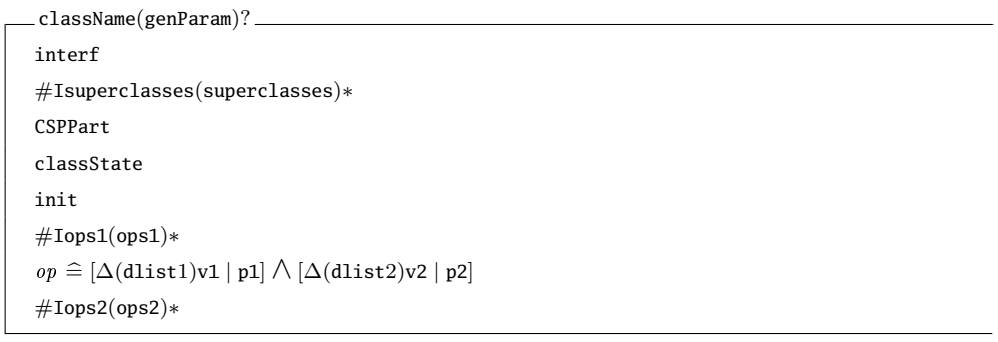
$\mathbf{vars}(\mathtt{p2}) \subset \mathbf{vars}(\mathtt{dlist2}) \cup \mathbf{vars}(\mathtt{v2})$

$\mathbf{vars}(\mathtt{v1}) \cup \mathbf{vars}(\mathtt{v2}) = \mathbf{vars}(\mathtt{v})$

DONE

AfterTemplate:

Index: Class1



Satz 35.1 (SplitOperationCSPOZ) *SplitOperationCSPOZ ist ein Refactoring für CSP-OZ unter der Verhaltenserhaltung* $\equiv_{\mathcal{FD}}$.

Beweis: Die Verhaltenserhaltung wird mit dem Beobachtungspunkt Z-Teil unter der Verhaltenserhaltung $\equiv_{[\![]\!]}$ gezeigt. Der Beweis ist ähnlich dem Beweis für eine Sicht aufgebaut. Für die meisten Teile der Semantik kann die Gleichheit einfach gezeigt werden. Der interessante (geänderte) Teil der Spezifikation ist vollständig innerhalb des Z-Teils der CSP-OZ-Klasse gekapselt. Hier soll sich daher auf die geänderte Operation beschränkt werden.

$$
\begin{aligned}
\mathsf{Op}_{\mathsf{op}}(\mathtt{className}) &= (\bigcup_{i\in B} \mathit{first}(\mathsf{Op}_{\mathsf{op}}(i)) \cup \mathbf{delta}(op \mathrel{\widehat=} [\Delta(\mathtt{dlist})v \mid \mathtt{p1};\ \mathtt{p2}]),\\
&\quad \bigwedge_{i\in B} \mathit{second}(\mathsf{Op}_{\mathsf{op}}(i)) \bullet \mathbf{schema}(op \mathrel{\widehat=} [\Delta(\mathtt{dlist})v \mid \mathtt{p1};\ \mathtt{p2}]))\\
&\quad \text{mit } \mathtt{superclasses}_{BT} \cap \mathrm{classdom}\mathsf{Op}_{\mathsf{op}} = B_{BT}\\
&= (\bigcup_{i\in B} \mathit{first}(\mathsf{Op}_{\mathsf{op}}(i)) \cup \mathbf{delta}(\mathtt{dlist})),\\
&\quad \bigwedge_{i\in B} \mathit{second}(\mathsf{Op}_{\mathsf{op}}(i)) \bullet \mathbf{state}(\mathtt{className}_{BT}) \bullet\\
&\qquad \mathbf{schema}([\Delta(\mathtt{dlist})v \mid \mathtt{p1};\ \mathtt{p2}])\\
&\quad \text{mit } \mathtt{superclasses}_{BT} \cap \mathrm{classdom}\mathsf{Op}_{\mathsf{op}} = B\\
&= (\bigcup_{i\in B} \mathit{first}(\mathsf{Op}_{\mathsf{op}}(i)) \cup \mathbf{delta}(\mathtt{dlist})),\\
&\quad \bigwedge_{i\in B} \mathit{second}(\mathsf{Op}_{\mathsf{op}}(i)) \bullet \mathbf{state}(\mathtt{className}_{BT}) \bullet [v \mid \mathtt{p1};\ \mathtt{p2}]\\
&\quad \text{mit } \mathtt{superclasses}_{BT} \cap \mathrm{classdom}\mathsf{Op}_{\mathsf{op}} = B\\
&= (\bigcup_{i\in B} \mathit{first}(\mathsf{Op}_{\mathsf{op}}(i)) \cup \mathbf{delta}(\mathtt{dlist1}) \cup \mathbf{delta}(\mathtt{dlist2})),\\
&\quad \bigwedge_{i\in B} \mathit{second}(\mathsf{Op}_{\mathsf{op}}(i)) \bullet \mathbf{state}(\mathtt{className}_{BT}) \bullet [\mathtt{v1} \mid \mathtt{p1}])\\
&\qquad \wedge \mathbf{state}(\mathtt{className}_{BT}) \bullet [\mathtt{v2} \mid \mathtt{p2}])\\
&\quad \text{mit } \mathtt{superclasses}_{BT} \cap \mathrm{classdom}\mathsf{Op}_{\mathsf{op}} = B\\
&= \Big(\bigcup_{i\in B} \mathit{first}(\mathsf{Op}_{\mathsf{op}}(i)) \cup (\mathbf{delta}(\mathtt{dlist1}) \cup \mathbf{delta}(\mathtt{dlist2}))),\\
&\quad \bigwedge_{i\in B} \mathit{second}(\mathsf{Op}_{\mathsf{op}}(i)) \bullet \mathbf{schema}(o\mathtt{p1} \mathrel{\widehat=} [\Delta(\mathtt{dlist1})\mathtt{v1} \mid \mathtt{p1}]))\\
&\quad \wedge \mathbf{schema}(o\mathtt{p1} \mathrel{\widehat=} [\Delta(\mathtt{dlist2})\mathtt{v2} \mid \mathtt{p2}]))\Big)\\
&\quad \text{mit } \mathtt{superclasses}_{BT} \cap \mathrm{classdom}\mathsf{Op}_{\mathsf{op}} = B\\
&= \Big(\bigcup_{i\in B} \mathit{first}(\mathsf{Op}_{\mathsf{op}}(i)) \cup (\mathbf{delta}(\mathtt{dlist1}) \cup \mathbf{delta}(\mathtt{dlist2}))),\\
&\quad \bigwedge_{i\in B} \mathit{second}(\mathsf{Op}_{\mathsf{op}}(i)) \bullet \mathbf{schema}(o\mathtt{p1} \mathrel{\widehat=} [\Delta(\mathtt{dlist1})\mathtt{v1} \mid \mathtt{p1}]))\\
&\quad \wedge \mathbf{schema}(o\mathtt{p1} \mathrel{\widehat=} [\Delta(\mathtt{dlist2})\mathtt{v2} \mid \mathtt{p2}]))\Big)\\
&\quad \text{mit } \mathbf{names}(\mathtt{superclasses}_{AT}) \cap \mathrm{classdom}\mathsf{Op}_{\mathsf{op}} = B\\
&= \mathsf{Op}_{\mathsf{op}}(\mathtt{className}_{AT})
\end{aligned}
$$

□

Es gleichen sich der Fall einer isolierten Sicht und eines Refactorings eines Modells mit nur einer Sicht stark. Der entscheidene Unterschied ist, dass im mehrsichtigen Fall auf die richtige lokale Semantik geachtet werden muss. Da in CSP-OZ zwei der drei Sichten in

die Klasse integriert sind, ist in `SplitOperationCSPOZ` auch eine zweite Sicht (CSP-Part) eingearbeitet, die unverändert übernommen wird.

7.4.4 Refactoring einer Sicht mit Bedingung an eine andere Sicht

Im vorigen Unterkapitel wurden Refactorings einer Sicht betrachtet, die unabhängig von anderen Sichten sind. Dort konnte der Beweis der Verhaltenserhaltung ohne Probleme direkt in der lokalen Semantik durchgeführt werden. In diesem Unterkapitel wird der Ansatz aus dem vorherigen verallgemeinert. Eine Sicht kann unter gewissen Umständen einem Refactoring unterzogen werden, wenn eine andere Sicht bestimmten Bedingungen genügt. Dies kann zwei Ursachen besitzen: Das Refactoring der Sicht ist nur unter zusätzlichen Bedingungen gültig, die von einer anderen Sicht sichergestellt werden können. Oder das Refactoring kann eine andere Sicht beeinflussen, dann ist es nur gültig, wenn die andere Sicht im gewissen Sinne gutmütig ist. Beide Fälle werden zusammen abgehandelt, da die Ursache für die zu nutzende Bedingung unterschiedlich sind, aber die Auswirkungen auf die Techniken gleich sind.

Für die Definition der Refactorings einer Sicht mit Bedingung an eine andere Sicht, wird der Begriff der gemeinsamen Semantik zweier Sichten benötigt:

Definition 36 (Gemeinsame Semantik von Sichten)
Sei $P = (P_1, \ldots, P_{imax})$ eine Sprache $(A, L, D, [\![.]\!])$ mit mehreren Sichten $P_i = (A_i, L_i, D_i, [\![.]\!]_i)$. Dann kombiniert eine Funktion $g : D_k \times D_j \to D_g$ in eine Menge D_g zwei lokale Semantiken D_k und D_j im Sinne der globalen Semantik $[\![.]\!]$, wenn es eine Funktion

$$h : (\underset{l \in \mathbb{N}_{\leq imax} \setminus \{i,k\}}{\times} D_l) \times D_g \to D$$

gibt, so dass

$$h([\![p_1]\!]_1, \ldots, [\![p_{k-1}]\!]_{k-1}, [\![p_{k+1}]\!]_{k+1}, \ldots \\ , [\![p_{j-1}]\!]_{j-1}, [\![p_{j+1}]\!]_{j+1}, \ldots, [\![p_{imax}]\!]_{imax}, g([\![p_k]\!]_{P_k}, [\![p_j]\!]_{P_j})) = [\![P]\!].$$

Dabei sind die p_i Instanzen der Sicht P_i. Das Paar (g, D_g) heißt dann gemeinsame Semantik der Sichten P_k und P_j.

Definition 37 (Refactoring einer Sicht mit Bedingung an eine andere Sicht)
Sei $P = (P_1, \ldots, P_{imax})$ eine Sprache $P = (A, L, D, [\![.]\!])$ mit mehreren Sichten P_i. Des Weiteren sei $\bar{P} = (\bar{p_1}, \ldots \bar{p}_{imax})$ eine Instanz der Sprache. Ein Refactoring $(pcon, T, \rightharpoonup)_P$ heißt Refactorings einer Sicht P_i unter Bedingung an eine andere Sicht P_j, *wenn es eine gemeinsame Semantik (g, D_g) der Sichten P_i und P_j gibt, so dass folgendes gilt:*

1. $\text{impact}(pcon) \subset \{P_i, P_j\}$ *(Vorbedingung stellt nur Bedingungen an die Sichten P_i und P_j),*

2. $\text{impact}(T) \subset \{P_i\}$ *(Nur die Sicht P_i wird verändert),*

3. $pcon \wedge g([\![\bar{p}_i]\!]_i, [\![\bar{p}_j]\!]_j) \rightharpoonup_l g([\![T(\bar{p}_i)]\!]_i, [\![\bar{p}_j]\!]_j) \Rightarrow [\![\bar{P}]\!] \rightharpoonup [\![\bar{p}_1, \ldots, T(\bar{p}_i), \ldots, \bar{p}_{imax}]\!]$ *(Die Verhaltenserhaltung des Gesamtmodells folgt aus der Verhaltenserhaltung der Sicht) für einen lokalen Verhaltenserhaltungsbegriff $\rightharpoonup_l$: $D_g \times D_g$.*

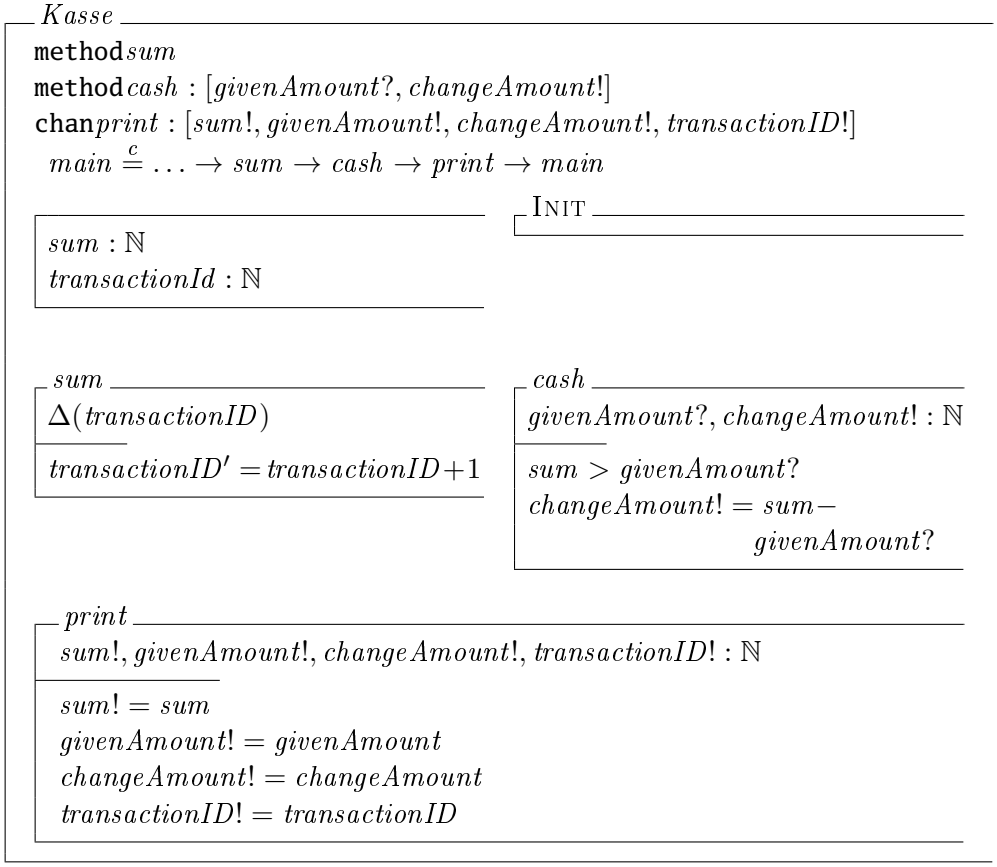


Abbildung 7.3: Beispielklasse für das Refactoring `MovePredicateToSuccedingOperation`

Der Hauptunterschied in dieser Definition im Vergleich zur Definition von unabhängigen Refactorings ist, dass die Vorbedingung *pcon* hier von zwei Sichten abhängig sein kann. Diese Definition geht im Bezug auf die Beobachtungspunkte einen Sonderweg. Die Verhaltenserhaltung wird über eine Funktion g definiert, die den Sichten eine gemeinsame Semantik gibt.

Dies ist häufig, jedoch nicht immer möglich. Wenn dies nicht möglich ist, müssen entsprechende Refactorings als Refactorings der dritten Gruppe (Refactorings mehrerer Sichten zusammen) betrachtet werden.

Ein Beispiel verdeutlicht das Vorgehen: Diesmal wird der Ausschnitt aus einem Kassensystem betrachtet, bei welchem, nachdem die Eingabe der Artikel beendet ist, der Betrag ausgegeben wird und anschließend die Geldmenge, die vom Kunden gegeben wird (Abbildung 7.3). Nachdem das Geld seinen Besitzer gewechselt hat, wird noch der Bon gedruckt, der eine Transaktions-ID enthält. Diese Transaktions-ID ist im Kassierprozess beim Summieren der Artikel gesetzt worden. Nun sollen die Prädikate, die die Transaktions-ID steuern, in die Operation *cash* verschoben werden. Dies ist nicht ohne Betrachtung des CSP-Teils möglich: Es kann sein, dass die Operation *cash* nicht ausgeführt wird, z.B. weil die Kasse eine Kartenzahlung erlaubt. Wenn der CSP-Teil betrachtet wird, ist zu sehen, dass *cash* immer nach *sum* ausgeführt wird und deshalb in diesem Fall das Verschieben

sicher ist. Insgesamt bedeutet dies, dass eine Bedingung an den CSP-Teil gestellt wird, damit das Refactoring in dem Z-Teil möglich ist: Die Operation *sum* muss immer von *cash* gefolgt werden.

Bei der Formulierung des Refactorings muss die Bedingung an die andere Sicht oder die anderen Sichten mit in das Template und die Precondition codiert werden. Im After-Template werden die entsprechenden Teile unverändert wiederholt, also der Code nicht geändert (siehe Listing 7.4.4).

```
Refactoring: MovePredicateToSuccedingOperation                          1
Parameter:
        class                   : classname;
        operation1, operation2  : Operation;
        schema3                 : OperationSchema
BeforeTemplate:                                                         6
    Define:
        v                       : Declaration;
        p2                      : PredicateList;
        dlist                   : DeltaList;
        interf                  : Interface;                            11
        superclasses            : InheritedClass;
        locDefs                 : LocalDefinition;
        classState              : State0;
        init                    : InitialState;
        ops1, ops2, ops3        : Operation0;                           16
        genParam                : FormalParameters;
        cspPart                 : CSPZ;
Index: Class1 Scope: Class0
```

className(*genParam*)?

interf

#*I* superclasses(superclasses)∗

CSPPart

state

init

#*I* ops1(ops1)∗

operation1 $\widehat{=}$ schema1 $\bigwedge$ schema3

#*I* ops2(ops2)∗

operation2 $\widehat{=}$ schema2

#*I* ops3(ops3)∗

```
  Precondition:                                                         21
        ∀ n ∈ ℕ; t ∈ traces(cspPart) • operation1 = t(n) ⇒ operation2 = t(n + 1)
        ∀ n ∈ ℕ; t ∈ traces(cspPart) • operation2 = t(n) ⇒ operation1 = t(n − 1)
        pre schema3
        input(schema3) = output(schema3) = ∅
        vars(schema1) ∩ vars(schema3) = ∅                               26
        vars(schema2) ∩ vars(schema3) = ∅
        ∃ state12, state3 • schema(state) = schema(state12) ⋀ schema(state3)
            ∧ vars(state12) ∩ vars(state3) = ∅
            ∧ vars(schema1) ⊆ vars(state12)
            ∧ vars(schema2) ⊆ vars(state12)                             31
            ∧ vars(schema3) ⊆ vars(state3)
Calculation: NONE
AfterTemplate:
Index: Class1   Scope: Class0
                                                                        36
```

className(*genParam*)?
interf
#*I* superclasses(superclasses)∗
CSPPart
state
init
#*I* ops1(ops1)∗
operation1 ≙ schema1
#*I* ops3(ops3)∗
operation2 ≙ schema2 ⋀ schema3
#*I* ops2(ops2)∗

Die Beweistechnik, die hier genutzt wird, folgt direkt aus der Definition von Refactorings einer Sicht mit Bedingung an eine andere Sicht: Es muss für die Semantik der Sicht gezeigt werden, dass, wenn die Precondition des Refactoring gilt, die Transformation auf der Sicht verhaltenserhaltend ist.

Das Refactoring soll auf Grundlage einer Bisimulation gezeigt werden. Da in diesem Fall das Refactoring nicht verhaltenserhaltend auf der Sicht ist, muss dies über die Semantik der Klasse gezeigt werden. Dabei wird nur die Semantik des Z-Teils verändert (vgl. Definition 36). Die Semantik des CSP-Teils verhindert im Beweis, dass etwas passiert, was nicht passieren soll. Sie garantiert daher, dass es sich wirklich um ein Refactoring handelt. Dies entspricht dem Fall aus Definition 36: Als gemeinsame Semantik wird hier CSP genutzt. Die lokale Verhaltenserhaltung wird die Bisimulation. Dabei wird hier die schwache Bisimulation gezeigt. Für den Schluss von der lokalen Verhaltenserhaltung werden die Techniken aus Definitionen 37 und 31 angewandt. Mithilfe der Beobachtungspunkte kann von der schwachen Bisimulation auf der Klassenebene auf die starke Bisimulation geschlossen (Satz 27.1) werden. Diese wiederum impliziert die Failure-Äquivalenz (vgl. [EF02]). Mit der Definition 37 stellt die lokale Verhaltenserhaltung die globale Verhaltenserhaltung sicher.

Satz 37.1 (`MovePredicateToSuccedingOperation`)
`MovePredicateToSuccedingOperation` ist ein Refactoring für CSP-OZ unter der Verhaltenserhaltung $\equiv_{\mathcal{F}}$.

Beweis: Es ist zu zeigen, dass es zwischen $procC_{BT} \parallel procZ_{BT}$ und $procC_{AT} \parallel procZ_{AT}$ eine schwache Bisimulation existiert. Im Folgenden soll als verkürzte Schreibweise $procC_{BT} = procC_{AT} = procC$ genutzt werden und ein Teilprozess aus $procC$ mit Q oder P benannt sein. Für den Beweis der Bisimulation (26) ist eine Relation R zu finden, welche die Bedingungen für eine Bisimulation erfüllt. Die Relation wird aus mehreren Teil-Relationen zusammengesetzt, in denen die verschiedenen Informationen verarbeitet werden. Zuerst ist klar, dass die Prozesse im uninitialisierten Zustand in Beziehung stehen:

$$R_1 = \{(procC \parallel procZ_{BT};\ procC \parallel procZ_{AT})\}$$

Die folgenden Mengen benutzen eine Induktion, um die Relation zu bilden. Für die erste Menge wird davon ausgegangen, dass alle Prozesse, deren Traces nicht mit den veränderten Operationen enden, in der Relation enthalten sind. Dies gilt nur unter der Annahme, dass eine schon in den Traces aufgetretene Hintereinanderausführung von `operation1` und `operation2` korrekt ist.

$$R_2 = \{(Q \parallel procS_{BT}(\theta state);\ Q \parallel procS_{AT}(\theta state)) \mid \\ procC \overset{tr}{\Rightarrow} Q \\ \wedge\ last(tr) \neq \texttt{operation1} \\ \wedge\ last(tr) \neq \texttt{operation2}\}$$

R_3 setzt die Prozesse in Relation, die gerade `operation1` ausgeführt haben.

$$R_3 = \{((Q \parallel procS_{BT}(\theta state_1);\ Q \parallel procS_{AT}(\theta state_{13})) \mid \\ procC \overset{tr}{\Rightarrow} Q \\ \wedge\ last(tr) = \texttt{operation1} \\ \forall\, x \in \mathbf{vars}(\texttt{schema1}) \bullet (\theta state_1)(x) = (\theta state_{13})(x)\}$$

Mit der Relation R_4 wird der Induktionsschritt geschlossen.

$$R_4 = \{((Q \parallel procS_{BT}(\theta state_{23});\ Q \parallel procS_{AT}(\theta state_{23})) \mid \\ ((P \parallel procS_{BT}(\theta state_1);\ P \parallel procS_{AT}(\theta state_{13})) \in R_3 \\ P \xrightarrow{\texttt{operation2}} Q \\ \}$$

Für die Menge R_4 ist zu zeigen, dass sie wohldefiniert ist, dass heißt, dass die entsprechenden States existieren. Dazu muss gezeigt werden, dass

$$procS_{BT}(\theta state_1) \xrightarrow{\texttt{operation2}_{BT}} procS_{BT}(\theta state_{23})$$
und
$$procS_{AT}(\theta state_{13}) \xrightarrow{\texttt{operation2}_{AT}} procS_{AT}(\theta state_{23})$$

gilt. Es gilt $\forall\, x \in \mathbf{vars}(\texttt{schema1}) \bullet (\theta state_1)(x) = (\theta state_{13})(x)\}$, da sie Vorzustände aus R_3 sind. Die Veränderungen durch die beiden Schemata `schema2` und `schema3` finden auf verschiedenen Variablenmengen statt. Das `schema1` arbeitet dabei nur auf Variablen, die in beiden Fällen den selben Wert haben und kann damit nur zu der gleichen Ergebnissmenge der entsprechenden Variablen kommen. Bleibt noch zu zeigen, dass die Variablen, die durch das `schema3` verändert werden, die entsprechenden Werte annehmen.

Die Relation

$$R = R_1 \cup \mathcal{R}_2 \cup \mathcal{R}_3 \cup \_4$$

ist eine schwache Bisimulation:

- Die initialen Zustände der beiden Transitionssysteme, stehen wegen R_1 in Relation zueinander.
- Nun muss über die Struktur gezeigt werden, das für jeden Übergang in jeweils beiden Transitionssystemen ein entsprechender im Anderen existiert und die beiden erreichten Zustände in Relation zueinander stehen. Durch die Struktur von R wird dies für jede Teilmenge separat gezeigt:
  - $(s_1, s_1') \in R_1$ mit Transition $\alpha \notin \{\texttt{operation2}, \texttt{operation2}\}$: Für jeden α-Übergang eines Transitionssystems existiert im Anderen auch ein α-Übergang. Wegen der Struktur von R_2 sind die erreichten Zustände in R.

- $(s_1, s_1') \in R_1$ mit Transition `operation1`: Für jede `operation1`-Übergang eines Transitionssystems existiert im Anderen auch ein `operation1`-Übergang. Wegen der Struktur von R_3 sind die erreichten Zustände in R.
- $(s_1, s_1') \in R_1$ mit Transition `operation2`: Aus keinen Zustand aus R_1 kann die Operation `operation2` ausgeführt werden, da im initialen Zustand kein OPeration ausgeführt wurde und damit insbesondere keine `operation1`.
- $(s_1, s_1') \in R_2$ mit Transition $\alpha \notin \{\texttt{operation2}, \texttt{operation2}\}$: Für jeden α-Übergang eines Transitionssystems existiert im Anderen auch ein α-Übergang. Wegen der Struktur von R_2 sind die erreichten Zustände in R.
- $(s_1, s_1') \in R_2$ mit Transition `operation1`: Für jede `operation1`-Übergang eines Transitionssystems existiert im Anderen auch ein `operation1`-Übergang. Wegen der Struktur von R_3 sind die erreichten Zustände in R.
- $(s_1, s_1') \in R_2$ mit Transition `operation2`: Aus keinen Zustand aus R_2 kann die Operation `operation2` ausgeführt werden, da dies durch die Definition von R_2 ausgeschlossen ist.
- $(s_1, s_1') \in R_3$ mit Transition $\alpha \notin \{\texttt{operation1}, \texttt{operation2}\}$: Die Menge R_3 enthält nur Zustände, bei denen die letzte Operation die Operation `operation1` war. Da nach diesen nur `operation2` ausgeführt werden kann, kann keine andere Operation hier auftreten.
- $(s_1, s_1') \in R_3$ mit Transition `operation1`: Diese Fall entspricht dem vorherigen.
- $(s_1, s_1') \in R_3$ mit Transition `operation2`: Für jede `operation2`-Übergang eines Transitionssystems existiert im Anderen auch ein `operation2`-Übergang. Wegen der Struktur von R_4 sind die erreichten Zustände in R.
- $(s_1, s_1') \in R_4$ mit Transition $\alpha \notin \{\texttt{operation2}, \texttt{operation2}\}$: Für jede `operation1`-Übergang eines Transitionssystems existiert im Anderen auch ein `operation1`-Übergang. Wegen der Struktur von R_2 sind die erreichten Zustände in R.
- $(s_1, s_1') \in R_4$ mit Transition `operation1`: Für jede `operation1`-Übergang eines Transitionssystems existiert im Anderen auch ein `operation1`-Übergang. Wegen der Struktur von R_2 sind die erreichten Zustände in R.
- $(s_1, s_1') \in R_4$ mit Transition `operation2`: Aus keinen Zustand aus R_4 kann die Operation `operation2` ausgeführt werden, da dies durch die Definition von R_4 ausgeschlossen ist.

Damit ist gezeigt, das R eine Bisimulation ist.

□

In diesem Beweis ist eine Bedingung an die CSP-Sicht verwendet worden. Es wurde innerhalb der CSP-OZ-Semantik ein CSP_Z Ausdruck für den CSP-Teil gebildet und dieser durch eine Parallelausführung des CSP-Teils so eingeschränkt, dass die gewollte Klassensemantik herauskommt. Das Problem bei dem Beweis ist nun, dass durch die recht kleine Änderung an den Operationen die Semantik des Z-Teils sich stark ändert. Die Semantik des Z-Teils hat sich echt geändert, somit kann dieses CSP-OZ Refactoring kein Refactoring des Z-Teils alleine sein. Erst die Einschränkung durch den CSP-Teil der Klasse führt zu einer Verhaltenserhaltung. Dabei werden die Äste des Transitionssystem des CSP_Z durch den CSP-Teil abgeschnitten und nur der Rest ist relevant.

Die Refactorings mit einer Bedingung an eine andere Sicht treten in CSP-OZ nicht häufig auf. Dies liegt an der Struktur von CSP-OZ. Die Sichten haben nur eine kleine Schnittmenge an Informationen, die konsistent sein müssen. In Sprachen, die mehr Überschneidungen in den Sichten haben (z.B. UML), treten diese Fälle häufiger auf. Dann muss über eine Bedingung z.B. sichergestellt werden, dass keine Elemente entfernt werden, die in anderen Sichten benötigt werden. So muss beim Entfernen einer Operation sichergestellt sein, dass diese nicht in einem Statediagramm benötigt wird. Dies ist der Grund, weshalb das in diesem Abschnitt gezeigte Refactoring `MovePredicateToSuccedingOperation` konstruiert ist, um diesen Fall zu verdeutlichen.

7.4.5 Refactorings mehrerer Sichten zusammen

Nachdem die verschiedenen Fälle des Refactorings einer Sicht betrachtet wurden, werden nun mehrere Sichten gleichzeitig einem Refactoring unterzogen. Manchmal müssen sich mehrere Sichten gleichzeitig ändern, um das Verhalten zu erhalten. Ein solches Refactoring ist `SplitClassCSPOZ` in CSP-OZ (vgl. Abschnitt 7.4.1). Es müssen sowohl der CSP-Teil als auch der Z-Teil in zwei neue Klassen verteilt werden. Auch die System-Sicht muss an die neuen Klassen angepasst werden.

Satz 37.2 (`SplitClassCSPOZ`)
`SplitClassCSPOZ` ist ein Refactoring für CSP-OZ unter der Verhaltenserhaltung $\equiv_{\mathcal{FD}}$.

Beweis: Der Beobachtungspunkt für dieses Refactoring ist die System-Ebene, also der CSP-Prozess, der das Zusammenspiel mit den anderen Klassen regelt. In dieser Komponentensicht wird die Klasse `class` durch zwei Klassen (`classNeuA` und `classNeuB`) ersetzt:

$$P(\texttt{class}) \equiv P(\texttt{classNeuA} \underset{\texttt{X} \cup \{\texttt{init}\}}{\parallel} \texttt{classNeuB})$$

Daraus folgt, dass der System-CSP-Prozess P, welcher mit der Klasse parametrisiert ist, nach dem Refactoring mit der Kombination der beiden neuen Klassen parametrisiert wird. Die Variable `X` steht dabei als Platzhalter für eine Menge von Events. Wegen den Konguренzeigenschaften von CSP ist dies gleichbedeutend mit:

$$\texttt{class} \equiv \texttt{classNeuA} \underset{\texttt{X} \cup \{\texttt{init}\}}{\parallel} \texttt{classNeuB}$$

Dies soll gezeigt werden. Mit der Semantik der CSP-OZ-Klasse (vgl. Kapitel 3.3.1) ergibt sich:

$$\begin{array}{l} [\![\texttt{class}(\texttt{formalPara})?]\!] \\ \equiv \\ procC \underset{\mathbf{events}(procC) \cap \mathbf{events}(procZ)}{\parallel} procZ \\ \equiv \\ (\texttt{P}_{BT} \underset{\texttt{X}}{\parallel} \texttt{Q}_{BT}) \underset{\mathbf{events}(procC) \cap \mathbf{events}(procZ)}{\parallel} \mathbf{init}(\texttt{init}_{BT}) \rightarrow procS(\theta\texttt{state}) \end{array}$$

In der letzten Zeile sind die ersten Meta-Variablen in der Formel aufgetaucht (`P`, `Q` und `init`). Die weitere Auswertung von *procS* führt zu:

$$\begin{array}{l} (\texttt{P}_{BT} \underset{\texttt{X}}{\parallel} \texttt{Q}_{BT}) \underset{\mathbf{events}(procC) \cap \mathbf{events}(procZ)}{\parallel} \mathbf{init}(\texttt{init}) \rightarrow \\ \quad (\ \Box\, op : \mathbf{names}(\texttt{op}_{BT}) \bullet procO_{\texttt{class}}(op, \theta\, \mathbf{schema}(\texttt{state}))) \end{array}$$

Im nächsten Schritt werden die Meta-Variablen durch die Entsprechungen des AfterTemplates ersetzt:

$$(\mathtt{P}_{AT} \underset{\mathtt{X}}{\parallel} \mathtt{Q}_{AT}) \quad \underset{\mathbf{events}(procC)\cap\mathbf{events}(procZ)}{\parallel} \quad \mathbf{init}(\mathtt{inita} \wedge \mathtt{initb}) \rightarrow (\ \Box\ op : \mathbf{names}(\mathtt{opa}_{AT} \cup \mathtt{opb}_{AT}) \bullet procO_{\mathtt{class}}(op, \theta\, \mathbf{schema}(\mathtt{statea} \wedge \mathtt{stateb})))$$

Als Nächstes werden die Meta-Variablen frei gestellt:

$$(\mathtt{P}_{AT} \underset{\mathtt{X}}{\parallel} \mathtt{Q}_{AT}) \quad \underset{\mathbf{events}(procC)\cap\mathbf{events}(procZ)}{\parallel} \quad \mathbf{init}(\mathtt{inita}) \wedge \mathbf{init}(\mathtt{initb}) \rightarrow (\ \Box\ op : \mathbf{names}(\mathtt{opa}_{AT}) \cup \mathbf{names}(\mathtt{opb}_{AT}) \bullet procO_{\mathtt{class}}(op, \theta\, \mathbf{schema}(\mathtt{statea}) \wedge \mathbf{schema}(\mathtt{stateb})))$$

Nun wird *ProcS* anhand der Teilmengen umgeformt. Dabei muss auch der Aufruf von $procO_{\mathtt{class}}$ geändert werden. Der Schritt kann so ausgeführt werden, weil die States und die Operationen streng getrennt sind. Hier muss mithilfe der Failures argumentiert werden, die Divergenzen müssen nicht betrachtet werden, weil im Z-Teil keine Divergenzen auftreten können (vgl. [Fis00]). Der entscheidene Punkt ist, dass *procO* rekursiv definiert ist, dass heißt sich immer wieder selbst aufruft. Nach jedem Trace ist damit die Refusal-Menge der Operationssynchronisation (vgl. Kapitel 3.3.1) nur von pre *Op* abhängig. Da alle anderen Transitionen τ-Transitionen sind, kann $procS_{\mathtt{class}}$ in zwei Teile geteilt werden:

$$procS_{\mathtt{classNeuA}}(\mathtt{statea}) \stackrel{c}{=} \Box\ op : \mathbf{names}(\mathtt{opa}_{AT}) \bullet procO_{\mathtt{classNeuA}}(op, \theta\mathtt{statea})$$

$$procO_{\mathtt{classNeuA}}(op, \mathtt{statea}) \stackrel{c}{=} \Box\, \mathbf{input}(op) \mid \mathrm{pre}\ op \bullet \sqcap\, \mathbf{output}(op);\ \mathtt{statea}' \mid \mathbf{schema}(op) \bullet op.(\theta\, \mathbf{input}(op) \wedge \theta\, \mathbf{output}(op)) \rightarrow procS_{\mathtt{classNeuA}}(\theta state')$$

$$procS_{\mathtt{classNeuB}}(\mathtt{stateb}) \stackrel{c}{=} \Box\ op : \mathbf{names}(\mathtt{opb}_{AT}) \bullet procO_{\mathtt{classNeuB}}(op, \theta\mathtt{stateb})$$

$$procO_{\mathtt{classNeuB}}(op, \mathtt{stateb}) \stackrel{c}{=} \Box\, \mathbf{input}(op) \mid \mathrm{pre}\ op \bullet \sqcap\, \mathbf{output}(op);\ \mathtt{stateb}' \mid \mathbf{schema}(op) \bullet op.(\theta\, \mathbf{input}(op) \wedge \theta\, \mathbf{output}(op)) \rightarrow procS_{\mathtt{classNeuB}}(\theta state')$$

Damit ergibt sich:

$$(\mathtt{P}_{AT} \underset{\mathtt{X}}{\parallel} \mathtt{Q}_{AT}) \quad \underset{\mathbf{events}(procC)\cap\mathbf{events}(procZ)}{\parallel} \quad \mathbf{init}(\mathtt{inita}) \wedge \mathbf{init}(\mathtt{initb}) \rightarrow (\ \Box\ op : \mathbf{names}(\mathtt{opa}_{AT}) \bullet procO_{\mathtt{class}}(op, \theta\, \mathbf{schema}(\mathtt{statea}))) \ |||\ (\ \Box\ op : \mathbf{names}(\mathtt{opb}_{AT}) \bullet procO_{\mathtt{class}}(op, \theta\, \mathbf{schema}(\mathtt{stateb})))$$

Im nächsten Schritt werden die Initialisationen aufgeteilt. Dabei müssen die Initialisationen in den beiden Teilen immer gleichzeitig stattfinden, weshalb sie in die Synchronisation aufgenommen werden müssen:

$$
\begin{array}{l}
(\mathtt{P}_{AT} \parallel \mathtt{Q}_{AT}) \qquad\qquad \parallel \\
\quad {\scriptstyle \mathtt{X}}\ \mathbf{events}(procC) \cap \mathbf{events}(procZ) \\
\qquad\qquad\qquad\qquad\qquad (\mathbf{init}(\mathtt{inita}) \rightarrow\ \Box\, op : \mathbf{names}(\mathtt{opa}_{AT}) \bullet \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad procO_{\mathtt{class}}(op, \theta\, \mathbf{schema}(\mathtt{statea}))) \\
\qquad\qquad\qquad\qquad\qquad\ \ \parallel \\
\qquad\qquad\qquad\qquad\qquad {\scriptstyle \{\mathtt{init}\}} \\
\qquad\qquad\qquad\qquad\qquad (\mathbf{init}(\mathtt{initb}) \rightarrow\ \Box\, op : \mathbf{names}(\mathtt{opb}_{AT}) \bullet \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad procO_{\mathtt{class}}(op, \theta\, \mathbf{schema}(\mathtt{stateb})))
\end{array}
$$

Jetzt sollen die Paralleloperatoren umsortiert werden. Dazu müssen erst einmal die Events sortiert werden. Wenn die beiden Terme, die mit **init** beginnen, als $procZ$ für die neue Klasse aufgefasst werden, gilt:

$$\mathbf{events}(procZ) = \mathbf{events}(procZ_1) \cup \mathbf{events}(procZ_2)$$

Wegen der Teilmengenbeziehungen:

$$
\begin{array}{l}
X \subset \mathbf{events}(procC) \\
\{\mathtt{init}\} \subset \mathbf{events}(procZ_1) \\
\{\mathtt{init}\} \subset \mathbf{events}(procZ_2)
\end{array}
$$

gilt:

$$
\begin{array}{l}
(\mathtt{P}_{AT} \qquad\qquad \parallel \qquad\qquad \mathbf{init}(\mathtt{inita}) \rightarrow (\ \Box\, op : \mathbf{names}(\mathtt{opa}_{AT}) \bullet \\
\quad {\scriptstyle \mathbf{events}(procC) \cap \mathbf{events}(procZ_1)} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad procO_{\mathtt{class}}(op, \theta\, \mathbf{schema}(\mathtt{statea})))) \\
\quad \parallel \\
{\scriptstyle X \cup \{\mathtt{init}\}} \\
(\mathtt{Q}_{AT} \qquad\qquad \parallel \qquad\qquad \mathbf{init}(\mathtt{initb}) \rightarrow (\ \Box\, op : \mathbf{names}(\mathtt{opb}_{AT}) \bullet \\
\quad {\scriptstyle \mathbf{events}(procC) \cap \mathbf{events}(procZ_2)} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad procO_{\mathtt{class}}(op, \theta\, \mathbf{schema}(\mathtt{stateb}))))
\end{array}
$$

Jetzt kann mit den Semantikdefinitionen des Z-Teils die Formel rückabgewickelt werden:

$$
\begin{array}{l}
(\mathtt{P}_{AT} \qquad\qquad \parallel \qquad\qquad \mathbf{init}(\mathtt{inita}) \rightarrow procS_1(\theta \mathtt{statea})) \\
\quad {\scriptstyle \mathbf{events}(procC) \cap \mathbf{events}(procZ_1)} \\
\quad \parallel \\
{\scriptstyle X \cup \{\mathtt{init}\}} \\
(\mathtt{Q}_{AT} \qquad\qquad \parallel \qquad\qquad \mathbf{init}(\mathtt{initb}) \rightarrow procS_2(\theta \mathtt{stateb})) \\
\quad {\scriptstyle \mathbf{events}(procC) \cap \mathbf{events}(procZ_2)} \\
\equiv \\
(\mathtt{P}_{AT} \qquad\qquad \parallel \qquad\qquad procZ_1) \\
\quad {\scriptstyle \mathbf{events}(procC) \cap \mathbf{events}(procZ_1)} \\
\quad \parallel \\
{\scriptstyle X \cup \{\mathtt{init}\}} \\
(\mathtt{Q}_{AT} \qquad\qquad \parallel \qquad\qquad procZ_2) \\
\quad {\scriptstyle \mathbf{events}(procC) \cap \mathbf{events}(procZ_2)}
\end{array}
$$

Die Menge der Events $\mathbf{events}(procC)$ muss noch angepasst werden. Da die Events in dem Refactoring streng aufteilbar sind, gilt:

$$
\begin{array}{l}
\forall\, x \in \mathbf{events}(procC) \setminus \mathbf{events}(Q_{AT} \bullet x \notin \mathbf{events}(procZ_1) \\
\forall\, x \in \mathbf{events}(procC) \setminus \mathbf{events}(P_{AT} \bullet x \notin \mathbf{events}(procZ_2)
\end{array}
$$

Deshalb können die Mengen passend verkleinert werden:

$$
\begin{array}{l}
(\mathtt{P}_{AT} \quad \underset{\mathbf{events}(\mathtt{P}_{AT}) \cap \mathbf{events}(procZ_1)}{\|} \quad procZ_1) \\
\quad \underset{X \cup \{\mathtt{init}\}}{\|} \\
(\mathtt{Q}_{AT} \quad \underset{\mathbf{events}(\mathtt{Q}_{AT}) \cap \mathbf{events}(procZ_2)}{\|} \quad procZ_2)
\end{array}
$$

Nun kann die Semantikformel für eine CSP-OZ-Klasse angewendet werden:

$$
\mathtt{classNeuA} \quad \underset{\mathtt{X} \cup \{\mathtt{init}\}}{\|} \quad \mathtt{classNeuB}
$$

Damit ist der Beweis abgeschlossen. □□

Es sind die drei Grundfälle für die Verhaltenserhaltungsbeweise betrachtet worden. Die Fälle können bei Refactorings, die mehrere Sichten betreffen, gemischt auftreten.

Kapitel 8

Diskussion und Ausblick

Refactorings haben, nach der Welt der Programmiersprachen, in den letzten Jahren die Welt der Modelle und Spezifikationen erobert. Da Modelle und Spezifikationen die Software vollständig beschreiben müssen, treten im Gegensatz zu Programmcode andere Fragestellungen auf. Erschwerend kommt hinzu, dass Modelle und Spezifikationen häufig mit Sichten arbeiten. Sie helfen bestimmte Sachverhalte in den Fokus zu bringen. Hier sollen nochmal die Ergebnisse dieser Arbeit zusammengefasst werden.

Es wurde eine formale Definition für Refactorings gegeben, die auf Modellsprachen anwendbar ist. Der Hauptunterschied zu den existierenden Definitionen [Fow04, Rob99, Cor04] ist die explizite formale Erfassung der Verhaltenserhaltung. Die Verhaltenserhaltung ist als Relation auf den semantischen Domänen der Sprache definiert. Eine Verhaltenserhaltungsrelation muss transitiv und reflexiv sein. Die Transitivität formalisiert die Beobachtung, dass die Hintereinanderausführung von Refactorings das Verhalten des ursprünglichen Programms erhalten muss. Die Reflexivität beschreibt, dass es ohne Änderung des Codes keine Änderung des Verhaltens gibt.

In der Definition ist bewusst die Verhaltenserhaltung nicht als Äquivalenzrelation gewählt worden, da es Arbeiten gibt, die die Verhaltenserhaltung nicht als Äquivalenzrelation betrachten [Cor04]. Die hier vorliegende Definition kann daher auch auf diese Arbeiten angewandt werden.

Es wurde mit der Sprachfamilie Re$\mathcal{L}$ eine Beschreibung für Refactorings von BNF-basierten Sprachen geschaffen, die sowohl für die Ausführung in einer IDE als auch für die formale Analyse geeignet ist. Das zentrale Element von Re$\mathcal{L}$ ist die Beschreibung der Refactorings mittels Templates. Re$\mathcal{L}$ nutzt dabei Techniken der domänenspezifischen Sprachen, um an eine Zielsprache angepasst zu werden.

Dieses Vorgehen hat im Vergleich zu anderen Ansätzen den Vorteil, neben neuen Eigenschaften auch vorhandene Techniken in einer Sprache zu vereinen.

Re$\mathcal{L}$ besitzt eine für direkte Beweise geeignete Struktur Durch die Verwendung von Meta-Variablen und Templates, um den Code vor und nach dem Refactoring zu beschreiben, kann eine partielle Auswertung einer kompositionellen Semantik genutzt werden. Dabei werden die Meta-Variablen nicht ausgewertet, sondern bleiben als Term innerhalb der Semantikklammern stehen. Auf diesen teil-ausgewerteten Ausdrücken werden die Beweise durchgeführt. Eine Übertragung des Refactorings in eine für Beweise geeignete Form ist nicht notwendig.

ReL unterstützt direkt explizit mehrsichtige Modelle Durch die Unterteilung von Templates in Subtemplates, die in verschiedenen Sichten verwendet werden können, eignet sich ReL gut für die Beschreibung von Refactorings die mehrere Sichten einschließen. Andere Refactoringbeschreibungen, die prinzipiell mehrere Sichten beherrschen, bauen entweder auf Multipurpose-Sprachen auf oder es wird das Refactoring aufgrund des Gesamtmodells betrachtet. Zu den Ersten gehört JunGL [Ver08]. JunGL besteht aus einem Framework, welches Zugriffe auf den Code oder das Modell erlaubt und wird ansonsten wie eine normale Programmiersprache genutzt. Mehrsichtige Refactorings auf dem zugrunde liegenden Modell können mit den meisten Refactoringbeschreibungssystemen genutzt werden. Im mehrsichtigen Fall werden die Refactorings dabei unübersichtlich, was hauptsächlich aus der „Entfernung“ der Modellelemente im annotierten AST herrührt.

ReL eignet sich für domänenspezifische Sprachen Die generierte Toolunterstützung erlaubt es, für beliebige BNF-basierte Sprachen schnell eine Refactoringumgebung zu erstellen. Dadurch kann auch bei Sprachen, die ad hoc erstellt werden, schnell mit der direkten Spezifikation von Refactorings begonnen werden. Dies ist besonders bei domänenspezifischen Sprachen der Fall.

ReL vereinfacht die strukturierte Erstellung von Refactorings Durch den Aufbau von ReL ist eine für Programmierer natürliche Notation von Refactorings gegeben. Die Erstellung eines Refactorings kann, wie im Beispiel gezeigt, strukturiert durchgeführt werden. Dabei wird erst eine unvollständige Beschreibung des Refactorings erstellt, die in weiteren Schritten verfeinert wird.

ReL erlaubt einfachere Refactorings durch Polymorphie zu definieren Das Repository von ReL erlaubt es Refactorings polymorph zu erstellen. Ein Refactoring kann mit den gleichen Parametern mehrmals definiert sein. Die Auswahl erfolgt aufgrund der Erfüllung der Vorbedingung. Wenn mehrere Versionen eines Refactorings die Vorbedingung erfüllen, wird die eher definierte Version gewählt.

ReL unterstützt zusammengesetzte Refactorings Eine große Zahl von Refactorings lässt sich durch die Zusammensetzung anderer Refactorings beschreiben. Daher bietet ReL die Möglichkeit, Refactorings aus anderen Refactorings aufzubauen.

Der zweite große Block in dieser Arbeit beschäftigt sich mit dem Beweisen der Verhaltenserhaltung von Refactorings. Dabei werden insbesondere Techniken für mehrsichtige Sprachen betrachtet. Dazu werden, ausgehend von den Techniken für eine Sicht, die Techniken für mehrere Sichten eingeführt. Die Semantik des Codes wird in den Beweisen nur soweit wie nötig ausgewertet, um die Verhaltenserhaltung zu zeigen. Eine zweite wichtige Technik ist der Beobachtungspunkt. Er erlaubt es, von der Verhaltenserhaltung im Innern des Codes oder Modells auf die Verhaltenserhaltung des Gesamtmodells zu schließen. Die Beobachtungspunkte können dazu auch andere Verhaltenserhaltungsbegriffe nutzen als das Gesamtmodell. Dies ist insbesondere nützlich, wenn von einer schwachen Verhaltenserhaltung auf eine starke geschlossen werden kann. In CSP-OZ, einer Beispielsprache dieser Arbeit, tritt dies im Z-Teil auf. Hier kann von der schwachen Failure-Äquivalenz auf die Failure-Divergences-Äquivalenz des Gesamtsystems geschlossen werden.

Die Beobachtungspunkte werden im mehrsichtigen Fall zur Reduktion der betrachteten Sichten verallgemeinert. In dieser Arbeit wurden drei Fälle der Wechselwirkung zweier Sichten behandelt, welche die Grundbausteine für den mehrsichtigen Fall darstellen.

Unabhängiges Refactoring einer Sicht Einige Refactorings beschränken sich auf eine einzelne Sicht. Wenn das Refactoring unabhängig von anderen Sichten durchgeführt werden kann, dann ist es ein *Unabhängiges Refactoring einer Sicht*. In diesem Fall kann die Verhaltenserhaltung des Gesamtsystems aus einem passenden Verhaltenserhaltungsbegriff der Sicht gefolgert werden. So ist es in CSP-OZ möglich, zwei Schemata, die durch eine Konjunktion verbunden sind, in ein einzelnes Schema umzuwandeln, und dabei als Verhaltenserhaltung die Semantik-Äquivalenz zu nutzen. Aus dieser Relation folgt dann die Verhaltenserhaltung des CSP-OZ Gesamtsystems in der Failure-Divergences-Äquivalenz.

Refactorings einer Sicht unter Bedingung an eine andere Sicht Nicht bei allen Refactorings, die nur eine Sicht betreffen, kann die Verhaltenserhaltung ohne Berücksichtigung anderer Sichten durchgeführt werden. Andere Sichten müssen oft einige Nebenbedingungen erfüllen, damit das Refactoring verhaltenserhaltend durchgeführt werden kann.

Echt mehrsichtige Refactorings Der letzte Fall betrifft Refactorings, bei denen zwei oder mehr Sichten gleichzeitig geändert werden. In diesem Fall ist es nicht möglich den Beweis auf eine Sicht zu beschränken.

Die vorgestellten Techniken vereinfachen die Beweise der Verhaltenserhaltung durch die Fokussierung auf den „interessanten“ Bereich. So genügt es häufig, eine einzelne Sicht zu betrachten. Auch werden die Beweistechniken von der Beschreibungssprache Re$\mathcal{L}$ unterstützt, welche die benötigten Informationen in einer guten Struktur bereitstellt.

8.1 Verwandte Arbeiten

Das Thema Refactoring ist in den letzten Jahren ständig präsent in der Literatur und der Forschung. Neben der Literatur, die Refactorings selber betrachtet, gibt es viele Arbeiten, die einen Teilbereich bearbeiten, ohne das Thema Refactorings direkt zu adressieren. So sind alle Arbeiten zu semantikerhaltenden Transformationen für Refactorings relevant, wenn die Ausgangssprache und die Zielsprache dieselbe ist. Verwandte Arbeiten zur Beschreibung von Refactorings sind schon in der Analyse von bestehenden Beschreibungsansätzen in Kapitel 5.1 beschrieben, daher wird hier der Fokus auf die Korrektheit der Verhaltenserhaltung gelegt.

Transformationen Refactorings sind Transformationen innerhalb einer Sprache mit der speziellen Eigenschaft der Verhaltenserhaltung. Daher ist im Zusammenhang mit Refactorings der ganze Themenkomplex der Transformationen interessant. Da die Semantikerhaltung die stärkste Verhaltenserhaltung ist, betrifft dies insbesondere semantikerhaltende Transformationen. Oft werden Modelltransformationen durch Graphtransformationen beschrieben. Cabot et al. [CCGdL10] beschreibt einen Ansatz für die Analyse von Graphtransformationen, deren Semantik durch OCL induziert ist. Einer Graphtransformation wird eine Semantik in OCL gegeben, indem die Beschreibung des Vorzustandes und die Nichtanwendbarkeitsregeln in eine OCL-Vorbedingung umgewandelt werden. Entsprechend

wird die Beschreibung des Nachzustandes als eine OCL-Nachbedingung ausgedrückt. Dabei können Anwendbarkeit (Applicability) oder Konflikte zwischen den Refactoring-Regeln (Conflict) gut analysiert werden. Dieser Ansatz ähnelt von den verwendeten Prinzipien der Semantikbildung von Re$\mathcal{L}$, aber durch die fehlende Einbeziehung der Semantik der Zielsprache, lässt sich durch diesen Ansatz nicht die Verhaltenserhaltung eines Refactorings beweisen.

Auch [BHM09] analysiert Graphtransformationen, indem diesen zu eine gut für die Analyse geeignete Semantik gegeben wird. Die semantische Domäne der Rewriting Logic [BN99] erlaubt es, die Semantik der Transformationen auf die Modelle anzuwenden. Eigenschaften der Transformation können durch Model Checking nachgewiesen werden. Die dargestellte Übertragung eignet sich für die Prüfung von Linear Time Logic [Pnu77]-Bedingungen und ist daher nicht für die Prüfung der Verhaltenserhaltung geeignet.

Generische Refactorings Refactorings innerhalb eines Paradigmas (Objekt-Orientierung, Funktionale Programmierung) ähneln sich. Sie haben über Sprachgrenzen hinweg einige Gemeinsamkeiten. Daher stellt sich die Frage, ob eine generische Beschreibung gefunden werden kann, die Refactorings in verschiedenen Sprachen beschreibt. Arbeiten, die den Weg in diese Richtung beschreiten, sind u.a. [Läm02], in welcher generische Refactorings in Haskel beschrieben werden. Die Refactoringskelette müssen als Anpassung für die jeweilige Zielsprache ausprogrammiert werden. Somit sind die Refactorings an sich nicht generisch. Einen Schritt weiter ist der Ansatz von [MMBJ09]. Aufbauend auf einem generischen Meta-Modell werden Refactorings spezifiziert. Die Anpassung an die Zielsprache wird über eine Meta-Modellierung der Zielsprache zusammen mit einer Adaption zwischen dem generischen Modell und dem Modell der Zielsprache hergestellt. Dies hat den Vorteil, dass die Refactorings selber kaum Anpassungen bedürfen. Dies wird mit der Einschränkung erkauft, dass sich nur Refactorings ausdrücken lassen, die sich in der Schnittmenge der Sprachmodelle befinden. Dies verhindert insbesondere eine tiefere Analyse von einzelnen Codestücken. Refactorings, die dies benötigen (Extract Method), sind daher mit dem Ansatz nicht möglich.

Korrektheit von Refactorings Die Korrektheit von Refactorings ist eines der großen Ziele bei der Erforschung von Refactorings [SEdM08a]. Die Ergebnisse in diesem Bereich lassen sich in zwei Gruppen einteilen. Es gibt Ansätze, die versuchen einen allgemeinen Mechanismus für die Korrektheitsbeweise der Refactorings zu finden [BM06, ST08, ERW07] und es gibt Arbeiten, die sich mit der Korrektheit von Refactorings für bestimmte Formalismen beschäftigen [MS06, SPT03]. Beide Ansätze haben untereinander eine positive Wechselwirkung. Die vielen Beispiele spezieller Refactorings zeigen die Punkte auf, die in der Generalisierung beachtet werden müssen. Im Folgenden sollen einige interessante Arbeiten genauer betrachtet werden. Bei den Beweisansätzen fällt auf, dass die Refactoringbeschreibung fallbezogen per Hand in eine beweisbare Form überführt wird.

Bannwart et al. [BM06] betrachtet die Beweise von Refactorings auf einer kleinen objektorientierten Beispielsprache. Die Refactorings werden kurz natürlichsprachlich bewiesen und formal in der Semantik dargestellt. Auf dieser Darstellung wird die Verhaltenserhaltung als Semantik-Äquivalenz gezeigt. Wie bei den meisten Arbeiten zu Beweisen von Refactorings, zeigt sich hier die Übertragungslücke von der Beschreibung des Refactorings in eine beweisbare Form. Dies wird per Hand durchgeführt. Bei Bannwart et al. kann durch die Kleinheit der Sprache gut gesehen werden, dass die Übertragung des analysierten Re-

factorings in eine ausführbare Beschreibung vollständig ist. Bei größeren Sprachen ist dies schwieriger. An diesem Problem leiden die meisten Betrachtungen von beweisbar korrekten Refactorings. Eine Betrachtung der Korrektheit eines Refactorings direkt auf der Beschreibung ist, ähnlich wie im hier vorgestellten Ansatz, im Bereich der Graphtransformationen üblich.

Beweise von Refactorings werden in den meisten Fällen ad hoc, ohne generelle Richtschnur für das Vorgehen angegeben. Daher ist die Beweisführung ein schwieriger und aufwändiger Prozess, der nur von erfahrenen Personen durchgeführt werden kann. So liegt die Idee nahe, diese Beweisführung zu automatisieren. [ST08] stellt die Beweisführung von Refactorings für Lambda-Ausdrücke mittels des Theorembeweiser Isabelle/HOL [NPW02, Pau89] vor. Dabei muss die ganze Sprache in Isabelle/HOL abgebildet werden, bevor die Korrektheit von Refactorings für diese Sprache gezeigt werden können. Auch die Refactorings müssen in eine entsprechende Darstellung umgewandelt werden.

Neben einem Theorembeweiser kann auch Model Checking für die Überprüfung der Verhaltenserhaltung genutzt werden. Dies wird in den Arbeiten [GMB05], [EW08] und [ERW07] gezeigt. Auch hier muss die Zielsprache meist in den entsprechenden Formalismus umgeformt werden. Die Ergebnisse zeigen, dass dieses Vorgehen eine gute automatische Überprüfung der Verhaltenserhaltung ermöglicht. Dieses Vorgehen stößt bei Programmen schnell auf Größengrenzen, da das Model Checking an die Stateexplosiongrenze stößt.

Spezielle Refactorings Viele Arbeiten beschäftigen sich nicht mit Refactorings im Allgemeinen, sondern entweder mit speziellen neuen Refactorings oder mit der Übertragung von Refactorings auf andere Sprachen bzw. Paradigmen. So ist beispielsweise eine Sammlung von Refactorings für Z [SPT03] erstellt worden. Für Object-Z ist eine Grundmenge von Refactorings definiert worden, die ausreicht, jede innere verhaltenserhaltende Änderung auszudrücken [McC04, MS06]. Refactorings für Erlang und Haskell sind u.a. im Zusammenhang mit dem Wrangler Tool [LT08] definiert [LT06, Li06]. Insgesamt wurden Refactorings auf viele verschiedene Bereiche übertragen, einschließlich der Spezifikation von objektorientierten Datenbanken [Ber91], formale Methoden [DW06, SPT03] und funktionale Programmiersprachen [PP91].

Korrektheit mehrsichtiger Refactorings [BM07] betrachtet zwei Sichten: das UML-Klassen-Diagramm und OCL-Constraints für Operationen. In diesem Papier wird ein Refactoring der Art „Refactoring einer Sicht mit Bedingung an eine andere Sicht“ gezeigt. Dieses Refactoring wird bewiesen. Bei diesem Beweis müssen einige Konstrukte der UML in Graphtransformationen umgewandelt werden. So wird eine Semantik von OCL in Form von Graphtransformationen genutzt. Dies rührt daher, dass die vorgestellte Technik auf Graphtransformationen beruht.

Derrick und Wehrheim [DW06, DW10] betrachten eine Gruppe von Refactorings, die Redundanzen zwischen zwei Sichten einführen und entfernen können. Diese Refactorings bilden eine wichtige Grundlage für den Aufbau von Refactorings mehrsichtiger Modelle, weil sie für die Erstellung vieler zusammengesetzter Refactorings benötigt werden. Die Refactorings, die nach der Klassifikation dieser Arbeit als Refactorings einer Sicht mit Bedingung an eine andere Sicht betrachtet werden können, werden als Refactorings auf der Gesamtsemantik gezeigt.

Beschreibungen von Refactorings auf mehrsichtigen Modellen, die diese Refactorings auf dem Gesamtmodell betrachten (vgl. Kapitel 7.4.5), beinhalten oft technisch benötigte

Bestandteile, die für das eigentliche Refactoring uninteressant sind. Dies verkompliziert die Betrachtung der Verhaltenserhaltung. Abhilfe schafft die *Rule Extraction*, mit der die Refactorings vereinfacht werden können. Dies wird anhand von Graphtransformationen in [BHE08, Bis08] vorgestellt.

8.2 Ausblick

Die Prävalidierung und die Beschreibung von Refactorings ist durch diese Arbeit ein gutes Stück voran gebracht worden. Aus den neuen Erkenntnissen ergeben sich weitere interessante Fragestellungen, Forderungen und Ideen für die Zukunft.

Integration in eine IDE Refactorings sind ein Werkzeug in der Modellierung von Systemen und in der Entwicklung von Software. Ein Werkzeug sollte von einem Modellierer oder Entwickler angewandt werden können, ohne sich tief in das Themenfeld einarbeiten zu müssen oder den Hintergrund des Werkzeugs vollständig verstanden zu haben.

Wenn eine Technik als Tool in einer Entwicklungsumgebung eingearbeitet ist, muss der Nutzer des Tools nur die Auswirkungen auf seinen Code kennen, ohne die Hintergründe vollständig zu verstehen. Daher ist die Integration von Refactorings, so wie sie in dieser Arbeit behandelt wurden, in eine weit verbreitete IDE wünschenswert. Durch die einfache Konfiguration mittels Re$\mathcal{L}$-Repositories, kann auch für wenig genutzte Sprachen schnell eine Menge von Refactorings hinzugefügt werden.

Für die Entwicklung einer Refactoringkomponente müssen der Re$\mathcal{L}$-Implementierung die zusätzlichen Module, die im RMC-Tool getestet worden sind, hinzugefügt werden. Dazu gehören die Benutzerschnittstelle, die Erhaltung der Formatierung und die Erkennung von Refactoring-Möglichkeiten (vgl. Kapitel 5.13).

Formale Semantik für Programmier- und Modellsprachen Die in dieser Arbeit beschriebenen Techniken zur Prävalidierung von Refactorings bauen auf einer formalen Semantik auf. Für BNF-basierte Sprachen, die eine kompositionelle Semantik besitzen, kann mit den Techniken vor der Anwendung des Refactorings die Korrektheit der Verhaltenserhaltung gezeigt werden. Ziel ist es, diese Prävalidierung auch auf Programmiersprachen anwenden zu können. Diesen fehlt zur Zeit aber meist die benötigte formale Semantik. Ziel muss es daher sein, die Bedeutung der formalen Semantiken in den Bereich der Sprachentwicklung hinein zu tragen.

(Teil-)Automatisierung der Beweise Die Prävalidierung der Refactorings garantiert die Verhaltenserhaltung der Transformation. Jedes Refactoring muss dazu per Hand bewiesen werden. Wünschenswert ist es, dass diese Beweise automatisiert werden können. Als Grundlage können Beschreibungen in Re$\mathcal{L}$ genutzt werden, da diese in maschinenlesbarer Form vorliegen. Eine halbautomatische Technik wird zur Zeit schon von Tools zur Verfügung gestellt. Isabelle/HOL [NPW02, Pau89] kann aufgrund von internen Modellen (Theorien) interaktiv mit dem Benutzer schnell Beweise finden. Beim Einsatz von Isabelle/HOL würde die Hauptarbeit in einem Modell der Refactorings und der Zielsprache liegen.

Anhang A

Mathematische Grundlagen

In diesem Anhang werden einige Schreibweisen und Zusammenhänge erläutert, die in der Arbeit vorausgesetzt werden, aber nicht als allgemein bekannt vorausgesetzt werden. Dieser Anhang soll als Nachschlagewerk bei Fragen konsultiert werden. Die hier dargestellten Sachverhalte lehnen sich an [Int02] und [HMU02] an, wobei die Schreibweisen vereinheitlicht wurden. Wenn es Konflikte gab, wurden geänderte Schreibweisen eingeführt.

A.1 Multimengen

Definition 38 (Multimenge) *Eine* Multimenge N über einer Menge M *ist ein Paar* $N = (M, V)$*, wobei* V *eine Abbildung von* M *in die natürlichen Zahlen* $\mathbb{N}_1$ *ist. Die Zahl* $V(x)$ *gibt an, wie oft* x *in* M *vorkommt.*

Anschaulich ist eine Multimenge eine Menge, in der jedes Element beliebig oft vorkommen kann. Man notiert Multimengen dann auch wie Mengen explizit mit geschweiften Klammern und schreibt ein Element so oft hinein, wie es in der Multimenge vorkommt. Mengen sind in diesem Sinne ein Spezialfall von Multimengen, bei denen jeder Wert nur einmal vorkommen kann.

$$set((M, V)) = M$$

Die Teilmengenbeziehung der Mengen kann auf Multimengen übertragen werden.

Definition 39 *Seien* $N_1 = (M_1, V_1)$ *und* $N_2 = (M_2, V_2)$ *Multimengen. Dann ist* N_1 *eine Teilmenge von* N_2*, geschrieben* $N_1 \subseteq N_2$*, wenn*

1. $M_1 \subseteq M_2$
2. $\forall\, x \in M_1 \bullet V_1(x) \leq V_2(x)$.

Mithilfe der Teilmengenbeziehung werden wie bei normalen Mengen andere Mengenoperatoren ableitet. Für die Gleichheit gilt: Sind N_1 und N_2 Multimengen, dann ist

$$N_1 = N_2 :\Leftrightarrow N_1 \subseteq N_2 \wedge N_2 \subseteq N_1$$

In der Arbeit wird manchmal eine normale Menge mit einer Multimenge identifiziert. Dies ermöglicht, eine normale Menge in Relation zu einer Multimenge zu setzen.

Definition 40 (Identifizierung einer Menge und Multimenge) *Sei M eine Menge. Dann kann M mit einer Multimenge $N = (M, V)$ identifiziert werden, die*

$$\forall x \in M \bullet V(x) = 1$$

erfüllt.

A.2 Relationen und Funktionen

Funktionen sind wie üblich als spezielle Relationen definiert.

Definition 41 (Relation) *Seien A und B Mengen, dann ist eine Relation $\leftrightarrow$ zwischen A und B definiert als:*

$$A \leftrightarrow B := \mathbb{P}(A \times B)$$

.

Definition 42 (Funktionen) *Seien A und B Mengen, dann ist*

- $A \nrightarrow B := \{f : A \leftrightarrow B \mid \forall p, q : f \mid first(p) = first(q) \bullet second(p) = second(q)\}$ *eine partielle Funktion,*
- $A \rightarrow B := \{f : A \leftrightarrow B \mid \forall x : X \bullet \exists_1 y : Y \bullet (x, y) \in f\}$ *ein totale Funktion,*

In Z werden einige Funktionen, die meist nur auf totalen Funktionen betrachtet werden, schon auf Relationen definiert. Dadurch können diese auch auf partiellen Funktionen angewandt werden. In dieser Arbeit wird dies bei der Funktion dom benötigt:

Definition 43 (dom) *Seien A und B Mengen und $\leftrightarrow$ eine Relation auf A und B, dann ist der Definitionsbereich $dom : (A \leftrightarrow B) \rightarrow \mathbb{P}\, A$ definiert als:*

$$\forall r : A \leftrightarrow B \bullet dom\, r = \{p : r \bullet first(p)\}$$

Relationen auf einer Menge können, wenn sie nicht reflexiv bzw. transitiv sind, erweitert werden, so dass sie die beiden Eigenschaften erfüllen.

Definition 44 (transitive und transitiv-reflexive Hülle)
Die transitive Hülle $M \leftrightarrow^+ M$ einer Relation $M \leftrightarrow M$ auf einer Menge M ist gegeben durch:

$$x \leftrightarrow^+ y :\Leftrightarrow \exists\, n \geq 0\, \exists\, x_1, \ldots, x_n \in M : x \leftrightarrow x_1 \leftrightarrow x_2 \leftrightarrow \cdots \leftrightarrow x_n \leftrightarrow y$$

Die reflexiv-transitive Hülle R^* ergibt sich durch:

$$x \leftrightarrow^* y :\Leftrightarrow x = y \vee x \leftrightarrow^+ y$$

A.3 Z: Abstrakter Datentyp

Die Verbindung der Daten mit der Definition aller zulässigen Operationen auf ihnen wird *Abstrakter Datentyp* (ADT) genannt. In Z wird dies durch die Bestandteile eines State-Schema für die Daten, einem Init-Schema für die Initialisierung und einer Menge von Operations-Schemata beschrieben. Grob gesehen, gleichen die jeweiligen Schematypen denen aus Object-Z. Der Hauptunterschied ist, dass Schamata in Z keine Delta-Liste besitzen. Dafür wird im Deklarationsteil das Schema durch direkte Angabe vollständig hinzugefügt. Dabei ist es möglich, ein Schema nur lesend einzubinden (angezeigt durch Ξ) oder alle Variablen des Schemas auch schreibend zu nutzen (Angabe eines Δ).

Definition 45 (Standart Z:ADT) *Ein Standrad Z Abstracter Datentyp (ADT) ist ein 3-Tupel* $(State, Init, \{Op_i\}_{i \in I})$ *wobei State ein State-Schema, Init ein Inizialisierungs-Schema und* $\{Op_i\}_{i \in I}$ *eine Menge von Operationsschemata die auf dem State arbeiten ist.*

A.4 Sequenzen

Sequenzen sind Folgen von Zeichen, Events oder anderen Elementen.

Definition 46 (Sequenz) *Sei A eine Menge.*

1. *Eine Sequenz* $s : seq\, A$ *ist eine Funktion* $s : \mathbb{N}_i \to A$.
2. *Die Länge der Sequenz s,* $\#s$*, ist definiert als* $\#s = i$.
3. *Eine Sequenz s kann geschrieben werden als* $s = \langle s(1), s(2) \ldots \rangle$.

Definition 47 (Konkatenation) *Seien* $s, t : seq\, A$ *zwei Sequenzen, dann ist die Konkatenation definiert als*

$$s \frown t := s \cup \{n : dom\, t \bullet n + \#s \mapsto t(n)\}.$$

Wenn $q \in A$*, dann ist*

$$s \frown q := s \frown \langle q \rangle$$

und

$$q \frown s := \langle q \rangle \frown s.$$

An einige Stellen werden in dieser Arbeit Funktionen auf Sequenzen genutzt, die direkt durch die Konkatenation ausgedrückt werden könnten, aber das Verständnis vereinfachen:

Definition 48 (Weitere Funktionen auf Sequenzen) *Seien A eine Menge und* $s : seq\, A$ *eine Sequenz über A mit* $s = \langle s_1, s_2 \ldots s_n \rangle$ *dann ist:*

- $head(s) = s_1$
- $last(s) = s_n$
- $tail(s) = \langle s_2 \ldots s_n \rangle$

In der Sprachtheorie werden Zeichenketten mit der Sequenz dieser Zeichen gleichgesetzt. Dabei wird für die Deklaration von Sequenzen eine alternative Notation genutzt, die es erlaubt, Aussagen über die Struktur der Sequenz zu definieren:

Definition 49 (Alternative Sequenzdeklarationen) *Seien A,B und C Mengen und b ein Element.*

1. $A^* := seq\, A$
2. $A^*B := \{s \frown t \mid s \in seq\, A;\ t \in B\}$
3. $AB^* := \{t \frown s \mid s \in seq\, A;\ t \in B\}$
4. $A^*BC^* := \{s \frown t \frown r \mid s \in seq\, A;\ t \in B;\ r \in seq\, C\}$
5. $A^*bC^* := \{s \frown b \frown r \mid s \in seq\, A;\ r \in seq\, C\}$

A.5 Quantorenschreibweise

Bei vielen CSP, CSP-OZ, Object-Z und Z Operatoren gibt es für eine wiederholte Operatorenanwendung Quantoren-Schreibweisen. Diese werden in Prinzip wie der Allquantor und der Existenzquantor in der Prädikatenlogik erster Ordnung gebildet. So kann der Schema-Konjunktion-Quantor durch einzelne Schema-Konjunktionen gebildet werden:

$$\bigwedge_{i \in \{i_1, i_2, \ldots, i_{max}\}} Op_i \quad := \quad Op_{i_1} \wedge Op_{i_1} \wedge \ldots \wedge Op_{i_{max}}$$

In der Inlineschreibweise wird oft auch einfach

$$\bigwedge i \in \{i_1, i_2, \ldots, i_{max}\} \mid Op_i := Op_{i_1} \wedge Op_{i_1} \wedge \ldots \wedge Op_{i_{max}}$$

genutzt.

Einige Operatoren, wie der generalized Parallel Operator in CSP, besitzen Parametrisierung, die als Suffix angegeben werden. In diesem Fall müssen die Parametrisierungen mit in den Quantoren angegeben werden. Dazu werden die Parametrisierungen von der Indexmenge durch ein $\bullet$ getrennt:

$$\Big\| i \in \{i_1, i_2, \ldots, i_{max}\} \bullet A_i \mid Q_i := Q_{i_1 A_{i_1}} \|_{A_{i_2}} Q_{i_2 A_{i_2}} \|_{A_{i_3}} \cdots {}_{A_{i_{max-1}}} \|_{A_{i_{max}}} Q_{i_{max}}.$$

Anhang B

Analysefunktionen

Analysefunktionen spielen eine wichtige Rolle in den Techniken in Re$\mathcal{L}$. In diesem Anhang sind die formalen Definitionen der in der Arbeit verwendeten Funktionen aufgelistet.

B.1 Analysefunktionen für FWHILE

Die Funktion usedFunc bestimmt die Menge der genutzten Funktionen in einem FWHILE-Programm.

usedFunc : $(STATS)* \to (NAMES, \mathbb{N})$

---

usedFunc$(MAIN\,(FUNC)*) = \text{usedFunc}(MAIN) \cup \bigcup_{f \in (FUNC)*} \text{usedFunc}(f)$

usedFunc(`"main"` $(MSTAT)*$ `"endmain"`) $= \bigcup_{s \in (MSTAT)*} \text{usedFunc}(s)$

usedFunc(`"func"` $FNAME$ `"("` $PARAM$ `")"` $(STAT)*$ `"endfunc"`)
$= \bigcup_{s \in (STAT)*} \text{usedFunc}(s)$

usedFunc(`"SKIP"`) $= \varnothing$

usedFunc(VAR `":="` $ITERM$ `";"`) $= \text{usedFunc}(ITERM)$

usedFunc(`"if"` $BTERM$ `"then"` $(STAT1)*$ (`"else"` $(STAT2)*$)? `"fi"`)
$= \text{usedFunc}(BTERM) \cup \bigcup_{s \in (STAT1)*} \text{usedFunc}(s) \cup \bigcup_{s \in (STAT2)*} \text{usedFunc}(s)$

usedFunc(`"while"` $BTERM$ `"do"` $(STAT)*$ `"od"`)
$= \text{usedFunc}(BTERM) \cup \bigcup_{s \in (STAT)*} \text{usedFunc}(s)$

usedFunc(`"return"` $ITERM$ `";"`) $= \text{usedFunc}(ITERM)$

usedFunc(`"return"` $BTERM$ `";"`) $= \text{usedFunc}(BTERM)$

usedFunc($(VAR \mid CONST \mid FUSE1)((\text{"+"}\mid\text{"-"})(VAR \mid CONST \mid FUSE2))*$)
$= \text{usedFunc}(FUSE1) \cup \bigcup_{f \in (FUSE2)*} \text{usedFunc}(f)$

usedFunc($ITERM1\ RELSYM\ ITERM2$)
$= \text{usedFunc}(ITERM1) \cup \text{usedFunc}(ITERM2)$

usedFunc($FNAME$ `"("` $PARAM$ `")"` $) = (FANAME, CountParam(PARAM))$

Die Menge der in einem FWHILE-Programm definierten Funktionen wird mit Hilfe der Funktion definedFunc berechnet:

definedFunc : $(STATS)* \to (NAMES, \mathbb{N})$

---

definedFunc$(MAIN\,(FUNC)*) = \bigcup_{f \in (FUNC)*} \text{usedFunc}(f)$

definedFunc(`"func"` $FNAME$ `"("` $PARAM$ `")"` $(STAT)*$ `"endfunc"`)
$= (FANAME, CountParam(PARAM))$

Die Analysefunktion calcReturn berechnet den Rückgabewert einer FWHILE-Funktion. Dazu wird die Typfunktion genutzt. Die Typfunktion kann nicht direkt als Analysefunktion verwendet werden, da sie nur auf wohlgetypten Termen definiert ist.

$$
\begin{array}{l}
\text{calcReturn} : \mathit{FUNC} \to \{\cap, \mathbb{B}, \mathit{ERR}\} \\
\hline
\text{calcReturn}(\ \texttt{"func"}\ \mathit{fname}\ \texttt{"("}\ \mathit{param}\ \texttt{")"}\ \mathit{statList}\ \texttt{"endfunc"}\ ) \\
\qquad = \mathit{ERR} \\
\qquad \text{für } \mathit{statList} \notin \operatorname{dom} \Sigma[\![.]\!]^{\tau} \\
\text{calcReturn}(\ \texttt{"func"}\ \mathit{fname}\ \texttt{"("}\ \mathit{param}\ \texttt{")"}\ \mathit{statList}\ \texttt{"endfunc"}\ ) \\
\qquad = \mathbf{int} \\
\qquad \text{für } \Sigma[\![\mathit{statList}]\!]^{\tau} = \mathbf{int} \\
\text{calcReturn}(\ \texttt{"func"}\ \mathit{fname}\ \texttt{"("}\ \mathit{param}\ \texttt{")"}\ \mathit{statList}\ \texttt{"endfunc"}\ ) \\
\qquad = \mathbf{bool} \\
\qquad \text{für } \Sigma[\![\mathit{statList}]\!]^{\tau} = \mathbf{bool} \\
\text{calcReturn}(\ \texttt{"func"}\ \mathit{fname}\ \texttt{"("}\ \mathit{param}\ \texttt{")"}\ \mathit{statList}\ \texttt{"endfunc"}\ ) \\
\qquad = \mathit{ERR} \\
\qquad \text{für } \Sigma[\![\mathit{statList}]\!]^{\tau} = \mathit{void}
\end{array}
$$

Funktionen werden in FWHILE durch ihren Namen und die Anzahl der Parameter angesprochen. Dies ist möglich, da nur **int** -Werte durch Variablen übergeben werden können. Es wird daher eine Analysefunktion benötigt, die die Parameter einer Parameterliste zählt.

$$
\begin{array}{l}
\text{CountParam} : \mathit{PARAM} \to \mathbb{N} \\
\hline
\text{CountParam}() = 0 \\
\text{CountParam}(\mathit{VAR}) = 1 \\
\text{CountParam}(\mathit{VAR}(\text{,}\ \mathit{VAR}1)*) = 1 + \#\mathit{VAR}1
\end{array}
$$

Variablen in FWHILE können geschrieben und gelesen werden. Die Funktion writevars bestimmt die Menge der Variablen deren Wert in dem gegeben Term geändert werden. Die Funktion readvars entsprechen die Menge der gelesenen Variablen.

$$
\begin{array}{l}
\text{writevars} : (\mathit{STAT})* \to \mathbb{P}\ \mathit{VAR} \\
\hline
\text{writevars}(\mathit{stat}\ \ \mathit{statList}1) = \text{writevars}(\mathit{stat}) \cup \text{writevars}(\mathit{statList}1) \\
\text{writevars}(\ \texttt{"SKIP"}\ ) = \varnothing \\
\text{writevars}(\ \texttt{"return"}\ \mathit{iterm}\ \texttt{";"}\ ) = \varnothing \\
\text{writevars}(\ \texttt{"return"}\ \mathit{bterm}\ \texttt{";"}\ ) = \varnothing \\
\text{writevars}(\mathit{var}\ \texttt{":="}\ \mathit{iterm}\ \texttt{";"}\ ) = \{\mathit{var}\} \\
\text{writevars}(\ \texttt{"if"}\ \mathit{bterm}\ \texttt{"then"}\ \mathit{statList}1\ \texttt{"else"}\ \mathit{statList}2\ \texttt{"fi"}\ ) \\
\qquad\qquad = \text{writevars}(\mathit{statList}1) \cup \text{writevars}(\mathit{statList}2) \\
\text{writevars}(\ \texttt{"if"}\ \mathit{bterm}\ \texttt{"then"}\ \mathit{statList}1\ \texttt{"fi"}\ ) = \text{writevars}(\mathit{statList}1) \\
\text{writevars}(\ \texttt{"while"}\ \mathit{bterm}\ \texttt{"do"}\ \mathit{statList}1\ \texttt{"od"}\ ) = \text{writevars}(\mathit{statList}1)
\end{array}
$$

Die Funktion readvars ist in mehrere Teile aufgespalten die verschiedene Arten von Termen untersucht:

$$
\begin{array}{l}
\text{readvars} : (STAT)* \to \mathbb{P}\ VAR \\
\hline
\text{readvars}(stat\ statList1) = \text{readvars}(stat) \cup \text{readvars}(statList1) \\
\text{readvars}(\texttt{"SKIP"}) = \varnothing \\
\text{readvars}(\texttt{"return"}\ iterm\ \texttt{";"}) = \text{readvars}(iterm) \\
\text{readvars}(\texttt{"return"}\ bterm\ \texttt{";"}) = \text{readvars}(bterm) \\
\text{readvars}(var\ \texttt{":="}\ iterm\ \texttt{";"}) = \text{readvars}(iterm) \\
\text{readvars}(\texttt{"if"}\ bterm\ \texttt{"then"}\ statList1\ \texttt{"else"}\ statList2\ \texttt{"fi"}) \\
\qquad = \text{readvars}(bterm) \cup \text{readvars}(statList1) \cup \text{readvars}(statList2) \\
\text{readvars}(\texttt{"if"}\ bterm\ \texttt{"then"}\ statList1\ \texttt{"fi"}) \\
\qquad = \text{readvars}(bterm) \cup \text{readvars}(statList1) \\
\text{readvars}(\texttt{"while"}\ bterm\ \texttt{"do"}\ statList1\ \texttt{"od"}) \\
\qquad = \text{readvars}(bterm) \cup \text{readvars}(statList1)
\end{array}
$$

$$
\begin{array}{l}
\text{readvars} : BTERM \to \mathbb{P}\ VAR \\
\hline
\text{readvars}(iterm1\ relsym\ iterm2) = \text{readvars}(iterm1) \cup \text{readvars}(iterm2) \\
\text{readvars}(fname\ var1\ \texttt{","}\ \ldots\ \texttt{","}\ vari) = \{var1, \ldots, vari\}
\end{array}
$$

$$
\begin{array}{l}
\text{readvars} : ITERM \to \mathbb{P}\ VAR \\
\hline
\text{readvars}(var) = \{var\} \\
\text{readvars}(const) = \varnothing \\
\text{readvars}(fname\ var1\ \texttt{","}\ \ldots\ \texttt{","}\ vari) = \{var1, \ldots, vari\} \\
\text{readvars}(iterm1\ \texttt{"+"}\ iterm2) = \text{readvars}(iterm1) \cup \text{readvars}(iterm2) \\
\text{readvars}(iterm1\ \texttt{"-"}\ iterm2) = \text{readvars}(iterm1) \cup \text{readvars}(iterm2)
\end{array}
$$

B.2 Typfunktion für FWHILE

In diesem Abschnitt wird die Typfunktion für FWHILE definiert:

$$\Sigma[\![.]\!]^\tau : FWHILE \to \{void, \mathbf{bool}\ , \mathbf{int}\ \}$$

Typing der Integerausdrücken wird definiert durch:

$$\Sigma[\![VAR]\!]^\tau = \mathbf{int}$$
$$\Sigma[\![CONST]\!]^\tau = \mathbf{int}$$
$$\Sigma[\![X + Y]\!]^\tau = \mathbf{int} \quad \text{für } \Sigma[\![X]\!]^\tau = \Sigma[\![Y]\!]^\tau = \mathbf{int}$$
$$\Sigma[\![X - Y]\!]^\tau = \mathbf{int} \quad \text{für } \Sigma[\![X]\!]^\tau = \Sigma[\![Y]\!]^\tau = \mathbf{int}$$
$$\Sigma[\![fname\ \texttt{"("}\ param\ \texttt{")"}\ )]\!]^\tau = [\![second([\![funcList]\!]^F(fname, count(param)))]\!]^\tau$$

Dabei ist die Funktion $[\![funcList]\!]^F$ die gleiche wie für die Semantik von FWHILE auf Seite 24 genutzt wird. *funcList* ist wie bei der Semantik der Code aller definierten Funktionen.

Vergleiche sind immer vom Typ **bool** :

$$\Sigma[\![(ITERM(\ \texttt{"="}\ \mid\ \texttt{"!="}\ \mid\ \texttt{"<"}\ \mid\ \texttt{">"}\ )ITERM))]\!]^\tau = \mathbf{bool}$$

Damit bleiben nur noch die Typen der verschiedenen Statements anzugeben:

$$\Sigma[\![(STAT)*]\!]^\tau = \mathbf{int} \quad \text{für } \exists\, x \in (STAT) *\bullet [\![x]\!]^\tau = \mathbf{int} \wedge \forall\, x \in (STAT)* \bullet [\![x]\!]^\tau \neq \mathbf{bool}$$
$$\Sigma[\![(STAT)*]\!]^\tau = \mathbf{bool} \quad \text{für } \exists\, x \in (STAT) *\bullet [\![x]\!]^\tau = \mathbf{bool} \wedge \forall\, x \in (STAT)* \bullet [\![x]\!]^\tau \neq \mathbf{int}$$
$$\Sigma[\![(STAT)*]\!]^\tau = void \text{ für } \forall\, x \in (STAT) *\bullet [\![x]\!]^\tau \neq \mathbf{bool} \wedge \forall\, x \in (STAT)* \bullet [\![x]\!]^\tau \neq \mathbf{int}$$
$$\Sigma[\![\ \texttt{"SKIP"}\ ]\!]^\tau = void$$
$$\Sigma[\![VAR\ \texttt{":="}\ ITERM\ \texttt{";"}\ )]\!]^\tau = void$$
$$\Sigma[\![\ \texttt{"if"}\ BTERM\ \texttt{"then"}\ (STAT1)*\ \texttt{"else"}\ (STAT2)*\ \texttt{"fi"}\ ]\!]^\tau = \mathbf{bool} \quad \text{für } [\![(STAT1)*]\!]^\tau = \mathbf{bool} \ \vee\ [\![(STAT2)*]\!]^\tau = \mathbf{bool}$$
$$\Sigma[\![\ \texttt{"if"}\ BTERM\ \texttt{"then"}\ (STAT1)*\ \texttt{"else"}\ (STAT2)*\ \texttt{"fi"}\ )]\!]^\tau = \mathbf{int} \quad \text{für } [\![(STAT1)*]\!]^\tau = \mathbf{int} \ \vee\ [\![(STAT2)*]\!]^\tau = \mathbf{int}$$
$$\Sigma[\![\ \texttt{"if"}\ BTERM\ \texttt{"then"}\ (STAT1)*\ \texttt{"else"}\ (STAT2)*\ \texttt{"fi"}\ )]\!]^\tau = void \text{ sonst}$$
$$\Sigma[\![\ \texttt{"if"}\ BTERM\ \texttt{"then"}\ (STAT)*\ \texttt{"else"}\ (STAT)*\ \texttt{"fi"}\ )]\!]^\tau = void$$
$$\Sigma[\![\ \texttt{"while"}\ BTERM\ \texttt{"do"}\ (STAT)*\ \texttt{"od"}\ )]\!]^\tau = void$$
$$\Sigma[\![MSTAT]\!]^\tau = void$$
$$\Sigma[\![\ \texttt{"return"}\ ITERM\ \texttt{";"}\ ))]\!]^\tau = \mathbf{int}$$
$$\Sigma[\![\ \texttt{"return"}\ BTERM\ \texttt{";"}\ ))]\!]^\tau = \mathbf{bool}$$

B.3 Analysefunktionen für CSP-OZ

CSP-OZ hat eine sehr große Grammatik. Daher werden für CSP-OZ nicht die vollständigen Analysefunktionen angegeben, sondern nur die benötigten Teile. Da die meisten Analysefunktionen den semantischen Hilfsfunktionen aus [Smi00] und [Fis00] entsprechen, kann die vollständige Definition dort nachgeschaltet werden. Dies ist auch die Quelle für die Definition der Funktionen.

Variablen Die Funktion **vars** bestimmt die Variablen einer Struktur, wie beispielsweise von einem Schema. In dieser Arbeit wird die Funktion nur auf den Schemata direkt verwendet. Im Gegensatz zu [Smi00] wird eine Variable als Paar aus dem Namen und dem Typ der Variablen beschrieben.

$$\begin{aligned}
&\mathbf{vars}([d \mid p]) = \mathbf{vars}(d)\\
&\mathbf{vars}(d_1;\ d_2) = \mathbf{vars}(d_1) \cup \mathbf{vars}(d_2)\\
&\mathbf{vars}(x_1, x_2, \ldots, x_n : T) = \{(x_1, T), (x_2, T), \ldots, (x_n, T)\}\\
&\mathbf{vars}(s_1 \wedge s_2) = \mathbf{vars}(s_1) \cup \mathbf{vars}(s_2)
\end{aligned}$$

Die Funktionen der Ausgabe-, Eingabe- und Simple-Variablen lassen sich durch die **vars**-Funktion darstellen. Dabei wird genutzt, dass sich Paare als Relationen auffassen lassen. Das ermöglicht es durch den dom-Operator alle Namen aus der Variablenmenge zu extrahieren:

$$\begin{aligned}
&\mathbf{input}(x) = \{y \mid y? \in \operatorname{dom} \mathbf{vars}(x)\}\\
&\mathbf{output}(x) = \{y \mid y! \in \operatorname{dom} \mathbf{vars}(x)\}\\
&\mathbf{simple}(x) = \mathbf{vars}(x) \setminus (\mathbf{output}(x) \cup \mathbf{input}(x))
\end{aligned}$$

Die Funktion **delta** gibt die delta-Variablen eines resultierenden Schema zurück:

$$\begin{aligned}
&\mathbf{delta}([\Delta(x_1, x_2..., x_n)\, d \mid p]) = \{x_1, x_2, \ldots, x_n\}\\
&\mathbf{delta}(s_1 \wedge s_2) = \mathbf{delta}(s_1) \cup \mathbf{delta}(s_2)
\end{aligned}$$

Ähnlich wie die Variablen werden die Kanäle einer CSP-OZ-Klasse berechnet. Die Kanäle werden als Triple dargestellt. Wobei $\mathcal{NAME}$ die Menge der mögliche Namen ist, $\mathcal{CT}$ die Menge der Typen und $\mathcal{ART}$ angibt von welcher Art, wie `chan` oder `method`, ein Kanal ist:

$$chans \in \mathcal{NAME} \times \mathcal{CT} \times \mathcal{ART}$$

Die Kanäle des Interface einer CSP-OZ-Klasse können wie folgt berechnet werden. Da die benötigten Variablen bisher nicht definiert sind, werden sie hier mit angegeben:

$$\begin{aligned}
&c_1, c_2 : \mathit{Identifier}\\
&T : \mathcal{CT}\\
&\mathit{chandef}_1, \mathit{chandef}_2 : \mathit{InterfaceRow}\\
&\mathbf{channels}(\mathtt{chan}\, c_1, c_2, \ldots, c_j : T) = \{(c_1, T, \mathtt{chan}), (c_2, T, \mathtt{chan}), \ldots, (c_2, T, \mathtt{chan})\}\\
&\mathbf{channels}(\mathtt{method}\, c_1, c_2, \ldots, c_j : T)\\
&\qquad\qquad = \{(c_1, T, \mathtt{method}), (c_2, T, \mathtt{method}), \ldots, (c_2, T, \mathtt{method})\}\\
&\mathbf{channels}(\mathit{chandef}_1;\ \mathit{chandef}_2) = \mathbf{channels}(\mathit{chandef}_1) \cup \mathbf{channels}(\mathit{chandef}_2)
\end{aligned}$$

Bei vielen Refactorings werden die Bezeichner von Operationen, Variablen, Klassen oder Kanälen benötigt. Die Funktion *name* wird für die verschiedenen Konstrukte verwendet.

Die Definition für Variablen findet sich in Kapitel 6.3. Hier soll nun die Berechnung für Operationen ergänzt werden. Da die Menge aller Operationen in einer CSP-OZ-Spezifikation direkt durch Meta-Variablen bestimmt werden kann, ist keine Funktion zur Berechnung der Menge der Operationen nötig. Daher werden die Operationsnamen direkt auf der einer Operation bestimmt:

$$name(op \mathrel{\widehat=} opdef) = op$$

Die Namen einer Menge von Operationen kann dann als Menge der Namen der Einzeloperationen bestimmt werden. Dabei wird ausgenutzt, dass sich in einem Binding einer Meta-Variablen eine Funktion befindet. Sei *ops* eine Meta-Variable in der sich eine Menge von Operationen befindet:

$$\mathbf{names}(ops) = \{name(op) \mid op \in \operatorname{ran} ops\}$$

Die entsprechende Funktion auf den Variablen ist simple, da sie eine Relation aus Namen und Typ bilden (vgl. Kapitel 6.3). Sei $vars : \mathbb{P}\mathcal{NAME} \times \mathcal{T}$ eine Menge von Variablen:

$$\mathbf{names}(vars) = \operatorname{dom} vars$$

Für Kanäle ist die Definition analog zu den Variablen:

$$\mathbf{names}(chans) = \operatorname{dom} chans$$

Die letzte Funktion berechnet für einen CSP-Term die Menge seiner Events:

$$\begin{array}{|l}
\mathbf{events} : CSP \to \mathbb{P}\, IDENTIFIERER \\
\hline
\mathbf{events}(x \to P) = \{x\} \cup \mathbf{events}(P) \\
\mathbf{events}(\texttt{Skip}) = \varnothing \\
\mathbf{events}(\texttt{Stop}) = \varnothing \\
\mathbf{events}(P \,{}_A\|_B\, Q) = \mathbf{events}(P) \cup \mathbf{events}(Q) \\
\mathbf{events}(P \underset{A}{\|} Q) = \mathbf{events}(P) \cup \mathbf{events}(Q) \\
\mathbf{events}(P \underset{A}{|||} Q) = \mathbf{events}(P) \cup \mathbf{events}(Q) \\
\mathbf{events}(P \,\Box\, Q) = \mathbf{events}(P) \cup \mathbf{events}(Q) \\
\mathbf{events}(P \sqcap Q) = \mathbf{events}(P) \cup \mathbf{events}(Q)
\end{array}$$

Anhang C

CSP-OZ: Grammatik

```
CompilationUnit  =  (<LATEX>)* (Paragraph (<LATEX>)*)*
Paragraph        =  <ZEDBEGIN> (BasicTypeDefinition                                  2
                              | AbbreviationDefinition
                              | FreeTypeDefinition)
                              (<CR> (BasicTypeDefinition
                                     | AbbreviationDefinition
                                     | FreeTypeDefinition))*                       7
                    <ZEDEND>
                  | <CSPBEGIN> CSPZ <CSPEND>
                  | AxiomaticDefinition
                  | GenericDefinition
                  | Schema0                                                        12
                  | Class0
BasicTypeDefinition  =  "[" IdentifierDefinition
                            ("," IdentifierDefinition)*
                        "]"
AxiomaticDefinition  =  <AXDEFBEGIN> Declaration                                   17
                            (<CR> Declaration)*
                            (<ST> PredicateList)?
                        <AXDEFEND>
GenericDefinition    =  <GENERICBEGIN>
                            (FormalParameters)? Declaration                        22
                            (<CR> Declaration)* (<ST> PredicateList)?
                        <GENERICEND>
AbbreviationDefinition  =  Abbreviation <DEFEQ> Expression0
Abbreviation         =  (VariableName (CSPParameters)?
                           | PrefixGenericName IdentifierDefinition                27
                           | IdentifierDefinition
                                  InfixGenericName IdentifierDefinition)
FreeTypeDefinition   =  IdentifierDefinition <TYPEDEF>
                            Branch (<HBAR> Branch)*
Branch               =    IdentifierDefinition                                     32
                        | VariableName "<<" Expression0 ">>"
Schema0              =  (<SCHEMABEGIN>
                             <LLBRACE> SchemaHeader <LRBRACE>
                             Declaration (<CR> Declaration)*
                             (<ST> PredicateList)?                                 37
                          <SCHEMAEND>
                        | SchemaHeader <SDEF> SchemaExpression)
SchemaHeader         =  SchemaName (FormalParameters)?
Class0               =  <CLASSBEGIN>
                            <LBRACE>                                               42
                                  ClassName (FormalParameters)? (CSPParameters)?
                            <RBRACE>
                            (VisibilityList | Interface)?
                            (InheritedClass)*
                            (LocalDefinition)*                                     47
                            (<CSPBEGIN> CSPZ <CSPEND>)?
```

```
                          (StateO)?
                          (InitialState)?
                          (OperationO | OperationEnable)*
                       <CLASSEND>                                                    52
SchemaName          =  <WORD>
ClassName           =  <WORD>
                             (<SUFFIX>
                                  (<LBRACE> (<WORD> | <INT>) <RBRACE>
                                  | (<WORD> | <INT>)))*                            57
FormalParameters    =  "[" (IdentifierDefinition)+ "]"
CSPParameters       =  "(" Declaration ")"
VisibilityList      =  <HARP> "(" DeclarationNameList ")"
InheritedClass      =  (<INHERITED> | <INHERITINTERFACE>) ClassName
                                  (ActualParameters)? (RenameList)?                62
LocalDefinition     =    ":" BasicTypeDefinition
                         | AxiomaticDefinition
                         | AbbreviationDefinition
                         | FreeTypeDefinition)
StateO              =    <STATEBEGIN>                                             67
                                 (Declaration (<CR> Declaration)*
                                 (<DELTA> Declaration)?
                                 (<ST> PredicateList)?
                               | <DELTA> Declaration
                                 (<CR> Declaration)*                              72
                                 (<ST> PredicateList)?
                                 | PredicateList)
                           <STATEEND>
                         | "["
                                 (Declaration (<DELTA> Declaration)?               77
                                 (<HBAR> PredicateList)?
                               | <DELTA> Declaration
                                    (<HBAR> PredicateList)?
                                    | PredicateList) "]"
InitialState        =    <INITBEGIN>                                              82
                              PredicateList
                           <INITEND>
                         | <INIT> <SDEF> "[" PredicateList "]"
OperationO          =    <OPBEGIN> <LBRACE> IdentifierUse <RBRACE>
                                 (DeltaList (<CR> Declaration)*                   87
                                           (<ST> PredicateList)?
                                 | Declaration (<CR> Declaration)*
                                              (<ST> PredicateList)?
                                 | PredicateList)
                              <OPEND>                                             92
                            | (<EFFECT> | <COM>)?
                                  IdentifierUse <SDEF> OperationExpression <CR>
OperationEnable     =  <OPENBEGIN> <LBRACE> IdentifierUse <RBRACE>
                                 (DeltaList (<CR> Declaration)*
                                 (<ST> PredicateList)?                            97
                               | Declaration (<CR> Declaration)*
                                 (<ST> PredicateList)?
                               | PredicateList)
                           <OPENEND>
                         | <ENABLE> IdentifierUse <SDEF> OperationExpression <CR>  102
DeltaList           =  <DELTA> "(" DeclarationNameList ")"
OperationExpression =  (<ZAND> Declaration
                                 (<HBAR> PredicateO <BULLET> OperationExpression)?
                         | <MULTCHOICE> Declaration
                                 (<HBAR> PredicateO <BULLET> OperationExpression)? 107
                         | <COMP> Declaration
                                 (<HBAR> PredicateO <BULLET> OperationExpression)?
                         | OperationExpression1)
OperationExpression1 =  OperationExpression2
                              (<BSLASH> "(" DeclarationNameList ")"                112
                              | <ZAND> OperationExpression1
                              | <PARALLEL> OperationExpression1
                              | <PARALELMARK> OperationExpression1
```

```
                          | <CHOICE> OperationExpression1
                          | <COMP> OperationExpression1                          117
                          | <BULLET> OperationExpression1)?
OperationExpression2  =  ("[" (DeltaList (<CR> Declaration)* (<ST> PredicateList)?
                                   | Declaration (<CR> Declaration)*
                                                 (<ST> PredicateList)?
                                   | PredicateList) "]"                          122
                                   | Expression0 (RenameList)?
                                   | "(" OperationExpression ")")
SchemaExpression      =  (<FORALL> SchemaText <BULLET> SchemaExpression)
                          | (<EXISTS> SchemaText <BULLET> SchemaExpression)
                          | (<EXISTSONE> SchemaText <BULLET> SchemaExpression)  127
                          | SchemaExpression1
SchemaExpression1     =  SchemaExpression2 (<ZAND> SchemaExpression1
                                   | <ZOR> SchemaExpression1
                                   | <SIMPL> SchemaExpression1
                                   | <SAQUIV> SchemaExpression1                  132
                                   | <HARP> SchemaExpression1
                                   | <BSLASH> SchemaExpression1
                                   | <COMP> SchemaExpression1
                                   | <IMPCOD> SchemaExpression1)?
SchemaExpression2     =  SchemaText                                             137
                          | SchemaReference
                          | <ZNOT> SchemaExpression2
                          | <PRE> SchemaExpression2
                          | "(" SchemaExpression ")"
SchemaText            =  Declaration (<HBAR> Predicate0)?                       142
SchemaReference       =  SchemaReference1 (RenameList)?
SchemaReference0      =    SchemaReference
                           | Expression6 "." SchemaReference1 (RenameList)?
SchemaReference1      =  SchemaName Decoration (ActualParameters)?
RenameList            =  "[" (RenameItem ("," RenameItem)* | Expr) "]"          147
RenameItem            =  DeclarationName <SLASH> DeclarationName
ActualParameters      =  "[" Expression0 ("," Expression0)* "]"
Declaration           =  BasicDeclaration (<SEMICOLON> BasicDeclaration)*
BasicDeclaration      =    DeclarationNameList ":" Type
                           | SchemaReference                                    152
Type                  =  Expression0
DeclarationNameList   =  DeclarationName ("," DeclarationName)*
DeclarationName       =    IdentifierDefinition
                           | OperatorName
PredicateList         =  Predicate0 ((<SEMICOLON> | <CR>) (PredicateList)?)?    157
Predicate0            =  <FORALL> SchemaText
                                  <BULLET> Predicate0
                         | <EXISTS> SchemaText
                                  <BULLET> Predicate0
                         | <EXISTSONE> SchemaText                               162
                                  <BULLET> Predicate0
                         | <LET> LetDefinition (<SEMICOLON> LetDefinition)*
                                  <BULLET> Predicate0
                         | <IF> Predicate0 <THEN> Predicate0
                                       (<ELSE> Predicate0)?                     167
                         | Predicate1
Predicate1            =  Predicate2 (<LAND> Predicate1
                                  | <LOR> Predicate1
                                  | <LIMP> Predicate1
                                  | <LAQU> Predicate1)?                         172
Predicate2            =  Expression1 ("." <INIT>
                                   | (Relation0 Expression0)*)
                           | PrefixRelationName Expression0
                           | <PRE> SchemaReference
                           | <TRUE>                                             177
                           | <FALSE>
                           | <LNOT> Predicate2
                           | "(" Predicate0 ")"
Relation0             =    <EQUALS>
                           | <INSET>                                            182
```

```
                          | <REL> <LBRACE> IdentifierUse <RBRACE>
                          | InfixRelationName
LetDefinition          =  VariableName <DEFEQ> Expression0
ExpressionZ            =    <LAMDA> SchemaText (<BULLET> Expression0)?
                          | <MY> SchemaText <BULLET> Expression0                    187
                          | <LET> LetDefinition (<SEMICOLON> LetDefinition)*
                                                <BULLET> Expression0
                          | Expression0
Expression0            =    <IF> Predicate0 <THEN> Expression0 <ELSE> Expression0
                          | Expression1                                             192
Expression1            =  Expression1a (InfixGenericName Expression1)?
Expression1a           =  Expression2 (<CROSS> Expression2)*
Expression2            =  Expression2a (InfixFunctionName Decoration Expression2)?
Expression2a           =  PrefixGenericName Expression4
                          | ("+" | "-" | <NEAGTE>) Decoration Expression4           197
                          | Expression4 FunctionOrRelProjectionExpression
FunctionOrRelProjectionExpression  =  "(" Expression0 ("," Expression0)* ")"
                          | (<RELPROJBEGIN> ExpressionZ <RELPROJEND> Decoration)*
Expression4            =  (<SUBCLASS>)? Expression4a
Expression4a           =  Expression5 (<UNION> Expression5)*                        202
Expression5            =  Expression6 ("." Identifier)?
Expression6            =  OpenRefOrVariable (ActualParameters)?
                          | SchemaReference
                          | ClassRef
                          | <SELF>                                                  207
                          | <INT>
                          | <TRUE>
                          | <FALSE>
                          | SetExpression
                          | <SEQUENZBEGIN> (Expression0)* <SEQUENZEND>              212
                          | <BAGBEGIN> (Expression0)+ <BAGEND>
                          | <TEHTA> SchemaName Decoration (RenameList)?
                          | "(" Expression0 ")"
                          | PrefixFunctionSymbol Expression6
PrefixFunctionSymbol   =     <HASH>                                                 217
                          | <FIRST>
                          | <SECONDE>
                          | <RAN>
                          | <DOM>
                          | <DISJOINT>                                            222
ClassRef               =  ClassName (ActualParameters)?
                                    (RenameList)?
                                    (<CONTAINMENT>)?
SetExpression          =    <LLBRACE>
                                   (SetLiteralList                                 227
                                   | SchemaExpression (<BULLET> Expression0)?)
                            <LRBRACE>
                          | <EXPRSET>
                               <LBRACE> Expression0 ("," Expression0)* <RBRACE>
                          | <EMPTYSET>                                            232
                          | <NAT>
                          | <NATONE>
                          | <INTEGER>
                          | <BOOL>
                          | <OID>                                                 237
                          | <ESET> <LBRACE>
                                      ( Expression0 ("," Expression0)*
                                      | SchemaExpression (<BULLET> Expression0)?)
                                   <RBRACE>
SetLiteralList         =  SetLiterals ("," SetLiterals)*                           242
SetLiterals            =  (<WORD> | <INT>) (InfixFunctionName (<WORD> | <INT>))?
OpenRefOrVariable      =  IdentifierUse | "(" OperatorName ")"
VariableName           =  IdentifierDefinition | "(" OperatorName ")"
Identifier             =  <WORD>
                              (<SUFFIX> (<LBRACE> (<WORD> | <INT>) <RBRACE>       247
                              | (<WORD> | <INT>)))*
                            Decoration
```

```
OperatorName          =    InfixFunctionName
                           | InfixGenericName
                           | InfixRelationName                                  252
                           | PrefixGenericName
                           | PrefixRelationName
                           | PostfixFunctionName
InfixFunctionName     =    "-"
                           | "+"                                                257
                           | <UNION>
                           | <MAP>
                           | <MAPSTO>
                           | <DOTDOT>
                           | <UPTO>                                             262
                           | <SETMINUS>
                           | <CONCAT>
                           | <FILTERING>
                           | <EXTRACT>
                           | <BUNI>                                             267
                           | <INTERSECTION>
                           | <STAR>
                           | <MATHDIV>
                           | <MATHMOD>
                           | <PROJECTION>                                       272
                           | <IRES>
                           | <PARTION>
InfixGenericName      =    <RELA>
                           | <TFUN>
                           | <FUN>                                              277
                           | <PFUN>
                           | <PINJ>
                           | <INJ>
                           | <TINJ>
                           | <PSUR>                                             282
                           | <PSURJ>
                           | <TSUR>
                           | <TSURJ>
                           | <BIJ>
                           | <FFUN>                                             287
                           | <FINJ>
InfixRelationName      =    <NEQ>
                           | <NOTIN>
                           | <NEM>
                           | <NMEM>                                             292
                           | <SUBSET>
                           | <SUBSETEQ>
                           | <SUPSET>
                           | <SUPSETEQ>
                           | <LEQ>                                              297
                           | <LE>
                           | <GEQ>
                           | <GE>
                           | <ZPREFIX>
                           | <ZSUFFIX>                                          302
                           | <INBAG>
PrefixGenericName      =    <PSET>
                           | <POWER>
                           | <ID>
                           | <FSET>                                             307
                           | <FINSET>
                           | <FSETONE>
                           | <PSETONE>
                           | <SEQ>
                           | <SEQONE>                                           312
                           | <ISEQ>
                           | <BAG>
PrefixRelationName     =    <DISJOINT>
PostfixFunctionName    =    <TANSCOLSURE>
```

```
                          | <RELEXTRANSCLOSUSURE>                                          317
                          | <RELATIONALINVERSE>
Decoration             =  ((<STOKE>)+ | <INDEC> | <OUTDEC>)?
CSPZ                   =  (ProzessIdentifier <CDEF> ProcExpr <CR>)+
Expr                   = <IF> Predicate0 <THEN> Expr <ELSE> Expr | Expression1
ProcExpr               =  ProcExpr1                                                   322
ProcExpr1              =   <QINTCHOISE> SchemaText <BULLET> ProcExpr2
                          | <QEXTCHOISE> SchemaText <BULLET> ProcExpr2
                          | <QPARR> ("[" Expr "]" SchemaText <BULLET> ProcExpr2
                                   | SchemaText <BULLET> "[" Expr "]" ProcExpr2)
                          | <QINTERLEAVE> SchemaText <BULLET> ProcExpr2                  327
                          | <QCOMP> SchemaText <BULLET> ProcExpr2
                          | <QLParr> <LLBRACE> Expr <LRBRACE> SchemaText
                                                        <BULLET> ProcExpr1
                          | ProcExpr2
ProcExpr2              =  ProcExpr3                                                   332
                            (<EXTCHOICE> ProcExpr2
                              | <INTCHOICE> ProcExpr2
                              | <COMP> ProcExpr2
                              | <PARALLEL> "[" Expr "]" ("[" Expr "]")? ProcExpr2
                              | <INTERLEAVE> ProcExpr2                               337
                              | <LPARALLEL> <LBRACE> Expr <RBRACE> ProcExpr2
                              | <INTERRUPT> ProcExpr2
                              | <HIDING> Expr
                              | RenameList)?
ProcExpr3              =  (Predicate0 <GUARD>)? ProcExpr4                             342
ProcExpr4              =   "(" ProcExpr ")"
                           | <STOP>
                           | <CSPSKIP>
                           | <CHAOS> "(" Expr ")"
                           | <DIVER>                                                  347
                           | <MAIN> ("(" Expr ("," Expr)* ")")?
                           | "[" SchemaText "]" <PREFIX> ProcExpr
                           | Prefix
                           | Expr <PREFIX> ProcExpr
                           | Call                                                     352
Call                   =  <WORD>
Prefix                 =  <WORD> ((FieldExpr)+ <PREFIX> ProcExpr
                                 | "(" Expr ("," Expr)* ")" (Expr)?)
FieldExpr              =    <OUTDEC> (SchemaText
                                      | <WORD>                                      357
                                      | <INT>
                                      | "(" Expr ")")
                           | "." (SchemaText
                                 | <WORD>
                                 | <INT>                                            362
                                 | "(" Expr ")")
                           | <INDEC> ("[" SchemaText "]" | <WORD>)
ProzessIdentifier      =  <WORD> ("(" SchemaText ")")? | <MAIN>
ProcCall               =  <WORD> "(" Expr ")"
Interface              =  (InterfaceRow <CR>)+                                        367
                           | <INTERFACEBEGIN> (InterfaceRow <CR>)* <INTERFACEEND>
InterfaceRow           =  (<chan> ("[" <INT> "]")?
                           | <method> ("[" <INT> "]")?
                           | <localchan> ("[" <INT> "]")?)
                           DeclarationNameList (":" "[" SchemaExpression "]")?        372
IdentifierDefinition   =  Identifier
IdentifierUse          =  Identifier
```

Literaturverzeichnis

[Abr96] J. R. Abrial. *The B-book: assigning programs to meanings.* Cambridge University Press, New York, NY, USA, 1996.

[AF08] Aharon Abadi and Yishai A Feldman. Re-Approaching the Refactoring Rubicon. In *Proceedings of the 2nd Workshop on Refactoring Tools*, WRT '08, pages 10:1–10:4. ACM, 2008.

[AMST09] Thorsten Arendt, Florian Mantz, Lars Schneider, and Gabriele Taentzer. Model Refactoring in Eclipse by LTK , EWL , and EMF Refactor : A Case Study. Technical report, Philipps-Universität Marburg, 2009.

[AS88] Klaus Alber and Werner Struckmann. *Einführung in die Semantik von Programmiersprachen.* Number 3-411-03182-4. BI-Wiss.-Verl., 1988.

[Bal99] Helmut Balzert. *Lehrbuch Grundlagen der Informatik*, chapter zur Psychologie des Programmierens, pages 554–558. Spectrum Akademischer Verlag, 1999.

[Ber91] Paul L. Bergstein. Object-preserving class transformations. *SIGPLAN Not.*, 26:299–313, November 1991.

[BHE08] Dénes Bisztray, Reiko Heckel, and Hartmut Ehrig. Verification of Architectural Refactorings by Rule Extraction. In José Fiadeiro and Paola Inverardi, editors, *Fundamental Approaches to Software Engineering*, volume 4961 of *Lecture Notes in Computer Science*, pages 347–361. Springer Berlin / Heidelberg, 2008.

[BHM09] Artur Boronat, Reiko Heckel, and José Meseguer. Rewriting logic semantics and verification of model transformations. In Marsha Chechik and Martin Wirsing, editors, *FASE*, volume 5503 of *LNCS*, pages 18–33, Berlin, Heidelberg, 2009. Springer-Verlag.

[Bis08] Dénes Bisztray. Verification of Architectural Refactorings: Rule Extraction and Tool Support. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *Graph Transformations*, volume 5214 of *Lecture Notes in Computer Science*, pages 475–477. Springer Berlin / Heidelberg, 2008.

[BKA+07a] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE Trans. Softw. Eng.*, 33(9):577–591, 2007.

[BKA+07b] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering*, 33:577–591, 2007.

[BM03] Bart Du Bois and Tom Mens. Describing the Impact of Refactoring on Internal Program Quality. In *International Workshop on Evolution of Large-scale Industrial Software Applications (ELISA)*, September 2003.

[BM06] Fabian Bannwart and Peter Müller. Changing Programs Correctly : Refactoring with Specifications. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 492–507. Springer Berlin / Heidelberg, 2006.

[BM07] Thomas Baar and Slaviša Marković. A Graphical Approach to Prove the Semantic Preservation of UML/OCL Refactoring Rules. In I. Virbitskaite and A. Voronkov, editors, *PSI 2006*, volume 4378 of *LNCS*, pages 70–83. Springer-Verlag Berlin Heidelberg, 2007.

[BMR+98] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-orientierte Softwarearchitektur. Ein Pattern-System.* Addison-Wesley-Longman, 1998.

[BN99] Franz Baader and Tobias Nipkow. *Term rewriting and all that.* Cambridge University Press, 1999.

[Boi04] Bart Du Bois. Opportunities and challenges in deriving metric impacts from refactoring. In Stéphane Ducasse Serge Demeyer and Kim Mens, editors, *Proceedings WOOR'04 (ECOOP'04 Workshop on Object-Oriented Re-engineering)*. Universiteit Antwerpen, Juni 2004.

[BPR08] Steffen Becker, Frantisek Plasil, and Ralf Reussner, editors. *Quality of Software Architectures. Models and Architectures, 4th International Conference on the Quality of Software-Architectures, QoSA 2008, Karlsruhe, Germany, October 14-17, 2008. Proceedings*, volume 5281 of *Lecture Notes in Computer Science*. Springer, 2008.

[BR98] John Brant and Don Roberts. The Refactoring Browser. In Serge Demeyer and Jan Bosch, editors, *ECOOP Workshops*, volume 1543 of *Lecture Notes in Computer Science*, page 549. Springer, 1998.

[BS03] E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis.* Springer-Verlag, 2003.

[CC90] E.J. Chikofsky and II Cross, J.H. Reverse engineering and design recovery: a taxonomy. *Software, IEEE*, 7(1):13 –17, January 1990.

[CCGdL10] Jordi Cabot, Robert Clarisó, Esther Guerra, and Juan de Lara. A UML / OCL Framework for the Analysis of Graph Transformation Rules. *Software and Systems Modeling*, 9:335–357, 2010.

[CCS02] M. L. Cornélio, A. L. C. Cavalcanti, and A. C. A. Sampaio. Refactoring by Transformation. In *REFINE'2002*, volume 70 of *Eletronic Notes in Theoretical Computer Science*. Elsevier, 2002.

[CG90] D. N. Card and R. L. Glass. *Measuring Software Design Quality.* Prentice-Hall, Upper Saddle River, NJ, USA, 1990.

[Che76] Peter P. Chen. The Entity-Relationship Model - Toward a Unified View of Data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976.

[Cop07] Tom Copeland. *Generating Parsers With JavaCC.* Centennial Books, 2007. ISBN: 0-9762214-3-8.

[Cor] Microsoft Corp. Microsoft VisualStudio. https://www.microsoft.com/germany/visualstudio/. Stand: 23.03.2011.

[Cor04] Márcio L. Cornèlio. *Refactorings as Formal Refinement.* PhD thesis, Universidade Federal de Pernambuco, 2004.

[Cun] Cunningham and Cunningham. Portland Pattern Repository. http://c2.com/.

[CW04] Alexandre Correa and Cláudia Werner. Applying Refactoring Techniques to UML / OCL Models Common Problems in UML / OCL Models. In T. Baar Et Al, editor, *UML 2004, LNCS 3273*, pages 173–187. Springer-Verlag Berlin Heidelberg, 2004.

[DB99] J. Derrick and E. A. Boiten. Calculating upward and downward simulations of state-based specifications. *Information & Software Technology*, 41(13):917–923, 1999.

[DB01] J. Derrick and E. A. Boiten. *Refinement in Z and Object-Z.* Springer, 2001.

[DDN00] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Finding Refactorings via Change Metrics. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '00, pages 166–177, Minneapolis, Minnesota, United States, 2000. ACM.

[dE98] W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison.* CUP, 1998.

[DW06] J. Derrick and H. Wehrheim. Model Transformations Incorporating Multiple Views. In *AMAST*, pages 111–126, 2006.

[DW10] J. Derrick and H. Wehrheim. Model transformations across views. *Science of Computer Programming*, 75(3):192–210, March 2010.

[EESV08] Torbjörn Ekman, Ran Ettinger, Max Schäfer, and Mathieu Verbraere. Refactoring bugs in Eclipse, IntelliJ IDEA and Visual Studio, 2008.

[EF02] Rik Eshuis and Maarten M. Fokkinga. Comparing refinements for failure and bisimulation semantics. *Fundam. Inf.*, 52:297–321, April 2002.

[EJ04] Niels Van Eetvelde and Dirk Janssens. Extending Graph Rewriting for Refactoring. In *Graph Transformations*, pages 399–415, 2004.

[EPTJ01] Gerson Suny E, Damien Pollet, Yves Le Traon, and Jean-marc J. equel. Refactoring UML models. *In UML*, pages:134–148, 2001.

[Erb10] Stephan Erb. A Survey of Software Refactoring Tools. 2010.

[ERW07] H.-C. Estler, T. Ruhroth, and H. Wehrheim. Modelchecking Correctness of Refactorings - Some Experiments. *Electron. Notes Theor. Comput. Sci.*, 187:3–17, 2007.

[EW01] Rik Eshuis and Roel Wieringa. A Formal Semantics for UML Activity Diagrams - Formalising Workflow Models, 2001.

[EW08] H. Estler and H. Wehrheim. Alloy as a Refactoring Checker? In J. Derrick E. Boiten and G. Schellhorn, editors, *Proceedings of the 13th BAC-FACS Refinement Workshop (REFINE 2008)*, volume 214, pages 331–357. Elsevier, 2008.

[FDR97] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement: FDR2 User Manual*, Oct 1997.

[Fis97] C. Fischer. CSP-OZ: A combination of Object-Z and CSP. In *FMOODS '97*, volume 2, pages 423–438. Chapman & Hall, 1997.

[Fis00] C. Fischer. *Combination and Implementation of Processes and Data: from CSP-OZ to Java.* PhD thesis, University of Oldenburg, 2000.

[Fou] Eclipse Foundation. Eclipse. http://www.eclipse.org/. Stand: 23.03.2011.

[Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture.* Addison-Wesley-Longman, 2002.

[Fow04] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison Wesley, 2004.

[FS07] Harald Fecher and Jens Schönborn. UML 2.0 State Machines: Complete Formal Semantics Via core state machine. In Luboš Brim, Boudewijn Haverkort, Martin Leucker, and Jaco van de Pol, editors, *Formal Methods: Applications and Technology*, volume 4346 of *Lecture Notes in Computer Science*, pages 244–260. Springer Berlin / Heidelberg, 2007.

[GHJV85] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of reusable object-oriented software.* Addison Wesley, 1985.

[GMB05] Rohit Gheyi, Tiago Massoni, and Paulo Borba. A rigorous approach for proving model refactorings. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 372–375, New York, NY, USA, 2005. ACM.

[GMP03] M. Genero, D. Miranda, and M. Piattini. Defining Metrics for UML Statechart Diagrams in a Methodological Way. In M. A. Jeusfeld and O. Pastor, editors, *ER (Workshops)*, volume 2814 of *LNCS*, pages 118–128. Springer, 2003.

[Gra90] T. Grams. *Denkfallen und Programmierfehler*. Springer-Verlag, 1990.

[HM95a] M. Hitz and B. Montazeri. Measuring Coupling and Cohesion in Object-Oriented Systems. In *Proc. Intl. Sym. on Applied Corporate Computing*, 1995.

[HM95b] M. Hitz and B. Montazeri. Measuring Product Attributes of Object-Oriented Systems. In *Proc. ESEC '95*. Springer, 1995.

[HMU02] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie* . Pearson, 2002.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12:576–580, October 1969.

[Hoa85] C.A.R. Hoare. *Communicating sequential processes*. Prentice Hall, 1985.

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of Semantics? *Computer*, 37(10):64–72, 2004.

[Int96] International Standard Organisation. Information technology - Syntactic metalanguage - Extended BNF, 1996. ISO/IEC 14977:1996(E).

[Int02] International Standard Organisation. Information technology - Z formal secification notation - Syntax, type system and semantics, 2002. ISO/IEC 13568:2002(E).

[ISO02] ISO/IEC. International Organization for Standardization (ISO) and International Electrotechnical Commision (IEC) 15939:2002: Software engineering - Software measurement process, 2002.

[Java] Download Java Referenz Implementation. http://jcp.org/aboutJava/communityprocess/final/jsr316/index.html. Stand: 23.03.2011.

[javb] Java. http://java.sun.com/. Stand: 23.03.2011.

[Javc] Java Language Specification. http://java.sun.com/docs/books/jls/. Stand: 23.03.2011.

[JRL92] Doug Brown John R. Levine, Tony Mason. *Lex and yacc*. A Nutshell handbook Computer Science Series. O'Reilly Media, Inc., 1992.

[jun] JUnit Homepage: Resources for Test Driven Development. http://www.junit.org/. Stand: 23.03.2011.

[Kar10] Nigel Karsidi. Literature Survey: Refactoring. In Eelco Visser, editor, *Proceedings of the Seminar on Meta-Programming (SMP 2010)*, 2010.

[KBS08] Nicholas A. Kraft, Brandon W. Bonds, and Randy K. Smith. Cross-language Clone Detection. In *SEKE*, pages 54–59. Knowledge Systems Institute Graduate School, 2008.

[KDO+08] Andrè Kemena, Bernhard Dietrich, Christoph Oberhokamp, Martin Kleine, Mathias Raacke, Meik Piepmeyer, Peter Winkelhane, Rudolf Braun, Steffen Dohle, Tobias Friedrich, and Yi Tan. Abschlussdokumentation der Projektgruppe RefaSo & ModelCockpit, April 2008.

[KGBH10] Lucia Kapová, Thomas Goldschmidt, Steffen Becker, and Jörg Henss. Evaluating Maintainability with Code Metrics for Model-to-Model Transformations. In George Heineman, Jan Kofron, and Frantisek Plasil, editors, *Research into Practice – Reality and Gaps*, volume 6093 of *Lecture Notes in Computer Science*, pages 151–166. Springer Berlin / Heidelberg, 2010.

[KGKK02] Sabine Kuske, Martin Gogolla, Ralf Kollmann, and Hans-Jörg Kreowski. An Integrated Semantics for UML Class, Object and State Diagrams Based on Graph Transformation. In *Proceedings of the Third International Conference on Integrated Formal Methods*, IFM '02, pages 11–28, London, UK, 2002. Springer-Verlag.

[KHE03] J. Küster, R. Heckel, and G. Engels. Defining and validating transformations of UML models. In *HCC*, pages 145–152. IEEE Computer Society, 2003.

[KR78] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language.* Prentice-Hall, 1978.

[Kur08] Ivan Kurtev. State of the Art of QVT : A Model Transformation Language Standard. In A. Schürr, M. Nagl and A. Zündorf, editors, *AGTIVE 2007,LNCS 5088*, number ii, pages 377–393. Springer-Verlag Berlin Heidelberg, 2008.

[KW00] A. Kleppe and J. Warmer. An Introduction to the Object Constraint Language (OCL). In *TOOLS (33)*, page 456. IEEE Computer Society, 2000.

[Läm02] Ralf Lämmel. Towards generic refactoring. In Bernd Fischer and Eelco Visser, editors, *ACM SIGPLAN Workshop on Rule-Based Programming*, pages 15–28. ACM, 2002.

[Li06] Huiqing Li. *Refactoring Haskell Programs.* PhD thesis, University of Kent, Computing Laboratory, 2006.

[LT06] Huiqing Li and Simon Thompson. Comparative Study of Refactoring Haskell and Erlang Programs. In *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 197–206, Washington, DC, USA, 2006. IEEE Computer Society.

[LT08] Huiqing Li and Simon Thompson. Tool support for refactoring functional programs. In *Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '08, pages 199–203, New York, NY, USA, 2008. ACM.

[LZ08] Hui Liu and Bin Zhu. Refactoring Formal Specifications in Object-Z. In *Computer Science and Software Engineering*, pages 342–345, Wuhan, Hubei, Dezember 2008.

[Mar09] Robert C. Martin. *Clean Code: A handbook of agile software craftsmanship.* Prentice Hall, 2009.

[MB05] Tiago Massoni and Paulo Borba. A Model-driven Approach to Formal Refactoring. *October*, 1:124–125, 2005.

[MB07] Slaviša Marković and Thomas Baar. Refactoring OCL annotated UML class diagrams. *Software & Systems Modeling*, 7(1):25–47, 2007.

[McC04] Tim McComb. Refactoring Object-Z Specifications. In Michel Wermelinger and Tiziana Margaria-Steffen, editors, *Fundamental Approaches to Software Engineering*, volume 2984 of *Lecture Notes in Computer Science*, pages 69–83. Springer Berlin / Heidelberg, 2004.

[MDJ02] Tom Mens, Serge Demeyer, and Dirk Janssens. Formalising Behaviour Preserving Program Transformations. In *Proceedings of the First International Conference on Graph Transformation*, ICGT '02, pages 286–301, London, UK, 2002. Springer-Verlag.

[MEDJ05] T. Mens, N. Van Eetvelde, S. Demeyer, and D. Janssens. Formalizing refactorings with graph transformations. *Journal of Software Maintenance*, 17(4):247–276, 2005.

[MGB05] Tiago Massoni, Rohit Gheyi, and Paulo Borba. Formal Refactoring for UML Class Diagrams. In *19TH BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING (SBES)*, pages 152–167, 2005.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37:316–344, December 2005.

[Mic97] Sun Microsystems. Java Code Conventions. http://www.oracle.com/technetwork/java/codeconventions-150003.pdf, 1997. Stand: 23.03.2011.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[MMBJ09] Naouel Moha, Vincent Mahé, Olivier Barais, and Jean-Marc Jézéquel. Generic Model Refactorings. In Andy Schürr and Bran Selic, editors, *Model Driven Engineering Languages and Systems*, volume 5795 of *Lecture Notes in Computer Science*, pages 628–643. Springer Berlin / Heidelberg, 2009.

[MOF02] Meta-Object Facility (MOF) 1.4 Specification. http://www.omg.org/technology/documents/formal/mof.htm, April 2002.

[MRG+06] Naouel Moha, Jihene Rezgui, Yann-Gael Gueheneuc, Petko Valtchev, and Ghizlane El Boussaidi. Using FCA to Suggest Refactorings to CorrectDesign Defects. In *CLA*, 2006.

[MRG09] Maddeh Mohamed, Mohamed Romdhani, and Khaled Ghadira. Classification of Model Refactoring Approaches. *Journal of Object Technology*, 8(6):143–158, 2009.

[MS01] Taichi Muraki and Motoshi Saeki. Metrics for applying GOF design patterns in refactoring processes. In *Proceedings of the 4th International Workshop on Principles of Software Evolution*, IWPSE '01, pages 27–36, New York, NY, USA, 2001. ACM.

[MS04] T. McComb and G. Smith. Architectural Design in Object-Z. In *Australian Software Engineering Conference (ASWEC'04)*, pages 77 – 86. IEEE Computer Society Press, 2004.

[MS06] T. McComb and G. Smith. Refactoring Object-Oriented Specifications: A Process for Deriving Designs. Technical Report SSE-2006-01, University of Queensland, Australia, May 2006.

[msd] Microsoft DSL Tools für VisualStudio. http://msdn.microsoft.com/en-us/library/bb126259.aspx. Stand: 23.03.2011.

[MT04] Tom Mens and Tom Tourwé. A Survey of Software Refactoring. *IEEE Trans. Softw. Eng.*, 30(2):126–139, 2004.

[MVEDJ05] Tom Mens, Niels Van Eetvelde, Serge Demeyer, and Dirk Janssens. Formalising Refactorings with Graph Transformations ⋆. *J. Softw. Maint. Evol.*, 17:247–276, July 2005.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. Isabelle/HOL A Proof Assistant for Higher-Order Logic. In *Lecture Notes in Computer Science*, volume 2283, 2002.

[OMG03] OMG. *MDA Guide Version 1.0.1*, 2003.

[Opd92] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[Pau89] Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363 – 397, September 1989.

[Per04] Roly Perera. Refactoring : To the Rubicon ... and Beyond ! In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '04, pages 2–3, New York, NY, USA, 2004. ACM.

[pic99] *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (FOCS)*, 1977.

[PP91] Maurizio Proietti and Alberto Pettorossi. Semantics preserving transformation rules for Prolog. In *Proceedings of the 1991 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '91, pages 274–284, New York, NY, USA, 1991. ACM.

[RGTL09] Sylvain Robert, Sébastien Gérard, François Terrier, and François Lagarde. A Lightweight Approach for Domain-Specific Modeling Languages Design. In *Proceedings of the 2009 35th Euromicro Conference on Software Engineering and Advanced Applications*, SEAA '09, pages 155–161, Washington, DC, USA, 2009. IEEE Computer Society.

[Rob99] D.B. Roberts. *Practical Analysis For Refactoring.* PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[Ros97] W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall, 1997.

[RSA10] Jan Reimann, Mirko Seifert, and Uwe Aßmann. Role-Based Generic Model Refactoring. In Dorina Petriu, Nicolas Rouquette, and Oystein Haugen, editors, *Model Driven Engineering Languages and Systems*, volume 6395 of *Lecture Notes in Computer Science*, pages 78–92. Springer Berlin / Heidelberg, 2010.

[RSM06] R. Ramos, A. Sampaio, and A. Mota. Transformation Laws for UML-RT. In R. Gorrieri and H. Wehrheim, editors, *FMOODS*, volume 4037 of *LNCS*. Springer, 2006.

[RSVG07] J. Ratzinger, T. Sigmund, P. Vorburger, and H. Gall. Mining Software Evolution to Predict Refactoring. In *ESEM '07*, Washington, DC, USA, 2007. IEEE Computer Society.

[Ruh06] Th. Ruhroth. Refactoring Object-Z Specifications. In *18th Nordic Workshop on Programming Theory*, 2006.

[RW07] T. Ruhroth and H. Wehrheim. Refactoring Object-Oriented Specifications with Data and Processes. In M. M. Bonsangue and E. B. Johnsen, editors, *FMOODS*, volume 4468 of *LNCS*, pages 236–251. Springer, 2007.

[SB88] Thomas Stahl and Jorn Bettin. *Modellgetriebene Softwareentwicklung.* Number 978-3-89864-448-8. dpunkt-Verl., 1988.

[Sch99] Steve Schneider. *Concurrent and Real Time Systems: The CSP Approach.* John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1999.

[SD06] R. Van Der Straeten and M. D'Hondt. Model Refactorings through Rule-Based Inconsistency Resolution. In J. Bézivin, editor, *Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 71210–1217, 2006.

[SDS+10] Max Schäfer, Julian Dolby, Manu Sridharan, Frank Tip, and Emina Torlak. Correct Refactoring of Concurrent Java Code. In Theo D'Hondt, editor, *European Conference on Object-Oriented Programming (ECOOP).* Springer-Verlag, 2010.

[SEdM08a] Max Schäfer, Torbjörn Ekman, and Oege de Moor. Challenge proposal: verification of refactorings. In *Proceedings of the 3rd workshop on Programming languages meets program verification*, PLPV '09, pages 67–72, New York, NY, USA, 2008. ACM.

[SEdM08b] Max Schäfer, Torbjörn Ekman, and Oege de Moor. Sound and Extensible Renaming for Java. In Gregor Kiczales, editor, *OOPSLA*. ACM Press, 2008.

[Smi00] G. Smith. *The Object-Z Specification Language*. KAP, 2000.

[Spi92] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall International Series in Computer Science, 1992.

[SPT03] S. Stepney, F. Polack, and I. Toyn. A Z Patterns Catalogue I: Specification and refactorings, v0.1. Technical Report YCS-2003-349, University of York, 2003.

[SPTJ01] Gerson Sunye, Damien Pollet, Yves Le Traon, and Jean-Marc Jezequel. Refactoring UML Models. In *UML 2001*, volume 2185. Springer Verlag, 2001.

[SSL01] Frank Simon, Frank Steinbrückner, and Claus Lewerentz. Metrics Based Refactoring. pages 30–, 2001.

[ST08] Nik Sultana and Simon Thompson. Mechanical verification of refactorings. In *Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '08, pages 51–60, New York, NY, USA, 2008. ACM.

[Sta73] Herbert Stachowiak. *Allgemeine Modelltheorie*. Number 3-211-81106-0. Springer, 1973.

[Sun] Sun. Netbeans. http://www.netbeans.org/. Stand: 23.03.2011.

[Szl06] M. Szlenk. Formal Semantics and Reasoning about UML Class Diagram. In *Dependability of Computer Systems, 2006. DepCos-RELCOMEX '06. International Conference on*, pages 51 –59, May 2006.

[TM03] T. Tourwé and T. Mens. Identifying Refactoring Opportunities Using Logic Meta Programming. In *CSMR '03*, page 91, Washington, DC, USA, 2003. IEEE Computer Society.

[TW4] TW4XP. http://www.tw4xp.com/. Stand: 23.03.2011.

[UML] Unified Modeling Language.

[VE08] H. Voigt and G. Engels. Kontextsensitive Qualitätsplanung für Software-Modelle. In Thomas Kühne, Wolfgang Reisig, and Friedrich Steimann, editors, *Modellierung 2008, 12.-14. March 2008, Berlin*, pages 165–180. GI-Edition, Lecture Notes in Informatics, LNI, 2008.

[Ver08] Mathieu Verbaere. A Language to Script Refactoring Transformations. *Environments*, 2008.

[vG90] Rob J. van Glabbeek. The Linear Time-Branching Time Spectrum (Extended Abstract). In Jos C. M. Baeten and Jan Willem Klop, editors, *CONCUR*, volume 458 of *Lecture Notes in Computer Science*, pages 278–297. Springer, 1990.

[vG93] Rob J. van Glabbeek. The Linear Time - Branching Time Spectrum II. In Eike Best, editor, *CONCUR*, volume 715 of *Lecture Notes in Computer Science*, pages 66–81. Springer, 1993.

[Voi09] Hendrik Voigt. *Kontextsensitive Qualitätsplanung von Softwaremodellen.* PhD thesis, Universität Paderborn, 2009.

[VR08] H. Voigt and T. Ruhroth. A Quality Circle Tool for Software Models. In Q. Li et al., editor, *ER*, volume 5231 of *LNCS*, pages 526–527. Springer, 2008.

[WD97] Jim Woodcock and Jim Davies. *Using Z.* Prentice-Hall, 1997.

[xte] Xtext - Language Development Framework. http://www.eclipse.org/Xtext/. Stand: 23.03.2011.

[YLM+98] Yijun Yu, Julio Cesar Leite, John Mylopoulos, Linda Lin Liu, Eric Yu, and Erik D Hollander. Software refactoring guided by multiple soft-goals. In *IN PROC. OF COLING–ACL '98*, pages 876–880, 1998.

[Zam03] A. V. Zamulin. Formal Semantics of Java Expressions and Statements. *Programming and Computer Software*, 29:259–270, 2003.

[Zie10] S. Ziegert. Generierung von Refaktoringwerkzeugen. Master's thesis, Universität Paderborn, Juni 2010.