Eine Algebra für Cache-Modelle zum methodenbasierten Caching im Applikationsserver-Bereich

Daniel Pfeifer
10. Februar 2004

Technischer Bericht

Universität Karlsruhe (TH)
Fakultät für Informatik
Institut für Programmstrukturen und Datenorganisation
Lehrstuhl Professor Lockemann

Zusammenfassung

Der vorliegende Bericht behandelt vornehmlich die formalen Aspekte für ein Verfahren zur Spezifikation von Schreib-Lese-Abhängigkeiten für Methoden aus objektorientierten Dienstschnittstellen. Die Spezifikation der Schreib-Lese-Abhängigkeiten wird mit Hilfe sogenannter Cache-Modelle repräsentiert; sie bestimmen, in welcher Weise ein Methodenaufruf von einem anderen anderen Methodenaufruf bei der Nutzung der Dienstschnittstelle abhängt.

Cache-Modelle spielen eine wichtige Rolle beim konsistenten Cachen von Methodenresultaten auf der Klientenseite eines Applikationsserver-Systems. Ein Applikationsserver ist im Kontext dieses Berichts ein Server oder Dienstgeber, der für entfernte Klienten eine objektorientierte Dienstschnittstelle mit Methoden anbietet. Genauer besteht die Dientschnittstelle dabei aus einer Menge abstrakter Klassen mit stark typisierten Methodensignaturen.

Ein methodenbasierter Cache dient zur Verbesserung vor allem der Antwortzeiten eines Applikationsserversystems und hält bereits einmal berechnete Methodenresultate von lesenden Aufrufen bezüglich der Dienstschnittstellen beim Klienten vor. Bei einem wiederholten Methodenaufruf vom Klienten kann das Resultat des Aufrufs aus dem methodenbasierten Cache geholt werden. Sofern der Aufruf mit den gleichen Argumenten wie zuvor stattfindet und sofern das vorgehaltene Resultat gültig ist, muss der eigentlich entfernte Methodenaufruf dann nicht an den Applikationsserver gesendet werden.

Eines der größten Probleme beim methodenbasierten Caching ist die Gültigkeit der beim Klient vorgehaltenen Methodenresultate zuzusichern: Das Resultat aus dem Cache sollte identisch sein zu dem Resultat, das der entsprechende lesende Methodenaufruf bei Übergabe an den Applikationsserver liefern würde. „Lesend" bedeutet hier hauptsächlich, dass der Aufruf den Zustand des Applikationsserver garantiert *nicht* verändert.

Um lesende Methodenaufrufe bzw. Methoden zu bestimmen und um ungültig gewordene Methodenresultate aus dem Cache entfernen zu können, spezifiziert ein Anwendungsentwickler ein Cache-Modell bezüglich der vorhandenen Dienstschnittstelle. Das Cache-Modell identifiziert solche Methoden, die ausschließlich lesen und gibt vor, wie die Abhängigkeiten zwischen schreibenden und lesenden Methoden im Klienten berechnet bzw. abgeschätzt werden können. Dies geschieht auf der Basis eines Formalismus, der im Rahmen des Berichts diskutiert wird.

Bei der Spezifikation von Cache-Modellen ist es wichtig, dass diese korrekt sind, das heißt, das sie die tatsächlichen Scheib-Lese-Eigenschaften der Implementierungen der Dienstmethoden reflektierten. Ansonsten besteht die Gefahr, dass ein methodenbasierter Cache ungültige Methodenresultate an den Klienten zurückliefert oder nicht alle schreibenden Methodenaufrufe für notwendige Zustandsveränderungen an den Applikationsserver delegiert.

Um die Korrektheit von Cache-Modellen abzusichern, wird hier ein einfaches, wenig aufwändiges Testverfahren vorgeschlagen. Streng genommen, kann damit allerdings nur die Anwesenheit von Fehlern in einem Cache-Modell nachgewiesen werden (also Inkorrektheit) und nicht deren Abwesenheit.

Der Entwickler eines Cache-Modells hat zahlreiche Freiheitsgrade bei der Entwicklung eines Cache-Modells, denn es gibt im Allgemeinen viele korrekte Cache-Modelle zu einer Dienstschnittstelle. Weniger „präzise" Cache-Modelle schätzen die Schreib-Lese-Eigenschaften von Dienstmethoden sehr grob (aber konservativ) ab und sind mit geringem Aufwand zu erstellen. Der resultierende Nachteil sind eventuell häufige und unnötige Invalidierungen im Cache und geringe Cache-Trefferraten. Präzisere Cache-Modelle hingegen führen meist zu besseren Trefferraten, sind aber aufwändiger zu spezifizieren.

Der Bericht formalisiert alle angesprochenen Aspekte von Cache-Modellen also deren Struktur, den Korrektheitsbegriff sowie die Präzision. Letztere wird dabei als eine Ordnungsrelation auf der Menge der korrekten Cache-Modelle zu einer gegebene Dienstschnittstelle eingeführt.

Neben der Wahl der Präzision gibt es auch die Möglichkeit, dass mehrere Entwickler *verschiedene, aber korrekte Cache-Modelle* zur gleichen Dienstschnittstelle definieren. Unter anderem kann damit eine Art Arbeitsteilung für die Entwickler erreicht werden, so dass diese sich relativ unabhängig voneinander auf verschiedene Teile der Dienstschnittstelle konzentrieren. Die korrekten Cache-Modelle können dann durch eine besondere Vereinigungsoperation zu einem gemeinsamen und korrekten Cache-Modell zusammengeführt werden, dass *mindestens so präzise ist wie jedes der Operandenmodelle*. In diesem Zusammenhang werden drei Vereinigungsoperationen unterschiedlicher Qualität und Komplexität vorgestellt und verschiedene, zum Großteil praxisrelevante algebraische Eigenschaften für diese nachgewiesen.

Darüber hinaus wird die Möglichkeit zur Normalisierung von Cache-Modellen untersucht, die allerdings vornehmlich von theoretischem Interesse ist. Abschließend handelt der Bericht einige Detailprobleme ab, die beim praktischen Einsatz eines Methoden-Caches entstehen können.

Inhaltsverzeichnis

1 Einführung **1**

2 Abstrakte Architektur eines Methoden-Caches **2**
2.1 Laufzeitaspekte 2
2.2 Struktur eines Methoden-Caches 4
2.3 Generierungsaspekte 5

3 Formale Methodenabhängigkeiten in Dienstschnittstellen **6**

4 Methodenbasierte Cache-Modelle **7**

5 Die Vereinigung von Cache-Modellen **13**

6 Normalisierung von Cache-Modellen **26**

7 Anwendungsbeispiele **29**

8 Implementierungs- und Anwendungsaspekte **35**
8.1 Korrektheitstests für Cache-Modelle 35
8.2 Cache-Kohärenz bei *n* Klienten 37
8.3 Umgehung des Methoden-Caches 40

1 Einführung

In diesem Bericht werden formale Aspekte für einen neuartigen Ansatz zum transparenten, methodenbasierten Caching im Applikationsserver-Bereich diskutiert. Daneben gibt es einen allgemeinen Einstieg in die Thematik, und ergänzend werden noch einige wichtige praktische Punkte des Verfahrens behandelt.

Einen umfassenden Überblick zum methodenbasierten Caching liefert bereits der technische Bericht [7] bzw. das Papier [8]. Unter anderem werden dort praktische Aspekte und realitätsnahe Leistungsexperimente bezüglich der Anwendung des Verfahrens im Applikationsserver-Bereich diskutiert. Zum besseren Verständnis des vorliegendes Berichts sei [7] als Einstiegslektüre empfohlen.

Im Gegensatz zu den meisten klassischen Cache- und Replikationsverfahren im Klient-Server-Bereich stützt sich der hier beschriebene Ansatz nicht auf das Vorhalten von Objektzuständen (etwa in Form von Attributwerten), sondern auf *das Caching von Resultaten von Methodenaufrufen*, die keine Seiteneffekte verursachen.

Existierende attributbasierte Cache-Verfahren für objektorientierte Klient-Server-Architekturen wurden etwa in den Arbeiten [2, 3, 6, 9, 10] und vielen weiteren Forschungsbeiträgen eingehend untersucht. Ein bedeutender Nachteil dieser Ansätze ist die Tatsache, dass dabei das Kapselungsprinzip zwischen Klient und Server verletzt wird, da so Server-seitige Objektzustände zum Klienten gelangen. Speziell im Applikationsserver-Bereich, wo objektorientierte Dienstschnittstellen meist nur aus Methoden bestehen, müsste man mit attributbasierten Cache-Verfahren auch die Implementierung der Dienst-Methoden zum Klienten verlagern, denn nur die Dienstmethoden dürfen auf die entsprechenden Attribute zugreifen. Offensichtlich untergräbt diese Vorgehensweise komplett die Aufgabentrennung von Klient- und Applikationsserver und ist daher nicht immer empfehlenswert.

Methodenbasiertes Caching kann *transparent, konsistent und effizient* betrieben werden. Hierzu wird in Abschnitt 2 die Architektur eines *Methoden-Caches* eingeführt.

Transparent bedeutet in erster Linie, dass der Klientencode, der die Dienstschnittstelle des Applikationsservers verwendet, bei der (eventuell nachträglichen) Einführung eines Methoden-Caches nicht verändert werden muss. Dies ermöglicht den Einsatz der Technologie auch in späten Entwicklungszyklen ohne großen Anpassungsaufwand.

Konsistent bedeutet, dass das gecachte Ergebnis eines Methodenaufrufs *immer* mit dem Wert übereinstimmt, den der Applikationsserver ohne einen entsprechenden Cache liefern würde. Weiter werden alle Zustandveränderungen beim Applikationsserver in der selben Weise ausgeführt wie bei einem entsprechenden System ohne Cache.

Effzient bedeutet schließlich, dass die zusätzlichen Laufzeitkosten für die Nutzung und Verwaltung eines solchen Caches im Vergleich mit den eingesparten Kosten vernachlässigbar sind. Darüber hinaus hält sich der Entwicklungsaufwand für den Einsatz eine Methoden-Caches in einem „vertretbaren“ Rahmen.

Transparenz wird dadurch erreicht, dass Teile des Methoden-Caches generiert werden. Dabei erzeugt man eine Menge von Klassen, welche die abstrakten Klassen der Dienstschnittstelle des Applikationsservers implementieren.

Der Cache ermöglicht Konsistenz, indem er alle Methodenaufrufe bezüglich der Dienstschnittstelle mit verfolgt. Wenn ein Aufruf den Serverzustand verändern könnte, wird er zum Applikationsserver geschickt, andernfalls kann das Resultat des Aufrufs eventuell aus dem Cache geholt werden. Desweiteren werden durch zustandsverändernde Methodenaufrufe zumindestens die Methodenresultate aus dem Cache gelöscht, die durch die Zustandveränderung

ungültig werden. Der Cache führt dabei die richtigen Lösch- und Delegationsoperationen auf Basis eines so genannten *Cache-Modells* aus, das vom Anwendungsentwickler bezüglich der Dienstschnittstelle erstellt werden muss. Auf Grund der Informationen aus dem Cache-Modell „weiß“ der Methoden-Cache, welche Methoden zum Applikationsserver zu senden sind und wann bestimmte Methodenresultate aus dem Cache entfernt werden müssen.

Darüber hinaus zeigen [8, 7] durch realitätsnahe Experimente, dass mit methodenbasiertem Caching auch sehr gute *Leistungsverbesserungen* für ein Applikationserver-System erzielt werden können.

Der Bericht ist wie folgt aufgebaut: Abschnitt 2 führt die abstrakte Architektur des Caches ein und behandelt dabei den oben angesprochenen Transparenzaspekt. In Abschnitt 3 werden Methodenabhängigkeiten formal eingeführt – sie liegen einer konsistenten Invalidierungsstrategie für das Methoden-Caching zu Grunde. Abschnitt 4 führt daraufhin Cache-Modelle auf einer formalen Basis ein. Wie man sehen wird, abstrahieren Cache-Modelle von den konkreten Methodenabhängigkeiten im Server und schätzen diese nur (konservativ) ab. Daher muss ein Anwendungsentwickler die realen Methodenabhängigkeiten nicht detailgenau modellieren, sondern kann erhebliche Vereinfachungen vornehmen, was den Entwicklungsaufwand für Cache-Modelle klein hält. Cache-Modelle müssen aber trotzdem korrekt sein, das heißt, wenn eine Abhängigkeit zwischen zwei Dienstmethodenaufrufen auftritt, muss diese durch das Cache-Modell „angezeigt“ werden. Abschnitt 4 definiert daher die Korrektheit von Cache-Modellen und weist die Konsistenz des Cache-Ansatzes für korrekte Modelle nach. Durch den Abschätzungscharakter von korrekten Modellen ergeben sich gewisse Freiheitsgrade bei deren Implementierung. Dies führt zum Präzisionsbegriff, der die korrekten Modelle für eine Dienstschnittstelle nach dem Grad ihrer Abschätzung von Methodenabhängigkeiten ordnet. In Abschnitt 5 wird vor allem die formale Vereinigung von korrekten Cache-Modellen (bezüglich einer Dienstschnittstelle) behandelt. Die Vereinigung ist in der Praxis nützlich, wenn etwa mehrere Entwickler am gleichen Cache-Modell arbeiten sollen und sich dabei Modellierungsaufgaben aufteilen. Beide können dann unabhängig voneinander korrekte Teilmodelle entwickeln, aus denen man (durch eine entsprechende Vereinigung) *automatisch* ein gemeinsames korrektes Cache-Modell berechnen kann. Für die so genannte Produktvereinigung kann man dabei garantieren, dass das derart erzeugte, gemeinsame Cache-Modell *mindestens so präzise ist wie jedes Teilmodell*. Abschnitt 5 stellt aufbauend auf der „naiven“ Vereinigung eine verfeinerte Vereinigung und schließlich die Produktvereinigung vor. Cache-Modelle können auch auf eine Normalform gebracht werden, die aber vornehmlich von theoretischem Interesse ist und in Abschnitt 6 abgehandelt wird. Abschnitt 7 ergänzt den Formalismus zu Cache-Modellen durch praxisorientierte Beispiele. Zuletzt handelt Abschnitt 8 wichtige Details zur Implementierung eines entsprechenden Cache-Systems ab. Insbesondere werden in Abschnitt 8.2 komplexere Klient-Server-Szenarien und deren Auswirkung auf das Methoden-Caching diskutiert.

2 Abstrakte Architektur eines Methoden-Caches

Dieser Abschnitt stellt die allgemeine Architektur eines Methoden-Caches vor und erläutert, wie der Cache mit einem Applikationsserver integriert werden kann. Dabei wird zwischen den Laufzeitaspekten des Methoden-Caches und dem Generierungszeitaspekten unterschieden. Die letzteren beziehen sich auf die Zeit, zu dem die Klassen des Caches erzeugt werden.

2.1 Laufzeitaspekte

In Abbildung 1 ist der Klient- und Serverteil eines Applikationsserver-Systems illustriert: Der Applikationsserver stellt eine objektorientierte Dienstschnittstelle bestehend aus abstrakten

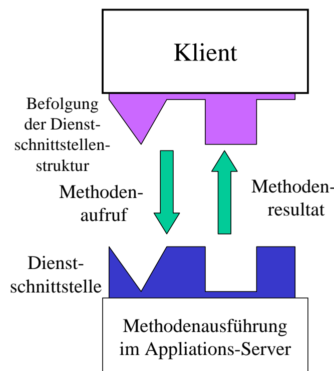


Abbildung 1: Abstraktes Interaktionsschema für einen Applikationsserver und einen Klienten

Klassen bereit, die abstrakte Methoden enthalten.[1] Der Klient kennt die abstrakten Klassen, deren Methoden und die damit zusammenhängenden Aufrufprotokolle. Er ruft eine entsprechende Methode auf und erhält vom Server ein Methodenresultat zurück (das auch leer sein kann). Der Applikationsserver hält intern den Implementierungscode zur Ausführung der Methoden der Dienstschnittstelle bereit.

Üblicherweise sind Methodenaufrufe vom Klienten zum Server entfernt, in manchen Systemkonfigurationen können jedoch Klient und Server im gleichen Anwendungsprozess vorkommen; die entsprechenden Aufrufe sind dann lokal. Desweiteren kann man zwischen Dienstmethoden unterscheiden, deren Implementierungen *niemals* den Zustand des Applikationsservers oder dessen Subsysteme verändern und solchen Methoden, die dies potentiell tun. Die erste Art von Dienstmethoden werden im folgenden als *Lesemethoden* bezeichnet und die letzeren als *Schreibmethoden*.

Abbildung 2 zeigt, wie sich die Architektur des Systems verändert, wenn ein *Methoden-Cache* auf der Klientenseite eingeführt wird. Der Methoden-Cache liegt dabei hinter dem Klientencode und bildet die Schnittstelle der Dienstmethoden nach. Aufrufe an Dienstmethoden, die vormals direkt zum Applikationsserver gingen, erreichen nun zuerst den Cache und werden dann möglicherweise zum Applikationsserver weiter delegiert. Der Methoden-Cache führt unterschiedliche Aktionen aus, je nach dem, ob es sich bei dem entsprechenden Aufruf um eine Lese- oder um eine Schreibmethode handelt. Ob eine Dienstmethode liest oder potentiell schreibt, ist dem Cache dabei auf Grund eines Cache-Modells bekannt, das in Abschnitt 4 eingeführt wird.

Falls es sich bezüglich des Aufrufs um eine Lesemethode handelt, versucht der Cache, ein vorgehaltenes Resultat zu dem Aufruf zu finden, wobei das Resultat bei einem früheren Methodenaufruf im Cache abgelegt wurde. Um die Korrektheit des Resultats sicherzustellen, muss der ursprüngliche Methodenaufruf auf einem identischen Objekt und mit identischen Argumentwerten stattgefunden haben.[2] Also ergibt sich ein entsprechender Cache-Schlüssel aus der Signatur der aufgerufenen Dienstmethode, den Argumentwerten und dem Aufrufobjekt (This-Objekt).

Wird der Schlüssel im Cache gefunden, liegt ein Treffer vor, und das gespeicherte Resultat wird direkt vom Cache zum aufrufenden Klientencode zurückgegeben. Ansonsten wird der Aufruf zum Applikationsserver delegiert. Der Cache speichert das von dort kommende Resultat

[1] Im Fall von EJBs handelt es sich hierbei um Java-Interfaces mit entsprechenden Methoden.

[2] Ob die entsprechenden Werte identisch, sind kann in der Praxis durch anpassbare Gleichheitstests bestimmt werden. Für Java verwendet hierzu standardmäßig die Vergleichsmethode `java.lang.Object.equals()`.

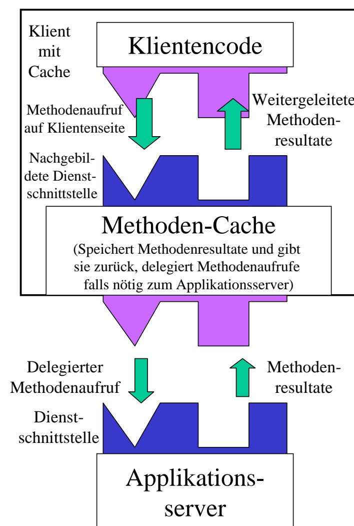


Abbildung 2: Interaktion von Applikationsserver und Klient bei Einführung eines Methoden-Caches auf der Klientenseite

anhand des Cache-Schlüssels und gibt es schließlich ebenfalls an den Klientencode weiter.

Im Falle des Aufrufs einer Schreibmethode leitet der Methoden-Cache den Aufruf zuerst an den Applikationsserver weiter, wo er ausgeführt wird. Danach muss er alle gecachten Resultate von Lesemethoden löschen, die eventuell von den Effekten einer Zustandsveränderung im Applikationsserver (durch den Schreibmethodenaufruf) betroffen sind. Die zu invalidierenden, gecachten Resultate werden dabei anhand der Argumentwerte sowie dem Resultatwert und dem Aufrufobjekt des Schreibmethodenaufrufs bestimmt. Mit Hilfe eines Cache-Modells, das von einem Anwendungsentwickler bezüglich der Dienstschnittstelle spezifiziert wurde, leitet das System aus diesen Werten ab, welche Einträge im Cache tatsächlich zu löschen sind. Abschließend wird das Resultat des Methodenaufrufs an den aufrufenden Klientencode weitergegeben.

Zur Vereinfachung wird im folgenden zunächst der Fall einer Eins-Zu-Eins-Beziehung von Klient und Server unterstellt, bei der der Methoden-Cache, wie in Abbildung 2 illustriert, auf der Klientenseite vorkommt. Alternative bzw. komplexere Klient-Server-Szenarien für das Methoden-Caching werden eingehend in Abschnitt 8.2 diskutiert.

2.2 Struktur eines Methoden-Caches

Abbildung 3 illustriert den Aufbau eines Methoden-Caches in vereinfachter Form als UML-Klassendiagramm: Die generierten Cache-Klassen implementieren die Dienstschnittstellen, so wie sie vom Applikationsserver bereitgestellt werden. Sie enthalten Code, um gecachte Methodenresultate zu finden und Aufrufe an den Applikationsserver weiterzuleiten.

Instanzen generierter Cache-Klassen agieren als Proxy-Objekte für Anwendungsobjekte im Applikationsserver (wie zum Beispiel EJBs). Zur Laufzeit des Methoden-Caches hält der Klientencode Referenzen (oder Handles) zu Proxy-Objekten aus dem Cache anstatt zu entsprechenden Anwendungsobjekten. Es gibt jedoch eine (eindeutige) Eins-Zu-Eins-Zuordnung von Proxy-Objekten und Applikationsserver-Objekten. Ein Proxy-Objekt besitzt hierzu eine privates Attribut, das eine Referenz zum entsprechende Applikationsserver-Objekt enthält. Die Referenz wird immer dann benötigt, wenn ein Methodenaufruf zum Server delegiert werden soll (zum

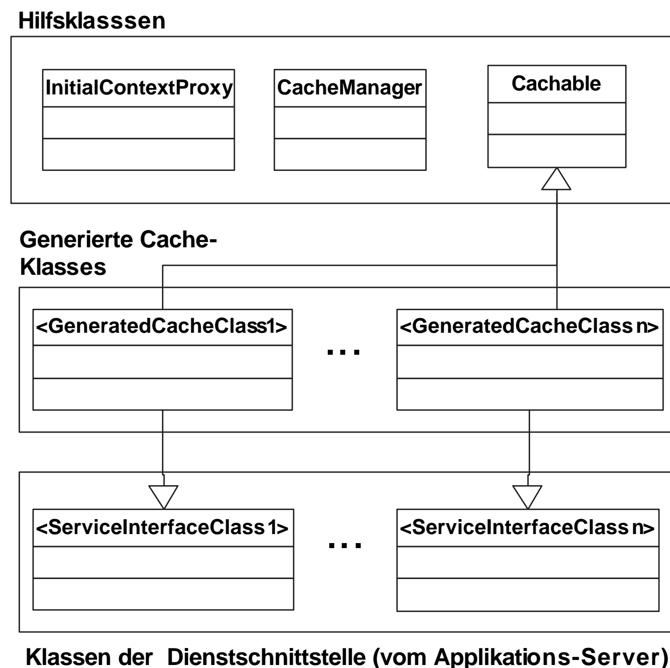


Abbildung 3: Vereinfachte Struktur des Methoden-Caches als UML-Klassendiagramm

Beispiel weil es keinen Treffer im Cache gab). Die Hilfsklassen des Caches stellen die Funktionalität bereit, die von allen generierten Cache-Klassen benötigt wird. Dies umfasst die Abbildung von Anwendungsobjekten auf Proxy-Objekte, das Speichern von Methodenresultaten und wichtige Teile der Konsistenzverwaltung von gecachten Resultaten, also insbesondere deren Invalidierung.

Es stellt sich die Frage, wie es möglich ist, dass der Klientencode ohne Änderungen initial auf die Proxy-Objekte des Methoden-Caches zugreift, anstatt sich direkt an den Applikationsserver zu wenden. In dem meisten Fällen ist das sehr einfach möglich durch das Überschreiben von Fabrik-Methoden (entsprechend dem Entwurfsmuster „Fabrik-Methode“ bzw. „abstrakte Fabrik“, [5]), die die initialien Referenzen auf entfernte Anwendungsobjekte im Klientencode bereitstellen bzw. entfernte Anwendungsobjekte erzeugen.

2.3 Generierungsaspekte

Für den Prototyp, der im Kontext dieses Berichts entwickelt wurde, werden die Cache-Klassen, die die Dienstschnittstelle nachbilden, generiert. Ein entsprechendes Werkzeug verwendet hierzu die Dienstschnittstelle selbst und das Cache-Modell als Eingabedaten und erzeugt die Quelldateien der generierten Klassen. Nachdem die letzteren übersetzt wurden, können sie zusammen mit dem Binärcode der Hilfsklassen im Laufzeitsystem verwendet werden.

Den Datenfluss für das Generatorwerkzeug zeigt Abbildung 4: Die Dienstschnittstelle steht als eine Menge übersetzter Java-Klassen zur Verfügung, deren Methoden-Struktur durch das Java Reflection API (`java.lang.reflect.*`) gelesen wird. Das Cache-Modell ist als XML-Datei repräsentiert, deren formale Semantik in Abschnitt 4 eingehend behandelt wird.

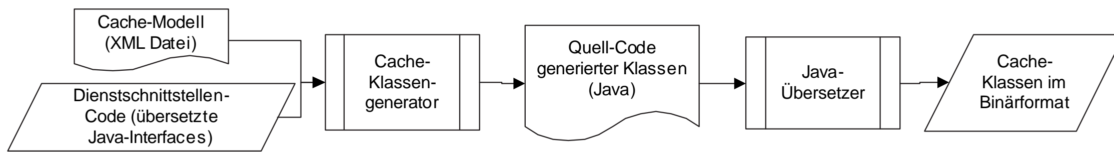


Abbildung 4: Datenfluss bei der Generierung von Cache-Klassen

Alternativ zur statischen Generierung der Cache-Klassen könnte man mit Hilfe des Entwurfmusters „Dynamischer Proxy" (Dynamic Proxy, [4]) die Cache-Klassen auch komplett dynamisch, also zur Laufzeit des Systems erzeugen. Der Hauptnachteil dabei ist, dass Programmcode-Fragmente, die in den XML-basierten Cache-Modellen vorkommen (siehe Abschnitt 7), dann weniger einfach in die Cache-Klassen eingebettet werden können.

3 Formale Methodenabhängigkeiten in Dienstschnittstellen

Dieser Abschnitt behandelt einen einfachen und abstrakten Formalismus zur Beschreibung von Schreib-Lese-Abhängigkeiten bezüglich Methoden in Dienstschnittstellen. Die Definitionen konzentrieren sich hierbei auf die Aspekte, die für die konsistente Invalidierung beim methodenbasierten Caching wesentlich sind; insbesondere werden die Typsicherheit und Signaturen von Methoden hier nicht berücksichtigt.

Definition 1. *Sei Serv ein Server mit einer abzählbaren Menge von möglichen Serverzuständen S. Weiter sei P eine abzählbare Menge von potentiellen Methodenargumenten oder Resultatwerten. Eine* Dienstschnittstelle *ist dann eine endliche Menge von Methoden M, wobei ein $m \in M$ eine berechenbare Funktion repräsentiert, für die gilt:*

$$m : S \times P \rightarrow S \times P, (s, p) \mapsto (s', r).$$

Es wird unterstellt, dass der Server Serv deterministisch arbeitet und sein Zustand nur durch die Methoden aus M beeinflusst wird.

Eine Methode bildet also unter der Aufnahme von Argumenten p einen Serverzustand s auf einen resultierenden Serverzustand s' ab und liefert dabei ein Resultat r.

Das nächste, einfache Beispiel wird zur Illustration der nachfolgenden Sätze und Definitionen benützt. Es behandelt einen Server mit zwei (positiven) ganzzahligen Zählern als Zustand und zwei Methoden *inc* und *get* zum Erhöhen bzw. Lesen der Zähler.

Beispiel 1. *Die möglichen Zustände S des Servers ServZ sind durch $S = \mathbb{N}^2$ gegeben und die Menge von Methodenargumenten bzw. Resultaten ist $P = \mathbb{N}$. Die Methoden $M = \{inc, get\}$ sind dann definiert als*

$$inc : \mathbb{N}^2 \times \mathbb{N} \rightarrow \mathbb{N}^2 \times \mathbb{N}, (s, p) \mapsto ((s.1 + (p+1) \; mod \; 2, s.2 + p \; mod \; 2), s.((p \; mod \; 2) + 1) + 1) \; und$$

$$get : \mathbb{N}^2 \times \mathbb{N} \rightarrow \mathbb{N}^2 \times \mathbb{N}, (s, p) \mapsto (s, s.((p \; mod \; 2) + 1)).$$

Der Anfangszustand von ServZ ist $(0, 0)$.

Mit dem Argument p von *inc* kann man jeweils den ersten oder den zweiten Zähler von *ServZ* addressieren (etwa durch $p = 0$ bzw. $p = 1$). Der Punktoperator $.i$ dient dabei zum Zugriff auf das i-te Element eines n-Tupels – im Beispiel ist i 1 oder 2. *inc* erhöht den addressierten Zähler um 1 und liefert den aktuellen Zählerwert. *get* liefert über seinen Parameter p den aktuellen Zustand des ersten bzw. zweiten Zählers im Server, verändert aber den Serverzustand nicht.

Definition 2. *Ein Tupel* $(m, p, r) \in M \times P^2$ *wird genau dann als* Methodenaufruf *bezeichnet, wenn gilt:*

$$\exists s, s' \in S : m(s, p) \mapsto (s', r).$$

Bezüglich Beispiel 1 ist $(inc, 1, 1)$ ein Methodenaufruf, denn für $s = (0, 0)$ gilt $inc((0, 0), 1) = ((0, 1), 1)$. $(inc, 1, 0)$ ist hingegen kein Methodenaufruf, da man kein geeignetes s und s' zu dem Tripel findet.

Definition 3. *Eine* Lesemethode *ist eine Methode* $m \in M$*, für die gilt:*

$$\forall s \in S : \forall p \in P : \exists r \in P : m(s, p) = (s, r).$$

Eine Schreibmethode $m' \in M$ *ist eine Methode, die keine Lesemethode ist.*

Offenbar ist get in Beispiel 1 eine Lesemethode und inc eine Schreibmethode.

Definition 4. *Eine* Abhängigkeit $dep(m, p, r, m', p', r')$ *existiert genau dann zwischen zwei Methodenaufrufen* (m, p, r) *und* (m', p', r')*, wenn* m *eine Lesemethode ist und wenn gilt:*

$$\exists s, s' \in S : \exists r'' \in P : m(s, p) = (s, r) \wedge m'(s, p') = (s', r') \wedge m(s', p) = (s', r'') \wedge r \neq r''.$$

Die Abhängigkeiten einer Dienstschnittstelle M *bilden die Relation* $dep \subseteq M \times P^2 \times M \times P^2$.

Gemäß der Definition hängt ein Lesemethodenaufruf (m, p, r) genau dann von einem Schreibmethodenaufruf (m', p', r') ab, wenn ein Serverzustand existiert, für den der Schreibmethodenaufruf das Resultat r des Methodenaufrufs (m, p, r) verändert.

Beispiel 2. *Bezüglich Beispiel 1 hat man unter anderem die Abhängigkeit* $dep(get, 1, 0, inc, 1, 1)$*, denn für* $s = (0, 0)$ *ist* $inc((0, 0), 1) = ((0, 1), 1)$ *und* $get((0, 0), 1) = ((0, 0), 0)$ *aber* $get((0, 1), 1) = ((0, 1), 1)$. *Andererseits ist* $(get, 0, 0, inc, 1, 1)$ *nicht in* dep. *Es existiert nämlich kein kein Serverzustand, in dem der Methodenaufruf* $(inc, 1, 1)$ *den Methodenaufruf* $(get, 0, 0)$ *beeinflusst. (Der Grund ist, dass* $(inc, 1, 1)$ *ausschließlich den zweiten Zähler verändert, aber* $(get, 0, 0)$ *liest den ersten Zähler im Server.)*

Auf Grund der Einfachheit von Beispiel 1 lässt sich dep *für diesen Fall sogar ohne weiteres allgemein charakterisieren:* $dep = \{(get, p, r, inc, p', r') \mid p, r, p', r' \in \mathbb{N} \wedge p \bmod 2 = p' \bmod 2 \wedge r + 1 = r'\}$. *Ein get-Aufruf hängt also hier genau dann von einem inc Aufruf ab, wenn inc auf den gleichen Zähler zugreift und wenn inc gerade den Zählerwert erhöht, den get (zuvor) gelesen hat.*

Lemma 1. *Es gibt keine Abhängigkeiten zwischen Aufrufen von Lesemethoden.*

Beweis. Lesemethoden verändern nach Definition keine Serverzustände. □

Man beachte, dass dep bereits von spezifischen Serverzuständen abstrahiert, da diese nicht in die Relation mit eingehen. Die Invalidierung von gecachten Methodenresultaten auf der Klientenseite soll ohne Kenntnis des Serverzustands erfolgen, um nicht gegen das Geheimhaltungsprinzip bezüglich der Dienstschnittstelle zu verstoßen. Die Entscheidung, ob ein Methodenresultat zu invalidieren ist, leitet sich dann nur aus einer Folge von Methodenaufrufen bezüglich der Dienstschnittstelle ab, also völlig ohne Kenntnis des Serverzustands. Deshalb werden Abhängigkeiten gemäß der obigen Definition auch nur aus diesem Blickwinkel betrachtet.

4 Methodenbasierte Cache-Modelle

Cache-Modelle sollen es ermöglichen, Abhängigkeiten zwischen Methodenaufrufen auf der Klientenseite zu bestimmen. Die tatsächlichen Abhängigkeiten zwischen Methodenaufrufen

bezüglich der Dienstschnittstelle können natürlich je nach Dienstimplementierung beliebig komplex sein. Die Bestimmung dieser tatsächlichen Abhängigkeiten würde also eine Simulation oder eine Ausführung entsprechender Methodenaufrufe auf der Klientenseite mit sich bringen. Das würde aber bedeuten, dass der Anwendungscode vom Server zum Klienten verlagert und Serverzustände beim Klienten offen gelegt werden müssten. Da dieser Ansatz gegen das Kapselungsprinzip verstößt und den Klient-Server-Ansatz komplett untergräbt, ist es kein aktzeptabler Weg zur Bestimmung von Abhängigkeiten zwischen Methodenaufrufen.

Stattdessen erlaubt es der im folgenden vorgeschlagene Ansatz, potentielle Abhängigkeiten zwischen Methodenaufrufen (so wie im letzten Abschnitt definiert) *abzuschätzen*. Hierzu wird von den Abhängigkeiten aus Definition 4 abstrahiert und letztere werden durch so genannte *Modellabhängigkeiten* ersetzt. Wenn der Methoden-Cache eine Modellabhängigkeit für das Resultat eines gecachten Lesemethodenaufrufs feststellt, wird der entsprechende Cache-Eintrag invalidiert. Das Vorkommen einer Abhängigkeit wie in Definition 4 impliziert für gecachte Methoden *immer* eine Modellabhängigkeit (sofern das entsprechende Cache-Modell korrekt ist) — die Umkehrung gilt allerdings nicht. Damit ist es möglich, auf Basis eines (korrekten) Cache-Modells den Methoden-Cache konsistent zu halten.

Als zentrale Datenstruktur zur Beschreibung von Modellabhängigkeiten in Cache-Modellen dient eine Kollektion *abstrakter Indizes*. Ein abstrakter Index ist eine Menge, die dabei hilft, Modellabhängigkeiten zu spezifizieren. Wenn ein Schreibmethodenaufruf und ein Lesemethodenaufruf auf das selbe Element eines abstrakten Indexes zugreifen, dann entspricht dies einer Modellabhängigkeit. Zur Laufzeit werden solche Zugriffe im Methoden-Cache für jeden Methodenaufruf bezüglich der Dienstschnittstelle berechnet und überprüft. Die Indizes sind abstrakt, weil sie reale Datenabhängigkeiten, wie sie auf der Serverseite durch die Implementierung der Dienstmethoden entstehen, repräsentieren können, aber nicht müssen. Die Indizes sind also lediglich ein Hilfsmittel, um für den Methoden-Cache eine leichtgewichtige, aber konsistente Invalidierungsstrategie zu entwickeln.

Die Anzahl der Indizes und die Art und Weise, wie der Cache auf deren Elemente zugreift, müssen vom Entwickler des Cache-Modells festgelegt werden: Für jede Dienstmethode muss der Entwickler hierzu angeben, ob es sich um eine Schreib- oder eine Lesemethode handelt. Weiter legt er die abstrakten Indizes für die Methode fest und gibt an, welche Indexelemente in Abhängigkeit eines jeweilige Methodenaufrufs angesprochen werden. Hierzu implementiert der Entwickler so genannte *Indexfunktionen*. Eine Indexfunktion wird zur Laufzeit mit den Argumenten und dem Resultat des Aufrufs einer Dienstmethode parametrisiert. Das Resultat einer Indexfunktion repräsentiert gerade das Indexelement, auf dessen Basis Modellabhängigkeiten für den Dienstmethodenaufruf hergestellt werden. Die folgenden Definitionen formalisieren die entsprechenden Konzepte.

Definition 5. *Es sei R eine Menge, für die eine berechenbare Relation* „gleich“ $\equiv \subseteq R^2$ *existiere, die reflexiv und symmetrisch sei. R wird als* Resultatmenge *bezeichnet, genau dann wenn gilt:*

$$\exists r \in R : \forall q \in R : r \equiv q.$$

Falls mehrere Elemente in $r \in R$ mit der oben genannten Eigenschaft existieren, sei genau eines davon ausgezeichnet und als Joker *bezeichnet. Der Joker wird durch das Symbol $\diamondsuit$ notiert. Falls eine Menge R mit der reflexiven, symmetrischen Relation $\equiv$ keinen Joker enthält, lässt sich R durch Hinzunahme eines entsprechenden Elements zu einer Resultatmenge vervollständigen. Eine derartige Vervollständigung wird mit $R^\diamond$ bezeichnet; falls R bereits eine Resultatmenge ist, gilt $R = R^\diamond$.*[3]

[3] $\equiv$ sollte auf $R^\diamond$ sinnvoller Weise nicht transitiv sein, da sonst wegen der Eigenschaft von $\diamondsuit$ alle $r \in R$ zueinander gleich sind.

Beispiel 3. *Im folgenden wird eine Resultatmenge für die natürlichen Zahlen* $\mathbb{N}$ *auf Basis der Gleichheitsrelation* $=$ *(für Zahlen) konstruiert. Dazu erweitert man* $\mathbb{N}$ *zu* $\mathbb{N}^\diamond = \mathbb{N} \cup \{\diamondsuit_\mathbb{N}\}$*, so dass gilt:* $\forall z \in \mathbb{N} : \diamondsuit_\mathbb{N} = z$.

Lemma 2. *Es seien* $R^\diamond$*,* $Q^\diamond$ *zwei Resultatmengen mit entsprechenden Jokern* $\diamondsuit_R$ *bzw.* $\diamondsuit_Q$*. Dann ist* $R^\diamond \times S^\diamond$ *eine Resultatmenge mit dem Joker* $(\diamondsuit_R, \diamondsuit_S)$ *und einer Gleichheitsrelation, die über den folgenden Ausdruck definiert ist:*

$$\forall (r,q),(r',q') \in R^\diamond \times Q^\diamond : (r,q) \equiv (r',q') :\Leftrightarrow r \equiv r' \wedge q \equiv q'.$$

Beweis. Die Aussage ist offensichtlich. □

Als Beispiel denke man etwa an $\mathbb{N}^\diamond \times \mathbb{N}^\diamond$ mit $\mathbb{N}^\diamond$ gemäß Beispiel 3.

Definition 6. *Cache-Modell: Sei* M *eine Dienstschnittstelle wie in Definition 1 und* $IFun$ *eine endliche Menge berechenbarer* Indexfunktionen*. Jedes Element* $f \in IFun$ *besitze die Signatur* $f : P \times P \to R^\diamond_f$*, wobei* $R^\diamond_f$ *eine Resultatmenge bezüglich* f *bezeichnet. Weiter sei* T *eine endliche Menge von Zeichenketten über einem endlichen Alphabet* α*, wobei* T *nicht das leere Wort* (ε) *enthalte. Ein* Cache-Modell *mod ist dann eine Funktion*

$$mod : M \to \{\mathtt{r},\mathtt{w}\} \times \wp(T \times IFun).^4$$

Weiter muss für mod gelten:

$$\forall m,m' \in M : \forall f,f' \in IFun : \forall t \in T :$$

$$(t,f) \in mod(m).2 \wedge (t,f') \in mod(m').2 \Rightarrow R^\diamond_f = R^\diamond_{f'}.^5$$

Die Anzahl und die Namen der abstrakten Indizes ist hier bestimmt durch T. Eine Funktion $f \in IFun$ mit $(t,f) \in mod(m).2$ gibt an, auf welches Indexelement aus $R^\diamond_f$ durch einen Methodenaufruf (m,p,r) zugegriffen wird (mit $m \in M$ und $p,r \in P$). Die Werte $\mathtt{r}$ und $\mathtt{w}$ drücken aus, ob die entsprechenden Methode liest oder schreibt.

Beispiel 4. *Zu Beispiel 1 werden nun zwei Cache-Modelle* mod_1 *und* mod_2 *definiert. Die Cache-Modelle verwenden die beiden Indexfunktionen* f *und* g*, die auf* $P = \mathbb{N}$ *operieren und Indexelemente aus der Resultatmenge* $\mathbb{N}^\diamond$ *liefern (siehe hierzu Beispiel 2):*

$$f,g : \mathbb{N} \times \mathbb{N} \to \mathbb{N}^\diamond, f(p,r) = p \; mod \; 2, g(p,r) = \diamondsuit_\mathbb{N}.$$

Mit $T = \{1\}$ *kann man* mod_1 *und* mod_2 *etwa wie folgt festlegen:*

$$mod_1 = \{inc \mapsto (\mathtt{w},\{(1,f)\}), get \mapsto (\mathtt{r},\{(1,f)\})\},$$

$$mod_2 = \{inc \mapsto (\mathtt{w},\{(1,g)\}), get \mapsto (\mathtt{r},\{(1,g)\})\}.$$

Wie man später sehen wird, sind die beiden Modelle auch (im Sinne eines noch zu näher bestimmenden Kriteriums) korrekt.

Definition 7. *Eine* Modell-Lesemethode *ist eine Methode* $m \in M$*, für die* $mod(m).1 = \mathtt{r}$ *gilt. Ansonsten handelt es sich bei* m *um eine* Modell-Schreibmethode*. Desweiteren repräsentiert* $mrm(mod)$ *die Menge der Modell-Lesemethoden für eine Cache-Modell mod. Formaler gilt also* $mrm(mod) = \{m \in M \mid mod(m).1 = \mathtt{r}\}$.

[4] $\wp(x)$ beschreibt die Potenzmenge von x.
[5] Durch $.i$ ist hier der Zugriff auf der i-te Element eines n-Tuples notiert.

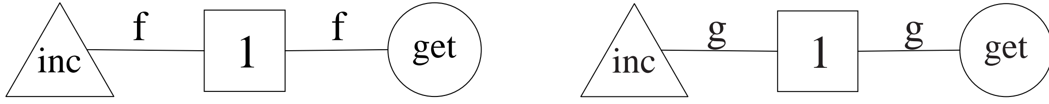


Abbildung 5: Die Cache-Modell-Graphen für mod_1 bzw. mod_2 aus Beispiel 4

Sowohl bei mod_1 als auch bei mod_2 aus Beispiel 1 ist *get* eine Modell-Lesemethode und *inc* eine Modell-Schreibmethode.

Cache-Modelle lassen sich auch durch annotierte, ungerichtete Graphen veranschaulichen – sogenannte *Cache-Modell-Graphen*. Hierzu verwendet man drei verschiedene (annotierte) Knotentypen:

- Kästchen (□) für abstrakte Indizes,
- Kreise (◯) bzw. Ellipsen für Modell-Lesemethoden und
- Dreiecke (△) für Modell-Schreibmethoden.

Die ungerichteten Kanten in Cache-Modell-Graphen verlaufen zwischen Methodenknoten (Kreisen oder Dreiecken) und Knoten für abstrakte Indizes. Ein Kante zeigt an, dass eine Methode im Cache-Modell über eine Indexfunktion auf einen abstrakten Index zugreift. Die Knoten enthalten jeweils den Namen der Methode bzw. den Namen des abstrakten Index. (Eventuell enthalten sie auch Teile der Signatur einer Methode.) Darüber hinaus wird eine Kante mit der Indexfunktion annotiert, über die die entsprechende Methoden auf den jeweiligen Index zugreift.

Abbildung 5 zeigt die Graphen für die Cache-Modelle mod_1 und mod_2 aus Beispiel 4.

Definition 8. *Eine* Modellabhängigkeit $mdep_{mod}(m,p,r,m',p',r')$ *existiert genau dann zwischen zwei Methodenaufrufen* (m,p,r) *und* (m',p',r') *für ein Cache-Modell mod, wenn m eine Modell-Lesemethode und* m' *eine Modell-Schreibmethode ist, so dass gilt:*

$$\exists f,f' \in IFun : \exists t \in T : (t,f) \in mod(m).2 \wedge (t,f') \in mod(m').2 \wedge f(p,r) \equiv f'(p',r').$$

Alle Modellabhängigkeiten für ein Modell mod bilden die Relation $mdep_{mod} \subseteq M \times P^2 \times M \times P^2$.

Die Relation $mdep_{mod}$ ist wohldefiniert, da wegen Definition 6 $R_f^\diamond = R_{f'}^\diamond$ gilt, und somit $f(p,r)$ mit $f'(p',r')$ bezüglich $\equiv$ vergleichbar ist.

Beispiel 5. *Für* mod_2 *aus Beispiel 4 gilt* $mdep_{mod_2}(get,p,r,inc,p',r')$ *für* alle *Methodenaufrufe* (get,p,r) *und* (inc,p',r'). *Dies folgt aus* $g(p,r) = \diamond_{\mathbb{N}} = g(p',r')$ *für alle* $p,r,p',r' \in \mathbb{N}$.

$mdep_{mod_1}$ *ist hingegen eine echte Teilmenge von* $mdep_{mod_2}$. *Man hat etwa* $(get,1,0,inc,1,1) \in mdep_{mod_1}$ *aber* $(get,2,0,inc,1,1) \notin mdep_{mod_2}$. *Dies kann man wie folgt einsehen: Zunächst sind* $(get,1,0)$, $(inc,1,1)$, *und* $(get,2,0)$ *Methodenaufrufe für den Serverzustand* $s=(0,0)$. *Für* $(get,1,0)$ *und* $(inc,1,1)$ *hat man* $f(1,0)=1$ *bzw.* $f(1,1)=1$. *Das Kriterium aus Definition 8 ist also erfüllt für den Index* $1 \in T$. *Für* $(get,2,0)$ *ergibt sich aber* $f(2,0)=0 \neq f(1,1)$. *Neben* $(1,f) \in mod_1(get).2 \cap mod_1(inc).2$ *gibt es keine weiteren Tupel, die eine mögliche Modellabhängigkeit bezüglich* $(get,2,0,inc,1,1)$ *herstellen könnten.*

Allgemein lässt sich $mdep_{mod_2}$ *charakterisieren durch* $mdep_{mod_2} = \{(get,p,r,inc,p',r') \mid p,r,p',r' \in \mathbb{N} \wedge p \bmod 2 = p' \bmod 2 \wedge r+1=r'\}$. *Das bedeutet, es gilt sogar* $mdep_{mod_2} = dep$ *mit* dep *aus Beispiel 2.*

Durch die Verwendung von Resultatmengen mit Jokern bei der Definition von Indexfunktionen kann die besondere Indexfunktion `all` wie nachfolgend definiert werden.

Definition 9. *Sei $IFun$ die Menge von Indexfunktionen für ein Cache-Modell mod. Für die Indexfunktion* $\texttt{all}_{R^\diamond} \in IFun$ *gelte dann:* $\forall p, q \in P : \texttt{all}(p,q) = \diamondsuit_R$. *Da der Joker* $\diamondsuit_R$ *für jede Resultatmenge* $R^\diamond$ *definiert ist, kann man sich die Angabe der Resultatmenge bezüglich* $\texttt{all}_{R^\diamond}$ *auch sparen. Im folgenden wird daher nur noch von der (generischen) Indexfunktion* `all` *gesprochen.*

Da `all` für jedes Argument den Joker liefert, entspricht die Anwendung von `all` gemäß Definition 8 dem Zugriff auf *alle* Elemente eines Index, denn die Beziehung $\diamondsuit \equiv r$ ist für jedes Indexelement $r \in R$ erfüllt. Die Möglichkeit `all` so definieren zu können, ist der Hauptgrund für die Verwendung von Resultatmengen (mit Jokern) bei der Definition von Indexfunktionen. Man hätte Indexfunktionen auch auf gewöhnlichen Mengen mit Gleichheitsrelationen definieren können, um dann `all` als Funktion mit besonderen Eigenschaften in den Definitionen 6 und 8 einzuführen. Dieser Weg hätte allerdings viele der nachfolgenden Beweise unnötig verkompliziert.

Bezüglich mod_2 aus Beispiel 4 kann man etwa die Funktion g genauso gut durch `all` ersetzen, denn es gilt $g = \texttt{all}_{\mathbb{N}^\diamond}$.

Definition 10. *Das folgende Modell heißt das* triviale Modell $mod_{triv}(M)$ *oder kurz* mod_{triv}*, da es unterstellt, dass jede Methode einer Dienstschnittstelle M eine Modell-Schreibmethode ist:*

$$T = \emptyset, \forall m \in M : mod_{triv}(m) = (w, \emptyset).$$

Für das triviale Modell gilt also $mrm(mod_{triv}) = \emptyset$ *und* $mdep_{mod_{triv}} = \emptyset$.

Für Beispiel 1 hat das triviale Modell konkret die folgende Struktur: $mod_{triv} = \{inc \mapsto (\texttt{w}, \emptyset), get \mapsto (\texttt{w}, \emptyset)\}$. Es unterscheidet sich also insbesondere von mod_1 und mod_2 aus Beispiel 4.

Definition 11. *Korrektheit von Cache-Modellen: Ein Cache-Modell mod ist genau dann* korrekt*, wenn jede Modell-Lesemethode eine Lesemethode ist und wenn gilt:*

$$\forall m, m' \in M, \forall p, p', r, r' \in P : m \in mrm(mod) \Rightarrow$$

$$(dep(m, p, r, m', p', r') \Rightarrow mdep_{mod}(m, p, r, m', p', r')).$$

$\mathcal{C}(M)$ oder kurz $\mathcal{C}$ bezeichnet die Menge aller korrekten Cache-Modelle für eine Dienstschnittstelle M.

Ein korrektes Modell muss also Abhängigkeiten für Modell-Lesemethoden immer durch Modellabhängigkeiten anzeigen. Die Umkehrung gilt aber nicht: Ein Cache-Modell kann einen „falschen Alarm“ auslösen und ein gecachtes Methodenresultat invalidieren, obwohl der entsprechende Methodenaufruf beim Server immer noch den gleichen Wert liefern würde. Zum Beispiel kann man für ein korrektes Cache-Modell eine Lesemethode als Modell-Schreibmethode deklarieren und Modellabhängigkeiten zwischen dieser Methode und anderen Modell-Lesemethoden festlegen.

Lemma 3. *Jedes Modell mod mit* $mrm(mod) = \emptyset$ *ist korrekt, insbesondere ist* mod_{triv} *korrekt.*

Beweis. Die Aussage ist offensichtlich. □

Beispiel 6. *Es wird nun die Korrektheit von* mod_1 *und* mod_2 *aus Beispiel 4 untersucht. In beiden Cache-Modellen ist get die einzige Modell-Lesemethode und sie ist in der Tat auch eine Lesemethode. Wie bereits in Beispiel 5 besprochen, gilt* $dep = mdep_{mod_2}$ *für* $M = \{get, inc\}$*, und somit muss* mod_2 *korrekt sein. Da* $mdep_{mod_2}$ *eine (echte) Teilmenge von* $mdep_{mod_1}$ *bildet (siehe Beispiel 5), ist* mod_1 *ebenfalls korrekt.*

Lemma 4. *Korrektheit der Delegation von Schreibmethodenaufrufen zum Server: Für ein korrektes Cache-Modell ist jede Schreibmethode auch eine Modell-Schreibmethode, das heißt Schreibmethodenaufrufe werden immer zum Server Serv delegiert.*

Beweis. Die Aussage ergibt sich direkt aus Definition 11. □

Satz 1. *Korrektheit von Cache-Resultaten: Für ein korrektes Cache-Modell mod zur einer Dienstschnittstelle M sei $m \in M$ eine Modell-Lesemethode und seq eine Sequenz von aufeinander folgenden Methodenaufrufen bezüglich M mit der folgenden Form:*[6]

$$seq = (m(s,p) \mapsto (s,res) = (s_1,res), m_1(s_1,p_1) \mapsto (s_2,r_1),$$

$$m_2(s_2,p_2) \mapsto (s_3,r_2), \ldots, m_n(s_n,p_n) \mapsto (s',r_n), m(s',p) \mapsto (s',res')).$$

Falls gilt:

$$\forall k \in \{1,\ldots,n\} : \neg mdep_{mod}(m,p,r,m_k,p_k,r_k),$$

dann ist $res = res'$.

Beweis. Induktion über n: Für $n = 0$ hat man $seq = (m(s,p) \mapsto (s,res), m(s,p) \mapsto (s,res) = (s,res'))$.

Die Behauptung gelte nun für eine Sequenz seq der Länge $n > 0$. Sei seq' die Sequenz der Länge $n+1$, die entsteht, wenn man die letzten beiden Elemente von seq durch die Elemente $m_n(s_n,p_n) \mapsto (s_{n+1} = s', r_n)$, $m_{n+1}(s_{n+1} = s', p_{n+1}) \mapsto (s'', r_{n+1})$ und $m(s'',p) \mapsto (s'', res'')$ ersetzt. Aus der Voraussetzung $\neg mdep_{mod}(m,p,r,m_{n+1},p_{n+1},r_{n+1})$ folgt $\neg dep(m,p,r,m_{n+1},p_{n+1},r_{n+1})$ wegen $m \in mrm(mod)$ und wegen der Korrektheit von mod. Nach Definition 4 heißt dies, dass $m_{n+1}(s_{n+1},p_{n+1})$ das Resultat für $m(s'',p)$ in seq' nicht beeinflusst. Damit gilt $res' = res''$ für seq' und mit der Induktionsvoraussetzung auch $res = res' = res''$. □

Definition 12. *Präzision von Cache-Modellen: Seien mod_1 und mod_2 zwei korrekte Cache-Modelle für die Dienstschnittstelle M, also $mod_1, mod_2 \in \mathcal{C}(M)$. mod_2 ist genau dann* präziser *als mod_1, wenn $mrm(mod_1)$ eine echte Teilmenge von $mrm(mod_2)$ ist oder wenn beide Mengen gleich sind und $mdep_{mod_2}$ eine echte Teilmenge von $mdep_{mod_1}$ ist. Formaler ausgedrückt, muss gelten:*

$$mrm(mod_1) \subsetneq mrm(mod_2) \vee (mrm(mod_1) = mrm(mod_2) \wedge mdep_{mod_2} \subsetneq mdep_{mod_1}).$$

Präzision wird notiert mit $\prec \subseteq \mathcal{C}(M)^2$. Weiter ist mod_1 genau dann äquivalent *zu mod_2 (notiert durch $mod_1 \sim mod_2$), wenn $mrm(mod_2) = mrm(mod_1) \wedge mdep_{mod_2} = mdep_{mod_1}$ gilt. Darüber hinaus ist $\preceq$ wie folgt definiert: $mod_1 \preceq mod_2 :\Leftrightarrow mod_1 \prec mod_2 \vee mod_1 \sim mod_2$.*

Falls die beiden Cache-Modelle mod_1 und mod_2 also die gleiche Menge von Modell-Lesemethoden enthalten, dann ist mod_2 gemäß der Definition präziser, da es besser modelliert, wann Methodenaufrufe nicht voneinander abhängig sind. Wenn auf der anderen Seite mod_2 mehr Modell-Lesemethoden enthält als mod_1, ist es präziser als mod_1, weil es eine größere Menge von Methoden zu cachen ermöglicht. Man beachte, dass für korrekte Cache-Modelle die Anzahl der Modell-Lesemethoden zwischen 0 (für das triviale Modell) und der Anzahl von Lesemethoden in M liegen kann.

Für die konkreten Cache-Modelle aus Beispiel 4 gilt $mod_1 \prec mod_2$, denn es ist $mrm(mod_1) = mrm(mod_2) = \{get\}$ und $mdep_{mod_2} \subsetneq mdep_{mod_1}$ (wie in Beispiel 5 behandelt). Ferner ist mod_1 nicht äquivalent zum trivialen Modell, weil $mrm(mod_{triv}) = \emptyset$ gilt.

Lemma 5. *$\preceq$ ist eine Halbordnung und $\sim$ eine Äquivalenzrelation.*

Beweis. Offenbar ist $\sim$ reflexiv, transitiv und symmetrisch. $\prec$ ist transitiv: Seien $mod_1, mod_2, mod_3 \in \mathcal{C}(M)$ mit $mod_1 \prec mod_2$ und $mod_2 \prec mod_3$. Falls $mrm(mod_1) \subsetneq mrm(mod_2)$ und $mrm(mod_2) = mrm(mod_3)$, folgt sofort $mrm(mod_1) \subsetneq mrm(mod_3)$. Die anderen Fälle sind ähnlich einfach. Da $\sim$ und $\prec$ transitiv sind, gilt dies auch für $\preceq$. $\prec$ ist antisymmetrisch: Sei nun $mod_1 \preceq mod_2 \wedge mod_2 \preceq mod_1$. Durch Einsetzen in die Definition von $\preceq$ und Ausmultiplizieren von $\wedge$ erhält man: $mrm(mod_1) = mrm(mod_2) \wedge mdep_{mod_2} \subseteq mdep_{mod_1} \wedge mdep_{mod_1} \subseteq mdep_{mod_2}$, also $mod_1 \sim mod_2$. □

[6] Die Methodenaufrufe sind auf Basis von Definition 2 ausgeschrieben.

Lemma 6. *Ein Element* $mod \in \mathcal{C}$ *ist genau dann ein kleinstes Element bezüglich* $\prec$*, wenn* $mrm(mod) = \emptyset$ *gilt. Ist mod ein kleinstes Element, so gilt:* $\forall mod' \in \mathcal{C} : mod \not\sim mod' \Leftrightarrow mod \prec mod'$*. Weiter gilt für 2 beliebige kleinste Elemente* $mod, mod' \in \mathcal{C}$*:* $mod' \sim mod$*. Das triviale Modell* mod_{triv} *ist ein kleinstes Element.*

Beweis. Zur ersten Aussage: Sei $mod \in \mathcal{C}$ und $mrm(mod) \neq \emptyset$. Dann ist $mod_{triv} \prec mod$, weil $mrm(mod_{triv}) = \emptyset \subsetneq mrm(mod)$. Falls andererseits $mrm(mod) = \emptyset$ und $mod' \in \mathcal{C}$ mit $mrm(mod') \neq \emptyset$ gilt, dann ist $mod \prec mod'$, wegen $mrm(mod) = \emptyset \subsetneq mrm(mod')$. Ein mod' mit $mrm(mod') = \emptyset$ ist mit mod bezüglich $\prec$ nicht vergleichbar. Die übrigen Aussagen des Lemmas sind damit offensichtlich. □

Das triviale Modell ist in der Praxis als Ausgangspunkt für die Entwicklung präziserer Cache-Modelle von erheblichem Nutzen. Ein Entwickler kann etwa die Modellabhängigkeiten für wenige Methoden relativ grob beschreiben und damit möglicherweise schnell zu einem korrekten Cache-Modell gelangen, das bereits gute Cache-Trefferraten aufweist. Hierzu deklariert er (ausgehend vom trivialen Modell) nur solche Lesemethoden als Modell-Lesemethoden, von denen der weiß, dass sie häufig (und eventuell mit wenig variierenden Argumentwerten) aufgerufen werden.

Auch die größten Elemente in $\mathcal{C}$ lassen sich durch die folgenden beiden Lemmata etwas genauer charakterisieren.

Lemma 7. *Für ein größtes Element* $mod \in \mathcal{C}$ *bezüglich* $\prec$ *ist die Menge der Modell-Lesemethoden gleiche der Menge der Lesemethoden.*

Beweis. Angenommen die Behauptung wäre falsch für ein größtes Element mod. Dann gäbe es eine Lesemethode m'' in mod, die eine Modell-Schreibmethode ist. Somit lässt sich mod wie folgt zu mod' abändern: Man setzt $mod'(m'').1 := \mathtt{r}$ und $mod'(m).1 := mod(m).1$ für $m \neq m'', m \in M$. Weiter definiert man $T' := T \cup \{t'\}$ mit einem entsprechend gewählten $t' \notin T$ sowie $mod'(m).2 := mod(m).2 \cup \{(t', \mathtt{all})\}$, falls $m = m'$ bzw. $mod(m).1 = \mathtt{w}$ und $mod'(m).2 := mod(m).2$ sonst.

Dass mod' korrekt ist, kann man, wie folgt, einsehen: Seien m, p, r, m', p', r' entsprechend dem Kriterium aus Definition 11 aber ansonsten beliebig gewählt. Falls $m \in mrm(mod')$, gibt es die Fälle $m = m''$ und $m \neq m''$. Wenn $m \neq m''$ und $dep(m, p, r, m', p', r')$ folgt $mdep_{mod}(m, p, r, m', p', r') = mdep_{mod'}(m, p, r, m', p', r')$ auf Grund der Korrektheit von mod und der Definition von mod'. Ansonsten gilt $mdep_{mod'}(m'', p, r, m', p', r')$ für *jeden* beliebigen Schreib-Methodenaufruf (m', p', r') und für jeden Methodenaufruf (m'', p, r). Also ist in beiden Fällen das Kriterium von Definition 11 erfüllt.

mod' ist präziser als mod, da offenbar $mrm(mod) \subsetneq mrm(mod')$ gilt. Damit hat man einen Widerspruch zu der Annahme, dass mod ein größtes Element ist. □

Lemma 8. *Ist für ein korrektes Modell* $mod \in \mathcal{C}$ *die Menge der Modell-Lesemethoden gleich der Menge der Lesemethoden, so gilt* $dep \subseteq mdep_{mod}$*.*

Beweis. Es gelte $dep(m, p, r, m', p', r')$ für geeignete m, p, r, m', p', r'. Dann ist m nach Definition 4 eine Lesemethode und nach Voraussetzung des Lemmas sogar eine Modell-Lesemethode. Wegen der Korrektheit von mod folgt mit Definition 11 sofort $mdep_{mod}(m, p, r, m', p', r')$. □

5 Die Vereinigung von Cache-Modellen

Wie man sich leicht vorstellen kann (und wie man in Abschnitt 7 noch genauer sehen wird), ist die Spezifikation von Cache-Modellen keine triviale Aufgabe. Ein Entwickler, der ein hinreichend präzises Cache-Modell entwerfen soll, muss die Implementierung der entsprechenden

Dienstschnittstelle verstehen und die „echten Datenabhängigkeiten" von darin enthaltenen Methoden kennen. In diesem Zusammenhang wäre es hilfreich, wenn man die Erstellung eines Cache-Modells in Teilaufgaben zerlegen könnte, die bis zu einem gewissen Grad unabhängig voneinander durchführbar sind.

Dieser Abschnitt stellt hierzu verschiedene Vereinigungsoperatoren für Cache-Modelle vor. Damit können für eine Dienstschnittstelle etwa mehrere Cache-Modelle unabhängig von einander entwickelt werden. Wenn diese korrekt sind, wird durch einen Vereinigungsoperator (automatisch) ein gemeinsames Cache-Modell erzeugt, das, wie man zeigen kann, dann ebenfalls korrekt ist. Dabei ist das Vereinigungsmodell insbesondere bei der am Ende dieses Abschnitts eingeführten *Produktvereinigung* mindestens so präzise wie jedes der Ausgangs-Cache-Modelle.

In einem nachfolgenden Schritt kann das Vereinigungsmodell noch automatisch „bereinigt" werden, das heißt, es werden Berechnungen von Indexfunktionen entfernt, die sich als überflüssig herausstellen. Die Präzision des Vereinigungsmodells ändert sich dabei nicht.

Die eingangs gestellte Forderung bezüglich der Zerlegung der Modellentwicklung in Teilaufgaben ist somit auf einfache Weise möglich. Zum Beispiel kann ein Entwickler A die Abhängigkeiten für eine Teilmenge der Methoden der Dienstschnittstelle modellieren und Entwickler B kann dies für eine andere Teilmenge tun. Beide können (bzw. müssen) dann unabhängig von einander absichern, dass ihre Teilmodelle korrekt sind. Wenn Entwickler A sich beispielsweise überhaupt nicht mit einer bestimmten Methode beschäftigen möchte, kann er diese (ausgehend vom trivialen Modell) von vornherein als Modell-Schreibmethode markieren und ist somit sicher, die Korrektheit seines Teilmodells bezüglich dieser Methode nicht zugefährden.

Wenn die in den Teilaufgaben betrachteten Methoden, so wie im eben geschilderten Beispiel, hinreichend verschieden sind, liefern die so genannte *verfeinerte Vereinigung* und die *Produktvereinigung* praktisch immer ein Gesamtmodell, das sogar *wesentlich* präziser ist als jedes der beiden Teilmodelle.

Nachfolgend wird als erstes eine naive, aber einfache Vereinigung von Cache-Modellen vorgeschlagen, die allerdings nicht die gewünschten Eigenschaften zur Präzision der resultierenden Vereinigungsmodelle besitzt. Darauf aufbauend wird dann die verfeinerte Vereinigung und schließlich die Produktvereinigung entwickelt.

Definition 13. *Vereinigung von korrekten Cache-Modellen: Seien $mod_1, mod_2 \in \mathcal{C}(M)$ zwei korrekte Cache-Modelle für die Dienstschnittstelle M mit T_i als die endliche Menge von Zeichenketten zu mod_i und $IFun_i$ als die Menge Indexfunktionen zu mod_i für $i \in \{1,2\}$. T_1 und T_2 seien über dem gleichen Alphabet α definiert, und ohne Beschränkung der Allgemeinheit gelte $T_1 \cap T_2 = \emptyset$.*[7] *Die* Vereinigung *$mod_1 \cup mod_2$ der beiden Cache-Modelle ist dann wie folgt definiert:*

$$mod_1 \cup mod_2 : M \to \{r, w\} \times \wp((T_1 \cup T_2) \times (IFun_1 \cup IFun_2)),$$

$$(mod_1 \cup mod_2)(m) \mapsto (h(m), \{(t,f) \mid (t,f) \in mod_1(m).2 \cup mod_2(m).2\}), \text{ wobei}$$

$$h(m) = \begin{cases} r & \text{falls } m \in mrm(mod_1) \cup mrm(mod_2) \\ w & \text{sonst} \end{cases}$$

Beispielsweise gilt für ein beliebiges korrektes Cache-Modell mod: $mod \cup mod_{triv} = mod$.

Satz 2. *Die Vereinigung zweier korrekter Cache-Modelle $mod_1, mod_2 \in \mathcal{C}$ ist korrekt, also $mod_1 \cup mod_2 \in \mathcal{C}$. Weiter gilt $mrm(mod_1) \setminus mrm(mod_2) \neq \emptyset \Rightarrow mod_2 \prec mod_1 \cup mod_2$ und entsprechend $mrm(mod_2) \setminus mrm(mod_1) \neq \emptyset \Rightarrow mod_1 \prec mod_1 \cup mod_2$. Die Vereinigung von Cache-Modellen ist symmetrisch und assoziativ.*

[7] Falls diese Bedingung nicht zutrifft, kann sie durch eine Umbenennung der entsprechenden Elemente erreicht werden.

Beweis. Zur Korrektheit von $mod_1 \cup mod_2$: Auf Grund der Definition von h in Definition 13 und der Korrrektheit von mod_1 und mod_2 ist jede Modell-Lesemethode eine Lesemethode.

Sei $dep(m,p,r,m',p',r')$ eine Abhängigkeit entsprechend Definition 11 mit der Modell-Lesemethode $m \in mrm(mod_1 \cup mod_2)$. Nach Definition 13 kommt dann m in mod_1 oder mod_2 als Modell-Lesemethode vor. Sei nun ohne Beschränkung der Allgemeinheit $m \in mrm(mod_1)$. Da mod_1 nach Voraussetzung korrekt ist, gilt also $mdep_{mod_1}(m,p,r,m',p',r')$. Nach Definition 8 existiert damit bezüglich mod_1 ein (t,f) zu m und ein (t,f') zu m' mit $f(p,r) \equiv f'(p',r')$. Da für m und (t,f) in Definition 13 $h(m) = mod_1(m).1 = r$ gilt, ist (t,f) in $(mod_1 \cup mod_2)(m).2$. Nach Lemma 4 ist für ein korrektes Cache-Modell jede Schreibmethode auch eine Modell-Schreibmethode. Wegen $dep(m,p,r,m',p',r')$ ist m' eine Modell-Schreibmethode bezüglich mod_1 und mod_2. Mithin ist nach Definition 13 m' eine Modell-Schreibmethode in $mod_1 \cup mod_2$, da $mod_1(m').1 = w$ und $mod_2(m').1 = w$ gilt. Weiter enthält $(mod_1 \cup mod_2)(m').2$ dann auch (t,f').

Zusammenfassend zeigen die vorausgegangenen Überlegungen, dass, falls $m \in mrm(mod_1)$ gilt, bezüglich $mod_1 \cup mod_2$ ein (t,f) zu m und ein (t,f') zu m' existiert mit $f(p,r) \equiv f'(p',r')$. Damit gilt also $mdep_{mod_1 \cup mod_2}(m,p,r,m',p',r')$.

Zur Präzisionsbeziehung: Ohne Beschränkung der Allgemeinheit wird gezeigt: $mrm(mod_1) \setminus mrm(mod_2) \neq \emptyset \Rightarrow mod_2 \prec mod_1 \cup mod_2$. Auf Grund der Definition von $mod_1 \cup mod_2$ und insbesondere von h gilt offenbar $mrm(mod_1), mrm(mod_2) \subseteq mrm(mod_1 \cup mod_2)$. Mit der Voraussetzung $mrm(mod_1) \setminus mrm(mod_2) \neq \emptyset$ gilt dann sogar $mrm(mod_2) \subsetneq mrm(mod_1 \cup mod_2)$. Dies impliziert $mod_2 \prec mod_1 \cup mod_2$ nach Definition 12.

Zur Symmetrie und Assoziativität: Die Eigenschaften übertragen sich direkt aus der Symmetrie und Assoziativität der Vereinigung von Mengen aus Definition 13. □

Lemma 9. *Monotonie der Vereinigung: Seien $mod_1, mod_2, mod_3 \in \mathcal{C}(M)$ drei korrekte Cache-Modelle für die gleiche Dienstschnittstelle M. Dann gilt:*

$$mod_1 \preceq mod_2 \Rightarrow mod_1 \cup mod_3 \preceq mod_2 \cup mod_3.$$

Beweis. Unter der Voraussetzung $mod_1 \preceq mod_2$ wird gezeigt: $mdep_{mod_2 \cup mod_3}(m,p,r,m',p',r') \Rightarrow mdep_{mod_1 \cup mod_3}(m,p,r,m',p',r')$.

Falls $mrm(mod_1) \subsetneq mrm(mod_2)$ hat man $mrm(mod_1 \cup mod_3) = mrm(mod_1) \cup mrm(mod_3) \subsetneq mrm(mod_2) \cup mrm(mod_3) = mrm(mod_2 \cup mod_3)$ und die Behauptung folgt sofort mit Definition 12.

Sei nun $mrm(mod_2) = mrm(mod_1)$ und $mdep_{mod_2 \cup mod_3}(m,p,r,m',p',r')$ eine Abhängigkeit entsprechend Definition 11. Wegen $T_2 \cap T_3 = \emptyset$ und wegen der Definition von $\cup$ gilt dann $mdep_{mod_2}(m,p,r,m',p',r')$ oder $mdep_{mod_3}(m,p,r,m',p',r')$. Mit $mdep_{mod_2}(m,p,r,m',p',r')$ hat man auch $mdep_{mod_1}(m,p,r,m',p',r')$ wegen $mod_2 \preceq mod_1$. Auf Grund der Definition von $\cup$ gilt sowohl mit $mdep_{mod_3}(m,p,r,m',p',r')$ als auch mit $mdep_{mod_1}(m,p,r,m',p',r')$ die Beziehung $mdep_{mod_1 \cup mod_3}(m,p,r,m',p',r')$, und die Behauptung folgt. □

Die Vereinigung ist in der Praxis vor allem dann nützlich, wenn verschiedene (korrekte) Teile eines Gesamt-Cache-Modells entwickelt wurden und in den Teilen unterschiedliche Mengen von Modell-Lesemethoden vorkommen (wobei sich die Mengen nicht gegenseitig enthalten). Durch die Vereinigungsoperation erhält man dann automatisch ein gemeinsames Cache-Modell, das korrekt ist und präziser als die beiden Teile.

Wenn zwei korrekte Cache-Modelle die gleichen Mengen von Modell-Lesemethoden enthalten oder die eine Menge eine Teilmenge der anderen bildet, gilt im allgemeinen leider nicht, dass die Vereinigung präziser ist als eines der beiden Operanden-Cache-Modelle. Es gilt im allgemeinen auch nicht, dass eines der beiden Modelle genau so präzise wie die Vereinigung ist – das folgende einfache Beispiel soll diese Sachverhalte illustrieren.

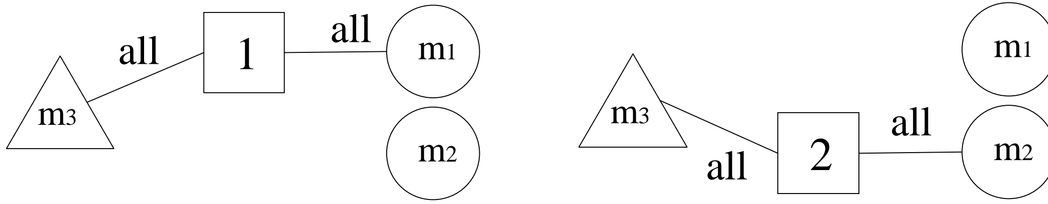


Abbildung 6: Die Cache-Modell-Graphen für mod_3 bzw. mod_4 aus Beispiel 7

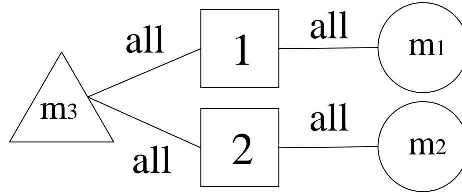


Abbildung 7: Der Cache-Modell-Graph für $mod_3 \cup mod_4$

Beispiel 7. *Sei $M = \{m_1, m_2, m_3\}$ und seien*

$$mod_3 = \{m_1 \mapsto (\texttt{r}, \{(1, \texttt{all})\}), m_2 \mapsto (\texttt{r}, \emptyset), m_3 \mapsto (w, \{(1, \texttt{all})\})\},$$

$$mod_4 = \{m_1 \mapsto (\texttt{r}, \emptyset), m_2 \mapsto (\texttt{r}, \{(2, \texttt{all})\}), m_3 \mapsto (w, \{(2, \texttt{all})\})\}$$

zwei korrekte Cache-Modelle zu M. Man hat also $mrm(mod_1) = mrm(mod_2)$. Als Vereinigung der beiden Cache-Modelle erhält man:

$$mod_3 \cup mod_4 = \{m_1 \mapsto (\texttt{r}, \{(1, \texttt{all})\}),$$

$$m_2 \mapsto (\texttt{r}, \{(2, \texttt{all})\})), m_3 \mapsto (w, \{(1, \texttt{all}), (2, \texttt{all})\})\}.$$

Abbildung 6 zeigt die entsprechenden Cache-Modell-Graphen; den Vereinigungsgraphen erkennt man in Abbildung 7.

In $mdep_{mod_3}$ existieren offenbar nur Modellabhängigkeiten zwischen m_1 und m_3. Entsprechend hat man in $mdep_{mod_4}$ nur Modellabhängigkeiten zwischen m_2 und m_3. Bezüglich $mod_3 \cup mod_4$ kommen aber sowohl die Modellabhängigkeiten zwischen m_1 und m_3 als auch die zwischen m_2 und m_3 vor.

Das Beispiel zeigt, dass die in Definition 13 festgelegte Vereinigung recht unbefriedigend ist, da sie im Falle, dass zwei korrekte Modelle die gleichen Modell-Lesemethoden besitzen, zu einer wesentlichen Verschlechterung der Präzision des Ergebnismodells führen kann. Im folgenden werden deshalb zwei verbesserte Definitionen für Vereinigungsoperatoren eingeführt und untersucht.

Die erste so genannte *verfeinerte Vereinigung* basiert auf der Bestimmung von *potentiellen Modellabhängigkeiten*. Eine potentielle Modellabhängigkeit besteht zwischen einer Modell-Lese und einer Modell-Schreibmethode genau dann, wenn beide auf den selben Index $t \in T$ zugreifen.

Definition 14. *Potentielle Modellabhängigkeit: Sei mod ein Cache-Modell zu einer Dienstschnittstelle M entsprechend Definition 6. Die Menge $if_{mod}(m, m')$ ist dann für zwei Methoden $m, m' \in M$ definiert als*

$$if_{mod}(m, m') = \{\ (t, f) \in mod(m).2 \mid \exists (t, f') \in mod(m').2\ \wedge mod(m).1 \neq mod(m').1\}.$$

Zwischen m und m′ existiert eine potentielle Modellabhängigkeit $pmdep_{mod}(m,m')$ *genau dann, wenn gilt:*

$$m \in mrm(mod) \ \wedge \ if_{mod}(m,m') \neq \emptyset.$$

Die Menge $if_{mod}(m,m')$ bestimmt also alle Tupel (t,f) bezüglich m, durch die eine potentielle Modellabhängigkeit $pmdep_{mod}(m,m')$ hergestellt wird. Für eine potentielle Modellabhängigkeit muss zusätzlich m eine Modell-Lesemethode und m' eine Modellschreibmethode sein.

Potentielle Modellabhängigkeiten sind in den nachfolgenden Betrachtungen von Interesse, weil sei auf (echte) Modellabhängigkeiten hindeuten. Genauer gesagt, kann eine Modellabhängigkeit zwischen zwei Methoden nur dann existieren, wenn sie auch potentiell voneinander abhängen.

In mod_3 aus Beispiel 7 existiert etwa die potentielle Modellabhängigkeit $pmdep_{mod_3}(m_1,m_3)$ mit $if_{mod_3}(m_1,m_3) = \{\texttt{all}\} = if_{mod_3}(m_3,m_1)$ und $if_{mod_3}(m_2,m_3) = \emptyset$ (also *nicht* $pmdep_{mod_3}(m_2,m_3)$). Analog gilt für mod_4: $pmdep_{mod_4}(m_2,m_3)$ und $if_{mod_4}(m_2,m_3) = \{\texttt{all}\} = if_{mod_4}(m_3,m_2)$ (aber *nicht* $pmdep_{mod_4}(m_1,m_3)$).

Lemma 10. *Es gilt:*

$$mdep_{mod}(m,p,r,m',p',r') \Rightarrow (pmdep_{mod}(m,m') \wedge if_{mod}(m,m') \neq \emptyset \wedge if_{mod}(m',m) \neq \emptyset).$$

Beweis. Dies folgt direkt aus Definition 8 und Definition 14. □

Die Umkehrung von Lemma 10 ist im allgemeinen falsch; hierzu sei auf Beispiel 8 verwiesen (siehe unten).

Definition 15. *Verfeinerte Vereinigung: Es gelten nun die gleichen Voraussetzungen wie in Definition 13. Die* verfeinerte Vereinigung *von Cache-Modellen* $mod_1 \uplus mod_2$ *ist dann wie folgt definiert:*

$$mod_1 \uplus mod_2 : M \rightarrow \{\texttt{r}, w\} \times \wp((T_1 \cup T_2) \times (IFun_1 \cup IFun_2)),$$

$$(mod_1 \uplus mod_2)(m) \mapsto (h(m), \{(t,f) \mid (h(m) = w \ \wedge ((t,f) \in mod_1(m).2 \cup mod_2(m).2) \ \vee$$

$$\exists m' \in M : (m \in mrm(mod_1) \wedge m, m' \notin mrm(mod_2) \wedge (t,f) \in if_{mod_1}(m,m')) \vee$$

$$(m \in mrm(mod_2) \wedge m, m' \notin mrm(mod_1) \wedge (t,f) \in if_{mod_2}(m,m')) \vee$$

$$(m \in mrm(mod_1) \cap mrm(mod_2) \wedge (t,f) \in if_{mod_1}(m,m') \wedge if_{mod_2}(m,m') \neq \emptyset))\}), \ wobei$$

$$h(m) = \begin{cases} \texttt{r} & falls\ m \in mrm(mod_1) \cup mrm(mod_2) \\ w & sonst \end{cases}$$

Die Menge der Modell-Schreib- bzw. Modell-Lesemethoden wird bei der verfeinerten Vereinigung genauso wie in Definition 13 berechnet; und für Methoden, die in mod_1 und mod_2 Modell-Schreibmethoden sind, hat man gleiche Tupelmengen $(mod_1 \uplus mod_2)(m).2$ und $(mod_1 \cup mod_2)(m).2$. Nach der obigen Definition gilt weiter: Wenn eine Methode m in mod_1 eine Modell-Lesemethode und in mod_2 eine Modell-Schreibmethode ist, dann werden für eine potentielle Modellabhängigkeit in mod_1 zwischen m und einer weiteren Methode m' alle Tupel aus $if_{mod_1}(m,m')$ in $(mod_1 \uplus mod_2)(m).2$ übernommmen, sofern m' in beiden Modellen eine Modell-Schreibmethode ist. Analoges gilt, wenn man die Rollen von mod_1 und mod_2 vertauscht. Falls eine Methode m sowohl in mod_1 als auch in mod_2 eine Modell-Lesemethode darstellt und in *beiden* Modellen eine potentielle Modellabhängigkeit zwischen m und einer weiteren Methode m' existiert, dann werden die Tupel aus $if_{mod_1}(m,m')$ in $(mod_1 \uplus mod_2)(m).2$ übernommmen (aber nicht die Tupel aus $if_{mod_2}(m,m')$).

Satz 3. *Die verfeinerte Vereinigung zweier korrekter Cache-Modelle $mod_1, mod_2 \in \mathcal{C}$ ist korrekt, also $mod_1 \uplus mod_2 \in \mathcal{C}$. Weiter gilt $mod_1 \cup mod_2 \preceq mod_1 \uplus mod_2$ sowie $mod_1 \preceq mod_1 \uplus mod_2$. Die verfeinerte Vereinigung ist nicht symmetrisch.*[8]

Beweis. Zur Korrektheit von $mod_1 \uplus mod_2$: Auf Grund der Gleichung für h in Definition 15 und der Korrrektheit von mod_1 und mod_2 ist jede Modell-Lesemethode eine Lesemethode.

Sei $dep(m,p,r,m',p',r')$ eine Abhängigkeit entsprechend Definition 11 mit der Modell-Lesemethode $m \in mrm(mod_1 \uplus mod_2)$. Nach Definition 13 kommt dann m in mod_1 oder mod_2 als Modell-Lesemethode vor. Im folgenden werden ohne Beschränkung der Allgemeinheit die Fälle $m \in mrm(mod_1) \setminus mrm(mod_2)$ und $m \in mrm(mod_1) \cap mrm(mod_2)$ untersucht.

Sei nun $m \in mrm(mod_1) \setminus mrm(mod_2)$: Da mod_1 nach Voraussetzung korrekt ist, gilt also $mdep_{mod_1}(m,p,r,m',p',r')$. Nach Definition 8 existiert damit bezüglich mod_1 ein (t,f) zu m und ein (t,f') zu m' mit $f(p,r) \equiv f'(p',r')$. Insbesondere ist dabei (t,f) in $if_{mod_1}(m,m')$.

Falls m' in $mrm(mod_2)$ liegt, kann es nach Definition 11 keine Schreibmethode sein, da mod_2 nach Voraussetzung korrekt ist. Dies steht im Widerspruch zu $dep(m,p,r,m',p',r')$. Falls andererseits $m' \notin mrm(mod_2)$ gilt, ist die zweite Disjunktionsklausel erfüllt und somit erhält man $(t,f) \in (mod_1 \uplus mod_2)(m).2$. Für (t,f') ist entsprechend die erste Disjunktionsklausel erfüllt, so dass $(t,f') \in (mod_1 \uplus mod_2)(m').2$ gilt. Insgesamt gilt damit gemäß Definition 8 die Modellabhängigkeit $mdep_{mod_1}(m,p,r,m',p',r')$ auch bezüglich $mod_1 \uplus mod_2$.

Der Fall $m \in mrm(mod_2) \setminus mrm(mod_1)$ kann analog behandelt werden. Dort kommt statt der zweiten die dritte Disjunktionsklausel für (t,f) zur Anwendung.

Sei nun $m \in mrm(mod_1) \cap mrm(mod_2)$. Wegen der Korrektheit von mod_1 und mod_2 gilt also $mdep_{mod_1}(m,p,r,m',p',r')$ und $mdep_{mod_2}(m,p,r,m',p',r')$. Wegen $mdep_{mod_1}(m,p,r,m',p',r')$ und $m \in mrm(mod_1)$ existiert nach Definition 8 bezüglich mod_1 ein (t,f) zu m und ein (t,f') zu m' mit $f(p,r) \equiv f'(p',r')$. (t,f) liegt offenbar in $if_{mod_1}(m,m')$. Mit einer analogen Argumentation erhält man auch $if_{mod_2}(m,m')$, und insgesamt ist die vierte Disjunktionsklausel in Definition 15 erfüllt bezüglich (t,f) und m. Wie im vorigen Fall folgt außerdem $(t,f') \in (mod_1 \uplus mod_2)(m').2$. Wiederum gilt also die Modellabhängigkeit $mdep_{mod_1}(m,p,r,m',p',r')$ auch bezüglich $mod_1 \uplus mod_2$.

Zusammenfassend wurde für ein $m \in mrm(mod_1 \uplus mod_2)$ gezeigt: $dep(m,p,r,m',p',r') \Rightarrow mdep_{mod_1 \cup mod_2}(m,p,r,m',p',r')$. Also ist $mod_1 \uplus mod_2$ korrekt.

Zur Beziehung $mod_1 \preceq mod_1 \uplus mod_2$: Offenbar gilt $mrm(mod_1) \subseteq mrm(mod_1 \uplus mod_2)$ wegen h in Definition 15. Falls $mrm(mod_1) \subsetneq mrm(mod_1 \uplus mod_2)$ gilt die Aussage auf Grund der Definition von $\preceq$. Sei nun $mrm(mod_1) = mrm(mod_1 \uplus mod_2)$. Auf Grund der Gleichung für h in Definition 15 folgt daraus $mrm(mod_2) \subseteq mrm(mod_1)$.

Es gelte nun $mdep_{mod_1 \uplus mod_2}(m,p,r,m',p',r')$ für die Lesemethode m. Nach Definition 8 existiert damit bezüglich $mod_1 \uplus mod_2$ ein (t,f) zu m und ein (t,f') zu m' mit $f(p,r) \equiv f'(p',r')$. Falls $m \in mrm(mod_2)$ gilt, hat man offenbar $mod_1(m).1 = mod_2(m).1$, und (t,f) muss sich durch die vierte Disjunktionsklausel von Definition 15 für die Menge $(mod_1 \uplus mod_2)(m).2$ qualifiziert haben. Das heißt $(t,f) \in if_{mod_1}(m,m') \subseteq mod_1(m).2$. Falls $m \notin mrm(mod_2)$ gilt, muss sich (t,f) durch die dritte Disjunktionsklausel von Definition 15 für die Menge $(mod_1 \uplus mod_2)(m).2$ qualifiziert haben. Somit muss $(t,f) \in if_{mod_1}(m,m')$ gelten. In jedem Fall bedeutet dies also: $(t,f) \in mod_1(m).2$.

$(t,f') \in (mod_1 \uplus mod_2)(m').2$ muss offenbar aus der ersten Disjunktionsklausel stammen (da m' eine Modell-Schreibmethode ist). Falls (t,f') in $mod_1(m').2$ liegt, hat man auch $mdep_{mod_1}(m,p,r,m',p',r')$, und die Behauptung zu $\preceq$ ist wahr. Andernfalls gilt $(t,f') \in mod_2(m').2$. Dann hat man aber $t \in T_1 \cap T_2$ im Widerspruch zur Voraussetzung von Definition 13 bzw. 15. Für $mdep_{mod_1 \uplus mod_2}(m,p,r,m',p',r')$ gilt also: $(t,f) \in mod_1(m).2$ und $(t,f') \in mod_1(m').2$, und es folgt $mdep_{mod_1}(m,p,r,m',p',r')$. Zusammenfassend gilt also $mdep_{mod_1 \uplus mod_2} \subseteq mdep_{mod_1}$, und es folgt die behauptete Aussage.

[8]Die verfeinerte Vereinigung ist auch assoziativ. Wegen der geringen praktischen Bedeutung dieser Eigenschaft und dem verhältnismäßig großen zugehörigen Beweisaufwand wird die Assoziativität hier aber nicht näher untersucht.

Zu $mod_1 \cup mod_2 \preceq mod_1 \uplus mod_2$: Offenbar gilt durch die identische Gleichung für h in Definition 13 und 15: $mrm(mod_1 \cup mod_2) = mrm(mod_1 \uplus mod_2)$. Es bleibt zu zeigen, dass für ein $m \in M$ gilt: $(mod_1 \uplus mod_2)(m).2 \subseteq (mod_1 \cup mod_2)(m).2$. Für den Fall $(mod_1 \cup mod_2)(m).1 = w$ ist dies klar, da in beiden Definitionen dann nur die erste, identische Disjunktionsklausel anwendbar ist. Bei $(mod_1 \cup mod_2)(m).1 = r$ qualifizieren sich alle Tupel aus $mod_1(m).2$ für $(mod_1 \cup mod_2)(m).2$. Entsprechend qualifiziert sich für $(mod_1 \uplus mod_2)(m).2$ eine Teilmenge von $mod_1(m).2$. Analoge Überlegungen gelten für die Tupel aus $mod_2(m).2$. Damit folgt die Behauptung.

Zur Aussage, dass $\uplus$ nicht symmetrisch ist, sei auf das Beispiel 8 verwiesen. □

Lemma 11. *Monotonie der verfeinerten Vereinigung: Seien $mod_1, mod_2, mod_3 \in \mathcal{C}(M)$ drei korrekte Cache-Modelle für die gleiche Dienstschnittstelle M. Dann gilt:*

$$mod_1 \preceq mod_2 \Rightarrow mod_1 \uplus mod_3 \preceq mod_2 \uplus mod_3.$$

Beweis. Unter der Voraussetzung $mod_1 \preceq mod_2$ wird gezeigt: $mdep_{mod_2 \uplus mod_3}(m, p, r, m', p', r') \Rightarrow mdep_{mod_1 \uplus mod_3}(m, p, r, m', p', r')$.

Falls $mrm(mod_1) \subsetneq mrm(mod_2)$ ist die Argumentation identisch zum entsprechenden Teil des Beweises von Lemma 9.

Sei nun $mrm(mod_2) = mrm(mod_1)$ und $mdep_{mod_2 \uplus mod_3}(m, p, r, m', p', r')$ eine Abhängigkeit entsprechend Definition 11. Wegen $T_2 \cap T_3 = \emptyset$ und wegen der Definition von $\uplus$ gilt dann $mdep_{mod_2}(m, p, r, m', p', r')$ oder $mdep_{mod_3}(m, p, r, m', p', r')$ (oder beides).

Falls dabei $mdep_{mod_2}(m, p, r, m', p', r') \wedge mdep_{mod_3}(m, p, r, m', p', r')$ erfüllt ist, gilt wegen $mod_1 \preceq mod_2$ auch $mdep_{mod_1}(m, p, r, m', p', r')$ und somit sogar $mdep_{mod_1 \uplus mod_3}(m, p, r, m', p', r')$ durch die vierte Disjunktionsklausel von Definition 15. (Man erinnere sich hierzu an Lemma 10 bezüglich mod_3.)

Zum Fall $\neg mdep_{mod_2}(m, p, r, m', p', r') \wedge mdep_{mod_3}(m, p, r, m', p', r')$: Nach Definition 8 existiert bezüglich $mdep_{mod_2 \uplus mod_3}(m, p, r, m', p', r')$ ein (t, f) zu m und ein (t, f') zu m' mit $f(p, r) \equiv f'(p', r')$. Wegen $\neg mdep_{mod_2}(m, p, r, m', p', r')$ und wegen der Definiton von $\uplus$ muss aber $(t, f) \in mod_3(m).2$ bzw. $(t, f') \in mod_3(m).2$ gelten. Bei $\neg mdep_{mod_2}(m, p, r, m', p', r')$ kann $m \in mrm(mod_2)$ oder $m \notin mrm(mod_2)$ unterscheiden. Falls $m \in mrm(mod_2)$ gilt, muss für $(t, f) \in (mod_2 \uplus mod_3)(m).2$ die vierte Disjunktionsklausel zu $\uplus$ gelten, somit stammt (t, f) aus $if_{mod_2}(m, m') \cap if_{mod_3}(m, m')$. Dies steht im Widerspruch zu $T_2 \cap T_3 = \emptyset$. Ist andererseits $m \notin mrm(mod_2) = mrm(mod_1)$, muss sich (t, f) durch die dritte Disjunktionsklausel für $(mod_2 \uplus mod_3)(m).2$ qualifiziert haben. Ebenso liegt dann (t, f) in $(mod_1 \uplus mod_3)(m).2$, und weiter folgt $m' \notin mrm(mod_1)$ aus $m' \notin mrm(mod_2)$. Mithin qualifiziert sich (t, f') sowohl bezüglich $(mod_2 \uplus mod_3)(m').2$ als auch $(mod_1 \uplus mod_3)(m').2$ durch die erste Disjunktionsklausel. Also gilt $mdep_{mod_1 \uplus mod_3}(m, p, r, m', p', r')$.

Zum Fall $mdep_{mod_2}(m, p, r, m', p', r') \wedge \neg mdep_{mod_3}(m, p, r, m', p', r')$: Nach Definition 8 existiert bezüglich $mdep_{mod_2 \uplus mod_3}(m, p, r, m', p', r')$ ein (t, f) zu m und ein (t, f') zu m' mit $f(p, r) \equiv f'(p', r')$. Nun muss natürlich (im Gegensatz zum vorigen Paragraphen) $(t, f) \in mod_2(m).2$ bzw. $(t, f') \in mod_2(m').2$ gelten. (t, f) stammt dann bezüglich der Definition von $\uplus$ entweder aus der zweiten oder der vierten Disjunktionsklausel. Desweiteren gilt wegen $mod_1 \preceq mod_2$ auch $mdep_{mod_1}(m, p, r, m', p', r')$ mit entsprechenden Tupeln (t', g), (t', g') zu m und m', so dass $g(p, r) \equiv g'(p', r')$. Falls sich (t, f) durch die zweite Disjunktionsklausel für $(mod_2 \uplus mod_3)(m).2$ qualifiziert hat, trifft dies auch für (t', g) bezüglich $(mod_1 \uplus mod_3)(m).2$ zu, und außerdem ist dann $m' \notin mrm(mod_3)$. Analoges gilt für den Fall, dass sich (t, f) durch die vierte Disjunktionsklausel qualifiziert hat (ebenfalls mit $m' \notin mrm(mod_3)$). In jedem Fall hat man also $(t', g) \in (mod_1 \uplus mod_3)(m).2$. Ohnehin ist $(t', g') \in (mod_1 \uplus mod_3)(m').2$, da m' in mod_1, mod_2 und mod_3 eine Modell-Schreibmethode ist. Also gilt wiederum $mdep_{mod_1 \uplus mod_3}(m, p, r, m', p', r')$. □

Die verfeinerte Vereinigung von Cache-Modellen ist in der Praxis natürlich automatisiert durchführbar. Sie ist der Vereinigung aus Definition 13 in jedem Fall vorzuziehen, da sie in vielen

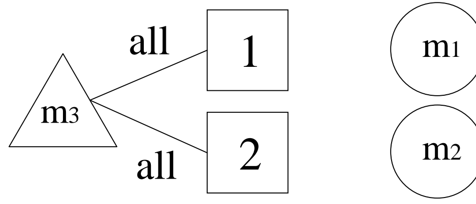


Abbildung 8: Der Cache-Modell-Graph für $mod_3 \uplus mod_4$

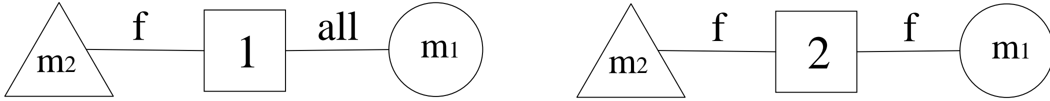


Abbildung 9: Die Cache-Modell-Graphen für mod_5 bzw. mod_6 aus Beispiel 8

Fällen präzisere Cache-Modelle liefert, aber mindestens genauso präzise ist wie die Vereinigung aus Definition 13. Für Beispiel 7 ergibt sich bei Anwendung der verfeinerten Vereinigung:

$$mod_3 \uplus mod_4 = \{m_1 \mapsto (\mathtt{r}, \emptyset), m_2 \mapsto (\mathtt{r}, \emptyset), m_3 \mapsto (\mathtt{w}, \{(1, \mathtt{all}), (2, \mathtt{all})\})\}.$$

Abbildung 8 zeigt den zugehörigen Cache-Modell-Graphen.

Allerdings gilt auch bei der verfeinerten Vereinigung im allgemeinen nicht $mod_2 \preceq mod_1 \uplus mod_2$ für zwei korrekte Cache-Modelle $mod_1, mod_2 \in \mathcal{C}(M)$. Hierzu wieder ein Beispiel.

Beispiel 8. *Sei $M = \{m_1, m_2\}, f \in IFun$ mit $f : P^2 \to (P^2)^\diamond, (p,r) \mapsto (p,r)$, und seien*

$$mod_5 = \{m_1 \mapsto (\mathtt{r}, \{(1, \mathtt{all})\}), m_2 \mapsto (\mathtt{w}, \{(1, f)\})\},$$

$$mod_6 = \{m_1 \mapsto (\mathtt{r}, \{(2, f)\}), m_2 \mapsto (\mathtt{w}, \{(2, f)\})\}$$

zwei korrekte Cache-Modelle zu M. Man hat also $mrm(mod_5) = mrm(mod_6)$ und

$$mod_5 \uplus mod_6 = \{m_1 \mapsto (\mathtt{r}, \{(1, \mathtt{all})\}), m_2 \mapsto (\mathtt{w}, \{(1, f), (2, f)\})\}.$$

Offenbar gilt $mod_6 \not\preceq mod_5 \uplus mod_6$: Angenommen es ist $m_1(p) = m_2(p) = p$ und $m_1(p') = m_2(p') = p'$ für $p, p' \in P$ mit $p \neq p'$. Damit ist $f(p,p) = (p,p) \neq f(p',p') = (p',p')$, und man hat $(m_1, p, p, m_2, p', p') \notin mdep_{mod_6}$, aber $(m_1, p, p, m_2, p', p') \in mdep_{mod_5 \uplus mod_6}$. (Im übrigen gilt hier sogar $mod_5 \uplus mod_6 \prec mod_6$.)

Abbildung 9 zeigt die Cache-Modell-Graphen zu mod_5 und mod_6 und in Abbildung 10 ist der Graph zu $mod_5 \uplus mod_6$ bzw. $mod_6 \uplus mod_5$ dargestellt.

Problematisch ist bei der verfeinerten Vereinigung also der Fall, bei dem $pmdep_{mod_1}(m, m')$ und $pmdep_{mod_2}(m, m')$ zusammen gelten. Die verfeinerte Vereinigung „entscheidet" sich dann sozusagen für die Modellierung der Abhängigkeiten gemäß mod_1. Wenn diese Modellierung über die (t, f)-Tupel weniger präzise ist als die entsprechende Modellierung gemäß mod_2, dann erhält $mod_1 \uplus mod_2$ eventuell eine geringere Präzision also mod_2.

Für das obige Beispiel ist leicht einzusehen, dass man sich bei der Vereinigung von mod_5 und mod_6 besser für $m_1 \mapsto (\mathtt{r}, \{(2, f)\})$ anstatt für $m_1 \mapsto (\mathtt{r}, \{(1, \mathtt{all})\})$ entscheidet. Im Zusammenhang mit der `all`-Funktion lässt sich die Definition der verfeinerten Vereinigung in dieser Hinsicht so verbessern, dass man in einigen Fällen die entsprechenden (t, f)-Tupel aus if_{mod_1} und in anderen

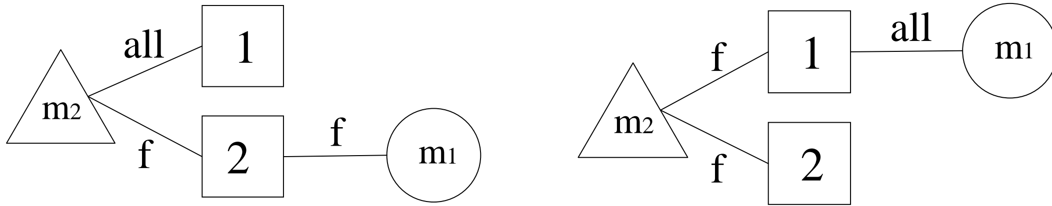


Abbildung 10: Die Cache-Modell-Graphen für $mod_5 \uplus mod_6$ bzw. $mod_6 \uplus mod_5$

Fällen aus if_{mod_2} in die Vereinigung aufnimmt.[9] Im allgemeinen ist das Problem aber leider nicht entscheidbar, da als Indexfunktionen beliebige berechenbare Funktionen vorkommen können.

Beispiel 8 zeigt auch, dass $\uplus$ nicht symmetrisch ist, denn es gilt:

$$mod_6 \uplus mod_5 = \{m_1 \mapsto (\mathtt{r}, \{(2, f\}), m_2 \mapsto (\mathtt{w}, \{(1, f), (2, f)\})\} \neq mod_5 \uplus mod_6.$$

Darüber hinaus kann man aus dem Beispiel ableiten, dass die Umkehrung von Lemma 10 im allgemeinen nicht gilt. Um dies einzusehen, betrachte man mod_6: Offenbar gilt dort $pmdep_{mod_6}(m_1, m_2)$. Aber für p, p' ist, wie bereits oben gesagt, $(m_1, p, p, m_2, p', p') \notin mdep_{mod_6}$.

Sowohl die Vereinigung aus Definition 13 als auch die verfeinerte Vereinigung hinterlassen in den resultierenden Cache-Modellen häufig (t, f)-Tupel, die zu keinen potentiellen Abhängigkeiten beitragen und somit gemäß Lemma 10 gar nicht mehr im Modell benötigt werden. So ergab sich etwa für Beispiel 8 die verfeinerte Vereinigung

$$mod_5 \uplus mod_6 = \{m_1 \mapsto (\mathtt{r}, \{(1, \mathtt{all})\}), m_2 \mapsto (\mathtt{w}, \{(1, f), (2, f)\})\},$$

und offenbar kann man ein äquivalentes Modell konstruieren, indem man $(2, f)$ aus $(mod_5 \uplus mod_6)(m_2).2$ entfernt.

Natürlich können ähnliche Probleme auch bei Cache-Modellen auftreten, die ein Anwendungsentwickler direkt spezifiziert hat. Deshalb wird nun ein Bereinigungsoperator für Cache-Modelle eingeführt, der entsprechend unnötige Tupel aus Cache-Modellen entfernt. In der Praxis ermöglicht die Bereinigung ein effizienteres Cache-Management, da sie eventuell unnötige Auswertungen von Indexfunktionen zur Laufzeit vermeidet. Falls dadurch eine Indexfunktion f überhaupt nicht mehr im Cache-Modell benötigt wird, kann man auch den Speicherplatz für deren Implementierungscode beim Klienten einsparen.

Definition 16. *Bereinigung: Sei mod ein Cache-Modell zur Dienstschnittstelle M. Das* bereinigte *Cache-Modell $mod\star$ ist dann wie folgt definiert:*

$$\forall m \in M : mod\star(m) = (mod(m).1, \{(t, f) \in mod(m).2 \mid \exists m' \in M : (t, f) \in if_{mod}(m, m')\}).$$

Satz 4. *Es gilt $mod \in \mathcal{C} \Leftrightarrow mod\star \in \mathcal{C}$ und $mod \in \mathcal{C} \Rightarrow mod \sim mod\star$. Desweiteren ist $mod\star$ idempotent, also $mod\star = mod\star\star$.*

Beweis. Offenbar ist $mrm(mod) = mrm(mod\star)$ wegen Definition 16.

Als erstes wird gezeigt: $mdep_{mod} \subseteq mdep_{mod\star}$. Sei (m, p, r, m', p', r) ein Element in $mdep_{mod}$. Nach Definition 8 existiert damit bezüglich mod ein (t, f) zu m und ein (t, f') zu m' mit $f(p, r) \equiv f'(p', r')$. Damit ist $(t, f) \in if_{mod}(m, m')$ und somit $(t, f) \in mod\star(m).2$. Analog folgt für (t, f'): $(t, f') \in mod\star(m).2$. Das heißt, (m, p, r, m', p', r) ist auch eine Modellabhängigkeit von $mdep_{mod\star}$.

[9] Da es sich hierbei nur um eine Fallunterscheidung verschiedener Spezialfälle handelt, ist dies aber vom theoretischen Standpunkt aus wenig interessant und wird hier nicht weiter diskutiert.

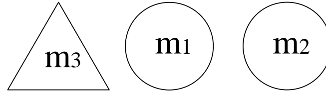


Abbildung 11: Der Cache-Modell-Graph zu $(mod_3 \uplus mod_4)\star$

Nun wird gezeigt: $mdep_{mod\star} \subseteq mdep$. Sei $(m,p,r,m',p',r) \in mdep_{mod\star}$. Also existiert bezüglich $mod\star$ ein (t,f) zu m und ein (t,f') zu m' mit $f(p,r) \equiv f'(p',r')$. Nach Definition 16 ist dann aber $(t,f) \in mod(m).2$. Für (t,f') gilt bezüglich m' eine analoge Beziehung. Insbesondere ist $(m,p,r,m',p',r) \in mdep_{mod}$.

Da $mrm(mod) = mrm(mod\star)$ und $mdep_{mod} = mdep_{mod\star}$, ist $mod\star$ nach Definition 11 korrekt, falls $mod \in \mathcal{C}$. Mithin gilt $mod \sim mod\star$.

Zur Idempotenz: Sei $(t,f) \in mod\star(m).2$. Dann existiert nach Definition 16 ein m' mit $(t,f) \in if_{mod}(m,m')$. Auf Grund von Definition 14 existiert dann auch ein (t,f') mit $(t,f') \in if_{mod}(m',m)$, also gilt $(t,f') \in mod\star(m').2$. Mithin ist (t,f) in $mod\star\star(m).2$, denn nach den Ergebnissen von oben gilt $(t,f) \in if_{mod\star}(m,m')$. $\square$

Die Anwendung der Bereinigung auf Beispiel 7 ergibt:

$$(mod_3 \uplus mod_4)\star = \{m_1 \mapsto (\mathtt{r},\emptyset), m_2 \mapsto (\mathtt{r},\emptyset), m_3 \mapsto (\mathtt{w},\emptyset)\}.$$

In Abbildung 11 sieht man den entsprechend einfachen Cache-Modell-Graphen.

Wendet man die Bereinigung auf Beispiel 8, erhält man

$$(mod_5 \uplus mod_6)\star = \{m_1 \mapsto (\mathtt{r},\{(1,\mathtt{all})\}), m_2 \mapsto (\mathtt{w},\{(1,f)\})\} = mod_5 \text{ bzw.}$$

$$(mod_6 \uplus mod_5)\star = \{m_1 \mapsto (\mathtt{r},\{(2,f)\}), m_2 \mapsto (\mathtt{w},\{(2,f)\})\} = mod_6.$$

Zum Abschluss wird nun die Produktvereinigung eingeführt, deren (korrektes) Resultat mindestens so präzise ist wie jedes ihrer (korrekten) Operandenmodelle. Das Resultat der Produktvereinigung ist auch mindestens so präzise wie das der verfeinerten Vereinigung. Der einzige Nachteil der Produktvereinigung ist die Tatsache, dass sie Funktionenprodukte aus den Indexfunktionen der Operandenmodelle bildet. Somit können die resultierenden Modelle komplex und unüberschaubar werden. Die Definition der Produktvereinigung verwendet die Konzepte *String-Produkt* und *Produktfunktion* – beide werden der Vollständigkeit halber nachfolgend definiert.

Definition 17. *Seien S und T Mengen von Zeichenketten (Strings) über einem gemeinsamen Alphabet α. Das* String-Produkt *ist dann definiert als $S \times T = \{s+t \mid s \in S \wedge t \in T\}$, wobei $+$ den Konkatenationsoperator auf Zeichenketten repräsentiert.*

Definition 18. *Seien f und g Funktionen mit der Signatur $f : A \to B$ bzw. $g : A \to C$. Die* Produktfunktion *$f \otimes g$ ist dann $f \otimes g : A \to B \times C, a \mapsto (f(a), g(a))$ Seien G und H Funktionenmengen mit einem gemeinsamen Definitionsbereich A. Das* Funktionenprodukt *$G \times H$ ist dann definiert als $G \times H = \{\, g \otimes h \mid g \in G \wedge h \in H \,\}$.*

Definition 19. *Produkt-Vereinigung: Es gelten nun die gleichen Voraussetzungen wie in Definition 13. Weiter enthalte das Alphabet α einen Buchstaben $\bullet$, der in keinem Wort aus $T_1 \cup T_2$ vorkommt, also $\forall t \in T_1 \cup T_2 : \bullet \notin t$.*[10] *Die* Produkt-Vereinigung *von Cache-Modellen $mod_1 \sqcup mod_2$ ist dann wie folgt*

[10] Falls notwendig, kann die Bedingung leicht durch Umbenennung entsprechender $t \in T_1$ bzw. $s \in T_2$ hergestellt werden. Dabei ist jedoch darauf zu achten, dass nach wie vor $T_1 \cap T_2 = \emptyset$ gilt und dass sich $|T_1|$ und $|T_2|$ durch die Umbenennung nicht verändern.

definiert:

$$mod_1 \sqcup mod_2 : M \to \{r,w\} \times \wp((T_1 \cup T_2 \cup T_1 \times \{\bullet\} \times T_2) \times (IFun_1 \cup IFun_2 \cup IFun_1 \times IFun_2)),$$

$$(mod_1 \sqcup mod_2)(m) \mapsto (h(m), \{(t,f) \mid (h(m) = w \wedge ((t,f) \in mod_1(m).2 \cup mod_2(m).2) \vee$$
$$\exists m' \in M : (m \in mrm(mod_1) \wedge m, m' \notin mrm(mod_2) \wedge (t,f) \in if_{mod_1}(m,m')) \vee$$
$$(m \in mrm(mod_2) \wedge m, m' \notin mrm(mod_1) \wedge (t,f) \in if_{mod_2}(m,m')) \vee$$
$$(m \in mrm(mod_1) \Leftrightarrow m \in mrm(mod_2) \wedge (t,f) \in p_{mod_1,mod_2}(m,m'))\}), \text{ wobei}$$

$$h(m) = \begin{cases} r & \text{falls } m \in mrm(mod_1) \cup mrm(mod_2) \\ w & \text{sonst} \end{cases} \quad \text{und}$$

$$p_{mod_1,mod_2}(m,m') = \{(s + \bullet + t, g \otimes h) \in (T_1 \times \{\bullet\} \times T_2) \times (IFun_1 \times IFun_2) \mid$$
$$(s,g) \in if_{mod_1}(m,m') \wedge (t,h) \in if_{mod_2}(m,m')\}.$$

Die Produktvereinigung und die verfeinerte Vereinigung aus Definition 15 sind sich recht ähnlich. Für den Fall, dass eine Methode m in mod_1 und mod_2 eine Modell-Lesemethode ist und in *beiden* Modellen eine potentielle Modellabhängigkeit zwischen m und einer weiteren Methode m' existiert, gibt es aber einen wichtigen Unterschied: Während bei die verfeinerte Vereinigung dann nur die Tupel aus $if_{mod_1}(m,m')$ (und nicht die Tupel aus $if_{mod_2}(m,m')$) in $(mod_1 \uplus mod_2)(m).2$ übernimmt, fügt die Produktvereinigung stattdessen die Tupel aus $p_{mod_1,mod_2}(m,m')$ zu $(mod_1 \sqcup mod_2)(m).2$ hinzu. Letztere kombinieren die Tupel aus $if_{mod_1}(m,m')$ und $if_{mod_2}(m,m')$ jeweils paarweise durch Stringkonkatentation bzw. Produktfunktionsbildung. Damit in $mod_1 \sqcup mod_2$ die potentielle Modellabhängigkeit zwischen m und m' erhalten bleibt, fügt die obige Definition zu $(mod_1 \sqcup mod_2)(m').2$ noch die Tupel aus $p_{mod_1,mod_2}(m',m)$ hinzu, die Kombinationen von Tupelpaaren aus $if_{mod_1}(m',m)$ und $if_{mod_2}(m',m)$ darstellen. Die Produktbildung erzeugt eine Modellabhängikeit zwischen m und m' für $mod_1 \sqcup mod_2$ genau dann, wenn es eine entsprechende Modellabhängikeit zwischen m und m' bezüglich mod_1 *und* bezüglich mod_2 besteht. Dies ist die Basis für die positiven Eigenschaften der Produktvereinigung und den Beweis des nachfolgenden Satzes.

Satz 5. *Sei $mod_1, mod_2 \in \mathcal{C}$. Die Produktvereinigung ist korrekt, also $mod_1 \sqcup mod_2 \in \mathcal{C}$, und es gilt $mod_1 \uplus mod_2 \preceq mod_1 \sqcup mod_2$ sowie $mod_1 \preceq mod_1 \sqcup mod_2$ und $mod_2 \preceq mod_1 \sqcup mod_2$. Desweiteren hat man $mod_1 \sqcup mod_2 \sim mod_2 \sqcup mod_1$.*[11]

Beweis. Zur Korrektheit von $mod_1 \sqcup mod_2$: Auf Grund der Gleichung für h in Definition 15 und der Korrrektheit von mod_1 und mod_2 ist jede Modell-Lesemethode eine Lesemethode.

Sei $dep(m,p,r,m',p',r')$ eine Abhängigkeit entsprechend Definition 11 mit der Modell-Lesemethode $m \in mrm(mod_1 \sqcup mod_2)$. Nach Definition 19 kommt dann m in mod_1 oder mod_2 als Modell-Lesemethode vor. Im folgenden werden ohne Beschränkung der Allgemeinheit die Fälle $m \in mrm(mod_1) \setminus mrm(mod_2)$ und $m \in mrm(mod_1) \cap mrm(mod_2)$ untersucht.

Sei nun $m \in mrm(mod_1) \setminus mrm(mod_2)$: Da mod_1 nach Voraussetzung korrekt ist, gilt also $mdep_{mod_1}(m,p,r,m',p',r')$. Nach Definition 8 existiert damit bezüglich mod_1 ein (t,f) zu m und ein (t,f') zu m' mit $f(p,r) \equiv f'(p',r')$. Insbesondere ist dabei (t,f) in $if_{mod_1}(m,m')$.

Falls m' in $mrm(mod_2)$ liegt, kann es nach Definition 11 keine Schreibmethode sein, da mod_2 nach Voraussetzung korrekt ist. Dies steht im Widerspruch zu $dep(m,p,r,m',p',r')$. Falls andererseits $m' \notin mrm(mod_2)$ gilt, ist die zweite Disjunktionsklausel erfüllt und somit erhält man $(t,f) \in (mod_1 \sqcup mod_2)(m).2$. Für (t,f') ist entsprechend die erste Disjunktionsklausel erfüllt, so dass

[11] Wie bei der verfeinerten Vereinigung könnte man auch hier die Assoziativität nachweisen. Wegen der geringen praktischen Bedeutung dieser Eigenschaft und des hohen Beweisumfangs wurde jedoch darauf verzichtet.

$(t,f') \in (mod_1 \sqcup mod_2)(m').2$ gilt. Insgesamt gilt damit gemäß Definition 8 die Modellabhängigkeit $mdep_{mod_1}(m,p,r,m',p',r')$ auch bezüglich $mod_1 \sqcup mod_2$.

Der Fall $m \in mrm(mod_2) \setminus mrm(mod_1)$ kann analog behandelt werden. Dort kommt statt der zweiten die dritte Disjunktionsklausel für (t,f) zur Anwendung.[12].

Sei nun $m \in mrm(mod_1) \cap mrm(mod_2)$. Wegen der Korrektheit von mod_1 und mod_2 gilt also $mdep_{mod_1}(m,p,r,m',p',r')$ und $mdep_{mod_2}(m,p,r,m',p',r')$. Wegen $mdep_{mod_1}(m,p,r,m',p',r')$ und $m \in mrm(mod_1)$ existiert nach Definition 8 bezüglich mod_1 ein (s,g) zu m und ein (s,g') zu m' mit $g(p,r) \equiv g'(p',r')$ $(s \in T_1)$. Analoges gilt für mod_2: Hier hat man $pmdep_{mod_2}(m,m')$ sowie ein (t,h) zu m und ein (t,h') zu m' mit $h(p,r) \equiv h'(p',r')$ $(t \in T_2)$. (s,g) liegt offenbar in $if_{mod_1}(m,m')$ und (t,h) liegt in $if_{mod_2}(m,m')$. Somit ist $(s+\bullet+t, g\otimes h)$ in p_{mod_1,mod_2} und die vierte Disjunktionsklausel ist für $(s+\bullet+t, g\otimes h)$ bezüglich Definition 19 erfüllt, also $(s+\bullet+t, g\otimes h) \in (mod_1 \sqcup mod_2)(m).2$. Analog kann man argumentieren, dass $(s+\bullet+t, g'\otimes h')$ in $(mod_1 \sqcup mod_2)(m').2$ liegt. Mit $g(p,r) \equiv g'(p',r')$ und $h(p,r) \equiv h'(p',r')$ gilt nach Definition 18 auch $(g\otimes h)(p,r) \equiv (g'\otimes h')(p',r')$ und somit insbesondere $mdep_{mod_1 \sqcup mod_2}(m,p,r,m',p',r')$.

Zusammenfassend wurde für ein $m \in mrm(mod_1 \sqcup mod_2)$ gezeigt: $dep(m,p,r,m',p',r') \Rightarrow mdep_{mod_1 \sqcup mod_2}(m,p,r,m',p',r')$. Also ist $mod_1 \sqcup mod_2$ korrekt.

Zu den Beziehungen $mod_1 \preceq mod_1 \sqcup mod_2$ und $mod_2 \preceq mod_1 \sqcup mod_2$: Ohne Beschränkung der Allgemeinheit wird lediglich $mod_1 \preceq mod_1 \sqcup mod_2$ untersucht. Die Argumentation für $mod_2 \preceq mod_1 \sqcup mod_2$ ist wegen der Symmetrie von Definition 19 analog.

Offenbar gilt $mrm(mod_1) \subseteq mrm(mod_1 \sqcup mod_2)$ wegen h in Definition 19. Falls $mrm(mod_1) \subsetneq mrm(mod_1 \sqcup mod_2)$ gilt die Aussage auf Grund der Definition von $\preceq$. Sei nun $mrm(mod_1) = mrm(mod_1 \sqcup mod_2)$. Auf Grund der Gleichung für h in Definition 19 folgt daraus $mrm(mod_2) \subseteq mrm(mod_1)$.

Es gelte nun $mdep_{mod_1 \sqcup mod_2}(m,p,r,m',p',r')$ für die Lesemethode m. Nach Definition 8 existiert damit bezüglich $mod_1 \sqcup mod_2$ ein (t,f) zu m und ein (t,f') zu m' $f(p,r) \equiv f'(p',r')$.

Falls $m \notin mrm(mod_2)$ gilt, muss sich (t,f) durch die zweite Disjunktionsklausel von Definition 19 für die Menge $(mod_1 \sqcup mod_2)(m).2$ qualifiziert haben. Somit muss $(t,f) \in if_{mod_1}(m,m') \subseteq mod_1(m).2$ gelten. $(t,f') \in (mod_1 \sqcup mod_2)(m').2$ muss offenbar aus der ersten oder der vierten Disjunktionsklausel stammen (da m' eine Modell-Schreibmethode ist). Angenommen (t,f') stammt aus der vierten Disjunktionsklausel. Dann kann (t,f) aber nicht gleich $(u+\bullet+v, g\otimes h)$ sein (mit $u+\bullet+v \in T_1 \times \{\bullet\} \times T_2$), denn sonst wäre $\bullet$ in einem Wort aus $T_1 \cup T_2$ im Widerspruch zur Voraussetzung aus Definition 19. Stamme also (t,f') aus der ersten und (t,f) aus der zweiten Disjunktionsklausel von Definiton 19. Dann liegt (t,f') in $mod_1(m').2$ und man hat auch $mdep_{mod_1}(m,p,r,m',p',r')$. Die Behauptung zu $\preceq$ ist also wahr.

Falls $m \in mrm(mod_2)$ gilt, hat man offenbar $m \in mrm(mod_1)$, und (t,f) muss sich durch die vierte Disjunktionsklausel von Definition 19 für die Menge $(mod_1 \sqcup mod_2)(m).2$ qualifiziert haben. Das heißt $(t,f) = (u+\bullet+v, g\otimes h)$ mit $(u,g) \in if_{mod_1}(m,m') \subseteq mod_1(m).2$. Angenommen (t,f') stammt aus der ersten Disjunktionsklausel. Dann wäre mit $t = u+\bullet+v$ das Symbol $\bullet$ in T_1 oder T_2 im Widerspruch zu Definition 19. Also stammt $(t,f') = (u'+\bullet+v', g'\otimes h')$ aus der vierten Disjunktionsklausel, und somit gilt $u = u'$ bzw. $v = v'$ (wegen $\forall t \in T_1 \cup T_2 : \bullet \notin t$). Mit der eingangs gestellten Annahme $mdep_{mod_1 \sqcup mod_2}(m,p,r,m',p',r')$ und der Folge $f(p,r) \equiv f'(p',r')$ erhält man $(g\otimes h)(p,r) \equiv (g'\otimes h')(p',r')$ und weiter $g(p,r) \equiv g'(p',r')$. Da (t,f') aus der vierten Disjunktionsklausel stammt, folgt mit der Definition von $p_{mod_1,mod_2}(m',m)$ gerade $(u,g') \in if_{mod_1}(m',m) \subseteq mod_1(m').2$. Insgesamt hat man $(u,g) \in mod_1(m).2$, $(u,g') \in mod_1(m').2$ und $g(p,r) \equiv g'(p',r')$. Also gilt auch für diesen Fall $mdep_{mod_1}(m,p,r,m',p',r')$.

Zusammenfassend gilt $mdep_{mod_1 \sqcup mod_2} \subseteq mdep_{mod_1}$, und es folgt die behauptete Aussage.

Zu $mod_1 \uplus mod_2 \preceq mod_1 \sqcup mod_2$: Offenbar gilt durch die identische Gleichung für h in Definition 13 und 15: $mrm(mod_1 \cup mod_2) = mrm(mod_1 \sqcup mod_2)$. Im folgenden wird also gezeigt

[12]Die Argumentation für den Fall $m \in mrm(mod_1) \setminus mrm(mod_2)$ ist also völlig analog zum Beweis von Satz 3

$mdep_{mod_1 \sqcup mod_2}(m,p,r,m',p',r') \subseteq mdep_{mod_1 \uplus mod_2}(m,p,r,m',p',r')$.

Es gelte nun $mdep_{mod_1 \sqcup mod_2}(m,p,r,m',p',r')$ für die Lesemethode m. Nach Definition 8 existiert damit bezüglich $mod_1 \sqcup mod_2$ ein (t,f) zu m und ein (t,f') zu m' $f(p,r) \equiv f'(p',r')$.

Falls $mod_1(m).1 \neq mod_2(m).1$, kann sowohl bezüglich $(mod_1 \sqcup mod_2)(m).2$ als auch bezüglich $(mod_1 \uplus mod_2)(m).2$ nur die identische zweite bzw. dritte Disjunktionsklausel für (t,f) gelten. Da in beiden Definition auch die erste Disjunktionsklausel identisch ist, hat man $(t,f') \in (mod_1 \uplus mod_2)(m').2$ und somit folgt die Behauptung.

Sei nun also $mod_1(m).1 = mod_2(m).1$ dann haben (t,f) bzw. (t,f'), wie weiter oben bereits eingesehen, die Form $(t,f) = (u + \bullet + v, g \otimes h)$ und $(t,f') = (u + \bullet + v, g' \otimes h')$. Weiter wurde argumentiert, dass dann $(u,g) \in mod_1(m).2$, $(u,g') \in mod_1(m').2$ und $g(p,r) \equiv g'(p',r')$ gilt (also auch $(u,g) \in if_{mod_1}(m,m')$). Völlig analog trifft dann aber auch $(v,h) \in mod_2(m).2$ und $(v,h') \in mod_2(m').2$ zu und insbesondere $if_{mod_2}(m,m') \neq \emptyset$. Damit gilt die vierte Disjunktionsklausel in Definition 15 für $(mod_1 \uplus mod_2)(m).2$ und $(u,g) \in (mod_1 \uplus mod_2)(m).2$. Ohnehin ist wegen der ersten Disjunktionsklausel von Definition 15 $(u,g') \in (mod_1 \uplus mod_2)(m').2$. In der Tat gilt also $mdep_{mod_1 \uplus mod_2}(m,p,r,m',p',r')$.

Zur Symmetrie $mod_1 \sqcup mod_2 \sim mod_2 \sqcup mod_1$: Diese Aussage folgt direkt aus der Symmetrie von Definition 19 bezüglich der Operanden mod_1 und mod_2. □

Lemma 12. *Monotonie der Produktvereinigung: Seien $mod_1, mod_2, mod_3 \in \mathcal{C}(M)$ drei korrekte Cache-Modelle für die gleiche Dienstschnittstelle M. Dann gilt:*

$$mod_1 \preceq mod_2 \Rightarrow mod_1 \sqcup mod_3 \preceq mod_2 \sqcup mod_3.$$

Beweis. Da Definition 15 und Definition 19 zum Teil übereinstimmen, ist auch der Beweis des obigen Lemmas ähnlich zum Beweis von Lemma 11. Es genügt deshalb die Fälle $mdep_{mod_2}(m,p,r,m',p',r') \wedge \neg mdep_{mod_3}(m,p,r,m',p',r')$ und $\neg mdep_{mod_2}(m,p,r,m',p',r') \wedge mdep_{mod_3}(m,p,r,m',p',r')$ für $mrm(mod_2) = mrm(mod_1)$ genauer zu betrachten, um $mdep_{mod_2 \sqcup mod_3}(m,p,r,m',p',r') \Rightarrow mdep_{mod_1 \sqcup mod_3}(m,p,r,m',p',r')$ nachzuweisen.

Zum Fall $\neg mdep_{mod_2}(m,p,r,m',p',r') \wedge mdep_{mod_3}(m,p,r,m',p',r')$: Nach Definition 8 existiert bezüglich $mdep_{mod_2 \sqcup mod_3}(m,p,r,m',p',r')$ ein (t,f) zu m und ein (t,f') zu m' mit $f(p,r) \equiv f'(p',r')$. Wegen $\neg mdep_{mod_2}(m,p,r,m',p',r')$ und wegen der Definiton von $\sqcup$ muss aber $(t,f) \in mod_3(m).2$ bzw. $(t,f') \in mod_3(m').2$ gelten. Bei $\neg mdep_{mod_2}(m,p,r,m',p',r')$ kann $m \in mrm(mod_2)$ oder $m \notin mrm(mod_2)$ unterscheiden. Falls $m \in mrm(mod_2)$ gilt, muss für $(t,f) \in (mod_2 \uplus mod_3)(m).2$ die vierte Disjunktionsklausel zu $\sqcup$ gelten. Somit wäre $\bullet \in t$ im Widerspruch zu $t \in T_3$ und der entsprechenden Voraussetzung von Definition 19. Ist andererseits $m \notin mrm(mod_2) = mrm(mod_1)$, muss sich (t,f) durch die dritte Disjunktionsklausel für $(mod_2 \sqcup mod_3)(m).2$ qualifiziert haben. Ebenso liegt dann (t,f) in $(mod_1 \sqcup mod_3)(m).2$, und weiter folgt $m' \notin mrm(mod_1)$ aus $m' \notin mrm(mod_2)$. Mithin qualifiziert sich (t,f') sowohl bezüglich $(mod_2 \sqcup mod_3)(m').2$ als auch $(mod_1 \sqcup mod_3)(m').2$ durch die erste Disjunktionsklausel. Also gilt $mdep_{mod_1 \sqcup mod_3}(m,p,r,m',p',r')$.

Zum Fall $mdep_{mod_2}(m,p,r,m',p',r') \wedge \neg mdep_{mod_3}(m,p,r,m',p',r')$: Nach Definition 8 existiert bezüglich $mdep_{mod_2 \sqcup mod_3}(m,p,r,m',p',r')$ ein (t,f) zu m und ein (t,f') zu m' mit $f(p,r) \equiv f'(p',r')$. Nun muss (im Gegensatz zum vorigen Paragraphen) $(t,f) \in mod_2(m).2$ bzw. $(t,f') \in mod_2(m').2$ gelten. Angenommen es würde stattdessen $(t,f) = (s \bullet u, e \otimes h)$ bzw. $(t,f') = (s \bullet v, e' \otimes h')$ mit $(u,h) \in mod_3(m).2$ und $(u,h') \in mod_3(m').2$ gelten. Dann hätte man $h(p,r) \equiv h(p',r')$ und weiter $mdep_{mod_3}(m,p,r,m',p',r')$ im Widerspruch zur Voraussetzung. (t,f) stammt also bezüglich der Definition von $\sqcup$ aus der zweiten Disjunktionsklausel. Desweiteren gilt wegen $mod_1 \preceq mod_2$ gilt auch $mdep_{mod_1}(m,p,r,m',p',r')$ mit entsprechenden Tupeln (t',g), (t',g') zu m und m', so dass $g(p,r) \equiv g'(p',r')$. Da sich (t,f) durch die zweite Disjunktionsklausel für $(mod_2 \sqcup mod_3)(m).2$ qualifiziert hat, trifft dies auch für (t',g) bezüglich $(mod_1 \sqcup mod_3)(m).2$ zu, und außerdem ist dann $m' \notin mrm(mod_3)$. Man hat also $(t',g) \in (mod_1 \sqcup mod_3)(m).2$. Ohnehin ist $(t',g') \in (mod_1 \sqcup$

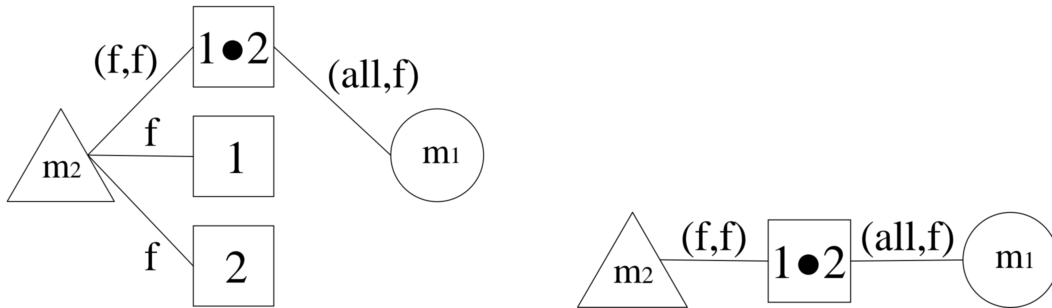


Abbildung 12: Die Cache-Modell-Graphen für $mod_5 \sqcup mod_6$ bzw. $(mod_5 \sqcup mod_6)\star$

$mod_3)(m').2$, da m' in mod_1, mod_2 und mod_3 eine Modell-Schreibmethode ist. Also gilt wiederum $mdep_{mod_1 \sqcup mod_3}(m, p, r, m', p', r')$. □

Beispiel 9. *Es werden hier wieder die Voraussetzungen von Beispiel 8 aufgegriffen. Das Resultat der Produktvereinigung ist dann:*

$$mod_5 \sqcup mod_6 = \{m_1 \mapsto (r, \{(1 \bullet 2, (\mathtt{all}, f))\}), m_2 \mapsto (w, \{(1, f), (2, f), (1 \bullet 2, (f, f))\})\} \text{ bzw.}$$

$$mod_6 \sqcup mod_5 = \{m_1 \mapsto (r, \{(2 \bullet 1, (f, \mathtt{all}))\}), m_2 \mapsto (w, \{(1, f), (2, f), (2 \bullet 1, (f, f))\})\}.$$

Auch hier bietet es sich natürlich an, die Bereinigung anzuwenden:

$$(mod_5 \sqcup mod_6)\star = \{m_1 \mapsto (r, \{(1 \bullet 2, (\mathtt{all}, f))\}), m_2 \mapsto (w, \{(1 \bullet 2, (f, f))\})\} \text{ bzw.}$$

$$(mod_6 \sqcup mod_5)\star = \{m_1 \mapsto (r, \{(2 \bullet 1, (f, \mathtt{all}))\}), m_2 \mapsto (w, \{(2 \bullet 1, (f, f))\})\}.$$

Abbildung 12 zeigt die Cache-Modell-Graphen zu $mod_5 \sqcup mod_6$ und $(mod_5 \sqcup mod_6)\star$. (Die Graphen zu $mod_6 \sqcup mod_5$ und $(mod_6 \sqcup mod_5)\star$ haben eine analoge Struktur.)

6 Normalisierung von Cache-Modellen

Ähnlich wie bei anderen Modellen aus der Informatik stellt sich auch bei Cache-Modellen die Frage, ob sich diese auf eine bestimmte Normalform zurückführen lassen. Wie man im folgenden sehen wird, trifft dies auch bei Cache-Modellen zu: man kann jedes korrekte Cache-Modell in ein äquivalentes Cache-Modell transformieren, das nur einen einzigen abstrakten Index verwendet.

Um zu derartig normalisierten Cache-Modellen zu gelangen, muss jedoch zunächst der Begriff der Resultatmenge ergänzt werden. Dabei fügt man Resultatmengen neben dem Joker (aus Definition 5) noch ein weiteres „besonderes" Element hinzu – die sogenannte Niete. Normalisierte Cache-Modelle entstehen dann durch die Einführung von Produktfunktionen, die auf dem kartesischen Produkt aller Indizes des Ausgangsmodells operieren. Das kartesische Produkt der ursprünglichen abstrakten Indizes bildet also *den* (einzigen) abstrakten Index des entsprechenden normalisierten Modells.

Im Gegensatz zu Normalformen aus anderen Bereichen der Informatik (wie etwa der Normalisierung von Relationen) hat die Normalsierung von Cache-Modellen nur geringe praktische Relevanz, und aus diesem Grund wurde sie ans Ende der formalen Betrachtungen dieses Berichts gestellt. Die Gründe für die geringe Relevanz werden nach der Definition der Normalform diskutiert. Die Normalisierung von Cache-Modellen ist vor allem von theoretischen Interesse, weil sie eine Einsicht in die Mächtigkeit von Cache-Modellen liefert.

Definition 20. *Erweiterte Resultatmenge: Es sei $R^\diamond$ eine Resultatmenge gemäß Definition 5. $R^\diamond$ wird als* erweiterte Resultatmenge *bezeichnet, genau dann wenn gilt:*

$$\exists r \in R^\diamond : \forall q \in R^\diamond : (q \neq \diamond \wedge q \neq r) \Rightarrow \neg(r \equiv q).$$

Falls mehrere Elemente in $r \in R$ mit der oben genannten Eigenschaft existieren, sei genau eines davon ausgezeichnet und als Niete *bezeichnet. Die Niete wird durch das Symbol $*$ notiert bzw. $*_{R^\diamond}$, um deutlich zu machen, dass die Niete zu $R^\diamond$ gehört. Falls die Menge $R^\diamond$ keine Niete enthält, lässt $R^\diamond$ sich durch Hinzunahme eines entsprechenden Elements zu einer erweiterten Resultatmenge vervollständigen. Eine derartige Vervollständigung wird mit $R^{\diamond *}$ bezeichnet; falls $R^\diamond$ bereits eine erweiterte Resultatmenge ist, gilt $R^\diamond = R^{\diamond *}$.*

Da offenbar jede erweiterte Resultatmenge eine Resultmenge ist, kann man entsprechend Definiton 6 natürlich auch erweiterte Resultatmengen bei der Spezifikation von Cache-Modellen verwenden. Ähnlich wie die Funktion `all` aus Definition 9, die auf dem Joker-Element basiert, lässt sich nun eine weitere generische Funktion `none` einführen. Sie macht von der Niete einer erweiterten Resultatmenge Gebrauch.

Definition 21. *Sei $IFun$ die Menge von Indexfunktionen für ein Cache-Modell mod. Die Indexfunktion $\texttt{none}_{R^{\diamond *}} \in IFun$ ist dann auf einer erweiterten Resultatmenge definiert durch den Ausdruck: $\forall p, r \in P : \texttt{none}(p, r) = *_{R^\diamond}$. Da die Niete $*_{R^\diamond}$ für jede erweiterte Resultatmenge $R^{\diamond *}$ definiert ist, kann man sich die Angabe der erweiterten Resultatmenge bezüglich $\texttt{none}_{R^{\diamond *}}$ auch sparen. Im folgenden wird daher nur noch von der (generischen) Indexfunktion `none` gesprochen.*

Mit `all`, `none` und der Bildung von Produktfunktionen entsprechend Definition 18 ist man in der Lage, die oben angekündigte Normalform für Cache-Modelle zu definieren.

Definition 22. *Sei mod ein Cache-Modell entsprechend Definition 6 mit $T = \{t_1, \ldots, t_n\}$. Jedes Element $f \in IFun$ besitze die Signatur $f : P \times P \to R_f^{\diamond *}$, wobei $R_f^{\diamond *}$ eine erweiterte Resultatmenge bezüglich f bezeichnet.*[13] *Desweiteren gelte $\forall f \in IFun : \forall p, r \in P : f(p, r) \neq *_{R_f^\diamond}$ und $\bullet$ sei ein Zeichen aus α mit $\forall t \in T : \bullet \notin t$. Das* normalisierte Cache-Modell *$mod\circ$ ist dann wie folgt definiert:*

$$mod\circ : m \mapsto (mod(m).1,$$

$$\{(t_1 + \bullet + \ \ldots + \bullet + t_n, h(t_1, m, s, f) \otimes \ \ldots \otimes h(t_n, m, s, f)) \mid (s, f) \in mod(m).2\}), \text{ wobei}$$

$$h(t, m, s, f) = \begin{cases} f & \text{falls } t = s \\ \texttt{all} & \text{falls } t \neq s \wedge m \in mrm(mod) \\ \texttt{none} & \text{sonst.} \end{cases}$$

`none` ist demnach in der Menge der Indexfunktionen $IFun$ zu $mod\circ$ enthalten, nicht aber in der entsprechenden Menge zu mod.

Satz 6. *Es sei mod ein Cache-Modell entsprechend Definition 6 und $mod\circ$ das zugehörige, normalisierte Cache-Modell. Dann gilt:*

$$\forall m, m' \in M : \forall p, r, p', r' \in P : mdep_{mod}(m, p, r, m', p', r') \Leftrightarrow mdep_{mod\circ}(m, p, r, m', p', r').$$

Falls mod korrekt ist, ergibt sich insbesondere $mod \sim mod\circ$.

[13] Falls für f keine erweiterte Resultatmenge vorliegt, kann diese entsprechend Definition 20 erzeugt werden.

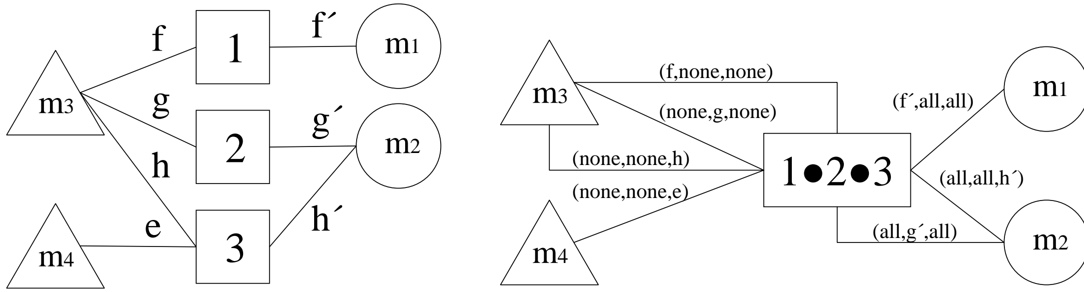


Abbildung 13: Die Cache-Modell-Graphen für mod_7 und $mod_7\circ$

Beweis. „⇒“: Es gelte $mdep_{mod}(m,p,r,m',p',r')$ für geeignete m, m', p, p' und r, r'. Nach Definition 8 existiert damit bezüglich mod ein (t_i, f) zu m und ein (t_i, f') zu m' mit $f(p,r) \equiv f'(p',r')$. Auf Grund der Definition von $\circ$ gilt offenbar $mod(m).1 = mod \circ (m).1$ bzw. $mod(m').1 = mod \circ (m').1$. Weiter hat man $(t_1 \bullet \ldots \bullet t_n, (\texttt{all}, \ldots, f, \ldots, \texttt{all})) \in mod \circ (m)$ bzw. $(t_1 \bullet \ldots \bullet t_n, (\texttt{none}, \ldots, f', \ldots, \texttt{none})) \in mod \circ (m')$ mit f und f' jeweils and Position i der entsprechenden n-Tupel. Wegen $\texttt{all}(p,r) \equiv \texttt{none}(p,r)$ ergibt sich auch $(\texttt{all}, \ldots, f, \ldots, \texttt{all})(p,r) \equiv (\texttt{none}, \ldots, f', \ldots, \texttt{none})(p,r)$ und somit $mdep_{mod\circ}(m,p,r,m',p',r')$.

„⇐“: Es gelte $mdep_{mod\circ}(m,p,r,m',p',r')$ für geeignete m, m', p, p' und r, r'. Nach Definition 8 und Definition 22 existiert damit bezüglich $mod\circ$ ein $(t_1 \bullet \ldots \bullet t_n, g)$ zu m und ein $(t_1 \bullet \ldots \bullet t_n, g')$ zu m' mit $g(p,r) \equiv g'(p',r')$. Auf Grund der Definition von $\circ$ muss dabei $g = (\texttt{all}, \ldots, f, \ldots, \texttt{all})$ mit $(t_i, f) \in mod(m).2$ bzw. $g' = (\texttt{none}, \ldots, f', \ldots, \texttt{none})$ mit $(t_i, f') \in mod(m').2$ gelten, wobei i die Position von f im Tupel g bzw. von f' im Tupel g' bezeichnet. Angenommen f und f' kämen an verschiedenen Positionen in g bzw. g' vor. Dann hätte man $f(p,r) \equiv none(p,r) = *$ im Widerspruch zur Voraussetzung von Definition 22. Somit folgt aus $g(p,r) \equiv g'(p',r')$ für die i-ten Elemente der Produktfunktionen g und g': $f(p,r) \equiv f'(p',r')$. Mit $mrm(mod) = mrm(mod\circ)$ gilt dann sogar $mdep_{mod}(m,p,r,m',p',r')$. □

Beispiel 10. *Abbildung 13 zeigt den Cache-Modell-Graphen für das Cache-Modell mod_7 und das zugehörige normalisierte Cache-Modell $mod_7\circ$. Auf die formale Darstellung der Cache-Modelle wurde der Einfachheit halber verzichtet.*

Aus dem Beispiel ist ersichtlich, dass es sich bei der Normalisierung um eine rein formale Umstrukturierung von Cache-Modellen handelt. Sie ist informationserhaltend in dem Sinn, dass das ursprüngliche Cache-Modell komplett wieder rekonstruiert werden kann.

Wie bereits eingangs erwähnt, belegt der obige Satz zur Normalisierung vor allem, dass man „im Prinzip“ auf die Möglichkeit zur Definition mehrerer Indizes im Formalismus von Cache-Modellen und insbesondere in Definition 6 hätte verzichten können. Mit der Verwendung eines einziges Index und der Einführung der Niete kann man bereits die gleiche Ausdruckskraft und Präzision erreichen. Aus praktischer Sicht ist es jedoch viel sinnvoller, mehrere abstrakte Indizes für ein Cache-Modell zuzulassen, da ein Entwickler das Cache-Modell dann wesentlich intuitiver und verständlicher entwerfen kann. Aus diesem Grund wurde sowohl im Formalismus als auch bei der praktischen Umsetzung der bereits bekannte Weg mit vielen abstrakten Indizes für ein Cache-Modell eingeschlagen.

Auch bei der Generierung der Cache-Modell-Klassen entsprechend Abschnitt 2 ist die etwaige Anwendung der Normalisierung zur Vereinfachung des Laufzeitsystems *nicht* von Nutzen.

In der Praxis ist es beim Einsatz von Cache-Modellen zur Systemlaufzeit wichtig, dass man zu einem Scheibmethodenaufruf mit geringen Aufwand die davon abhängigen, gecachten Lese-

methodenresultate invalidieren kann. Hierzu muss das System zu dem Resulat einer Indexfunktionsauswertung $h = f'(p,r)$, die durch einen Schreibmethodenaufruf (m',p,r) ausgelöst wird, alle gecachten Methodenresultate finden, die den Wert h im gleichen abstrakten Index zuvor „gelesen" haben. Bei gewöhnlichen (nicht normalisierten) Indizes eignen sich hierzu vor allem Hash-Tabellen mit entsprechenden Hash-Funktionen für Werte wie h. Nach der Normalisierung eines Cache-Modells ist die Bestimmung geeigneter Hash-Funktionen aber eventuell schwierig oder gar nicht möglich, denn die anzuwendenden Hash-Funktionen müssen komplexe Gleichheitsbeziehungen berücksichtigen, die im Zusammenhang mit der Produktbildung für die Funktionen `all` und `none` entstehen. Aus diesem Grund bietet sich die Normalisierung auch bei der Generierung der Cache-Modell-Klassen nicht an.

Zum Abschluss seien noch zwei interessante Eigenschaften normalisierter Cache-Modelle erwähnt:

Lemma 13. *Stabilität normalisierter Cache-Modelle: Sei mod ein Cache-Modell entsprechend Definition 6. Dann gilt:* $mod\circ = mod \circ \circ$.

Beweis. $mod\circ$ besitzt gemäß Definition 22 lediglich einen abstrakten Index. Wendet man $\circ$ auf $mod\circ$ an, hat man $|T| = |\{t_1\}| = 1$. Die Behauptung folgt dann sofort durch Einsetzen dieser Bedingung in Definition 22. □

Lemma 14. *Stabilität unter der Produktvereinigung: Es seien mod_1 und mod_2 zwei korrekte Cache-Modelle für die gleiche Dienstschnittstelle M, also $mod_1, mod_2 \in \mathcal{C}(M)$, und es sei $mrm(mod_1) = mrm(mod_2)$. Dann gilt: $(mod_1 \circ \sqcup\, mod_2\circ)\star = (mod_1 \circ \sqcup\, mod_2\circ)\circ$.*

Beweis. Es wird gezeigt: $mod_1 \circ \sqcup\, mod_2\circ$ hat lediglich einen abstrakten Index. Die Behauptung folgt dann durch Einsetzen dieser Bedingung in Definition 22 (siehe auch Beweis von Lemma 13).

Offenbar besitzen $mod_1\circ$ und $mod_2\circ$ jeweils nur einen abstrakten Index. Diese seien mit s und t bezeichnet. Wegen $mrm(mod_1\circ) = mrm(mod_1) = mrm(mod_2) = mrm(mod_2\circ)$ kann ein $(u,g) \in (mod_1 \circ \sqcup\, mod_2\circ)(m).2$ *nicht* durch die zweite oder die dritte Disjunktionsklausel aus Definition 19 in $(mod_1 \circ \sqcup\, mod_2\circ)(m).2$ gelangt sein. Damit gibt es kein (s,f) oder (t,f) in einem $if_{mod_1\circ\sqcup mod_2\circ}(m,m')$ (mit beliebigen $m,m' \in M$). Mithin kommen die abstrakten Indizes s und t im Cache-Modell $(mod_1 \circ \sqcup\, mod_2\circ)\star$ nicht mehr vor. Es bleibt als lediglich der abstrakte Index $s \bullet t$ und die Behauptung folgt. □

7 Anwendungsbeispiele

Im folgenden wird gezeigt, wie die formal behandelten Cache-Modelle in der Praxis umgesetzt werden können. Die vorgeschlagene Implementierung folgt dabei im wesentlichen dem Formalismus aus Abschnitt 4.

Der Java-Pseudocode aus Abbildung 14 stellt eine einfache Dienstschnittstelle dar, um einen Abonnement-Dienst zu implementieren. Einem Abonnent (Subscriber) kann dabei eine Liste von Abonnements etwa für Zeitschriften zugewiesen werden. Ein Abonnent wird identifiziert durch seine ID und ein Abonnement durch den entsprechenden Zeitschriftentitel. Pro Abonnent können neue Abonnements hinzugefügt werden oder existierende gelöscht werden. Die Liste der Abonnements kann für einen Abonnenten gelesen werden, und umgekehrt kann man für jede Zeitschrift auch die Liste der Abonnenten abfragen.

Cache-Modelle werden für einen entsprechenden Methoden-Cache mit Hilfe von XML-Dateien spezifiziert. Wie bei der formalen Definition können dabei abstrakte Indizes benannt werden. Indexelemente dürfen hier beliebige Java-Objekte sein – ähnlich wie dies der Formalismus in Definition 6 nahe legt. In der praktischen Umsetzung ersetzt ein Objektvergleich die

```
public interface Subscription {
  String getTitle();
  Vector getSubscriberList();
}

public interface Subscriber {
  String getID();
  Vector getSubscriptionList();
  void addSubscription(String title);
  void removeSubscription(String title);
}
```

Abbildung 14: Java-Interfaces für einen Abonnementdienst in Pseudocode

Gleichheitsbeziehung $\equiv$, die bezüglich Java über die Methode `java.lang.Object.equals()` realisiert ist. Für die Implementierung wird aber kein besonderes Indexelement erwartet, das dem Joker $\diamondsuit$ aus Definition 5 entspricht, und so handelt es sich bei den Indizies aus formaler Sicht nicht um Resultatmengen (wie in Definition 6). Stattdessen gibt es die Möglichkeit, direkt eine Indexfunktion `all` in der XML-Datei anzugeben, die der Funktion *`all`* aus Definition 9 entspricht (siehe unten).

Abbildung 15 präsentiert ein XML-basiertes Cache-Model für die Dienstschnittstelle von Abbildung 14. In der XML-Datei wird eingangs nur ein Index, nämlich `Subscriber`, über seinen Namen definiert (Zeile 2). Abbildung 16 zeigt den zugehörigen Cache-Modell-Graphen.

Jedes Java-Interface ist annotiert mit XML-Attributen und -Elementen, die den Elementen der Funktion *mod* aus Definition 6 entsprechen. Eine Java-Interface wird durch seinen Namen im XML-Element `interface` spezifiziert. Eine Interface-Methode wird ebenfalls durch ihren Namen identifiziert und mit Hilfe des XML-Attributs `access` entweder als Lese- oder als Schreibmethode deklariert.[14] Das `model`-Element bestimmt, auf welchen abstrakten Indizes die entsprechende Methode zugreift (Attribut `index`). Die Attributwerte für `ifun` legen Indexfunktionen als Java-Codefragmente fest. Eine Methode kann ein einzelnes Element eines Index oder alle Elemente lesen oder schreiben (letzteres wird markiert durch `ifun="all"`). Falls auf ein einzelnes Element zugegriffen wird, wertet der Methoden-Cache zur Laufzeit den Java-Ausdruck des `ifun`-Attrributs aus und berechnet so das entsprechende Indexelement. Der Ausdruck muss funktional sein (also frei von Seiteneffekten) und darf auf das `this`-Objekt und die Argumente sowie das Resultat der aufgerufenen Dienstmethode Bezug nehmen. Hierzu kann man das `this`-Objekt in dem Java-Ausdruck durch das Schlüsselwort `$this$` referenzieren; das Methodenresultat wird durch `$result$` und die Argumente werden durch die Parameternamen der aufgerufenen Methode angesprochen. (Die erste und die letzte Möglichkeit wurden im Beispiel angewendet.) Wie bereits erwähnt, kann es sich bei den Indexelementen in diesem Zusammenhang um beliebige Java-Objekte handeln, die aus der Auswertung des `ifun`-Ausdrucks resultieren. In Beispiel 15 spezifiziert `$this$.getID()` Elemente des `Subscriber`-Index als ID-Zeichenketten (Strings). Die angegebenen Java-Ausdrücke werden bei der Generierung der Cache-Klassen als Codefragmente in die zu erzeugenden Klassen eingebettet.

Im folgenden wird die Korrektheit des Modells aus Abbildung 15 kurz auf intuitiver Basis erklärt. Der einzige Index des Modells repräsentiert im wesentlichen die Menge der `Subscriber`-Objekte, die auf der Serverseite existieren (Zeil 2). Da diese Objekte eindeutig durch ihre ID identifiziert werden, liefert die angewendete Indexfunktion `$this$.getID()` ge-

[14] Falls die Identifikation durch den Namen nicht eindeutig ist (weil die entsprechende Methode überladen ist), kann man auch die volle Methodensignatur angeben.

```
1    <cachemodel>
2     <index name="Subscriber"/>
3
4     <interface name="Subscription">
5      <method name="getTitle" access="r"/>
6      <method name="getSubscriberList" access="r"
7              mapstrategy="collection" containedclass="Subscriber">
8       <model index="Subscriber" ifun="all"/>
9      </method>
10    </interface>
11
12    <interface name="Subscriber">
13     <method name="getID" access="r">
14      <model index="Subscriber" ifun="$this$.getID()"/>
15     </method>
16     <method name="getSubscriptionList" access="r"
17             mapstrategy="collection" containedclass="Subscription">
18      <model index="Subscriber" ifun="$this$.getID()"/>
19     </method>
20     <method name="addSubscription" access="w">
21      <model index="Subscriber" ifun="$this$.getID()"/>
22     </method>
23     <method name="removeSubscription" access="w">
24      <model index="Subscriber" ifun="$this$.getID()"/>
25     </method>
26    </interface>
27   </cachemodel>
```

Abbildung 15: Ein korrektes XML-basiertes Cache-Modell für den Abonnementdienst

rade diesen Wert. Durch das Modell sollen Veränderungen auf den Instanzen der `Subscriber`- und `Subscription`-Klasse widergespiegelt werden: Zum Beispiel drückt die Annotation für die Methode `addSubscription()` aus, dass das Hinzufügen eines Abonnements durch `addSubscription()` das entsprechende Abonnentenobjekt (vom Typ `Subscriber`) verändert (Zeilen 20, 21). Das Resultat der Methode `getSubscriberList()` ändert sich, wenn ein entsprechendes `Subscription`-Objekt zu einem Abonnenenten (`Subscriber`) durch `addSubscription()` hinzugefügt wird oder durch `removeSubscription()` entfernt wird. Da dieser Fall für jeden Abonnenten eintreten kann, wird `getSubscriberList()` in Zeile 6 invalidiert, sobald der Klient auf einem Abonnenten mittels `addSubscription()` bzw. `removeSubscription()` „schreibt" (Zeilen 20 and 23). Man beachte, dass jedes Listenobjekt für Abonnements als Komponente eines `Subscriber`-Objekts aufzufassen ist – somit bildet der Zustand des entsprechenden Listenobjekts einen Teil des Zustands des Abonnenten.[15]

In Abbildung 17 ist das Verhalten eines entsprechenden Methoden-Caches für eine Folge von Dienstmethodenaufrufen illustriert. Im Beispiel gibt es zwei Abonnenten für Zeitschriften: Joe und Dan. Joe hat entsprechend Abbildung 17 a) die Zeitschrift „Mickey Mouse" und „Superman" abonniert, während Dan nur die Zeitschrift „Superman" bestellt hat. Die Beziehungen sind mit Hilfe der durchgezogenen Pfeile bezüglich den Personen- und Dokumentsymbolen angedeutet.

Im ersten Schritt (Abbildung 17 a)) führt der Klient des Abonnementdienstes den Methodenaufruf `mouse.getSubscriberList()` aus. `mouse` sei dabei ein Java-Objekt, welches das Java-

[15] Die Bedeutung der Attribute `mapstrategy` and `containedclass` wird hier aus Platzgründen nicht ausgeführt.

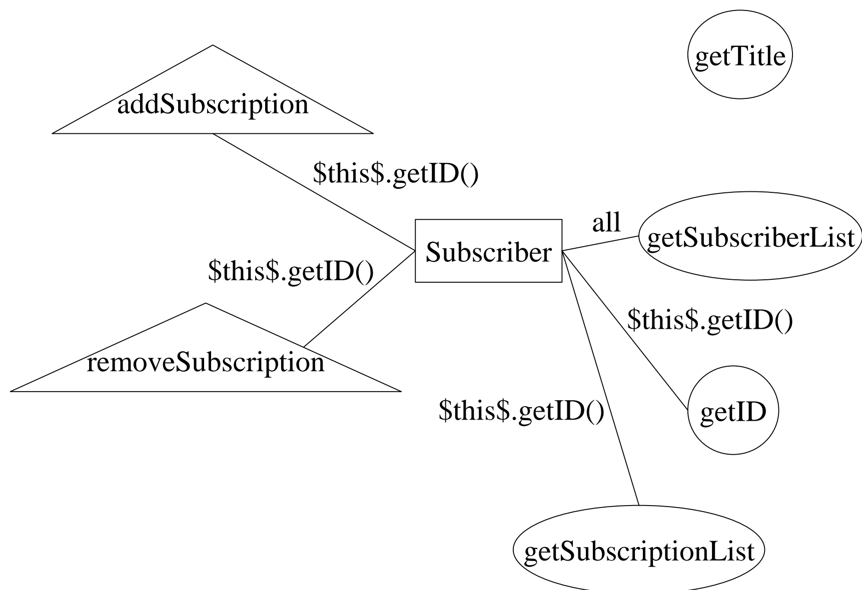


Abbildung 16: Der Cache-Modell-Graph zum Cache-Modell aus Abbildung 15

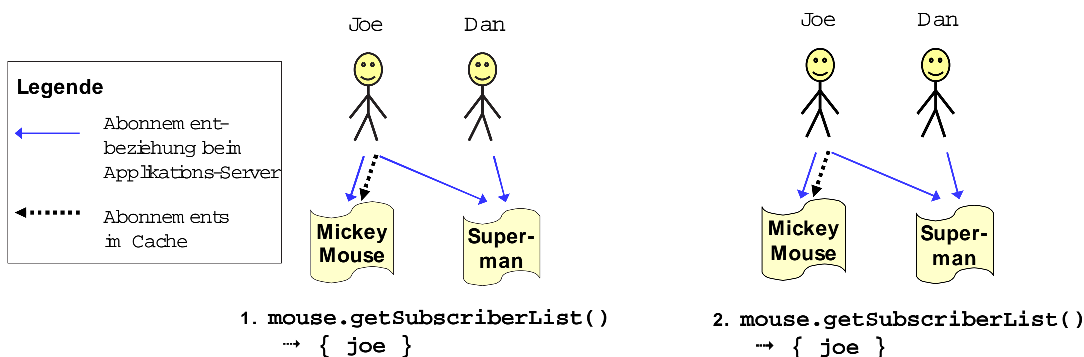
Interface `Subscription` aus Abbildung 14 implementiert und das Abonnement für die Zeitschrift „Mickey Mouse“ repräsentiert. Auf Grund der Ausgangsituation von Abbildung 17 a) liefert der Aufruf einen Java-Vektor, der als einziges Element ein Objekt enthält, das Joe als Abonnenten repräsentiert (hier angedeutet durch {`joe`}). Da `getSubscriberList()` eine Modell-Lesemethode ist, wird das gelieferte Methodenresultat im Cache vorgehalten; dies ist durch den gestrichelten Pfeil in der Abbildung angedeutet.

Der danach folgende, nochmalige Aufruf von `mouse.getSubscriberList()` kann dann aus dem Methode-Cache „bedient“ werden und führt nicht zu einem entfernten Methodenaufruf beim Applikationsserver (Abbildung 17 b)).

Nun folgt beim Klienten der Aufruf `dan.addSubscription(mouse)`, wobei `dan` ein `Subscriber`-Objekt bezeichnet, welches dem Abonnenten Dan entspricht. Damit verändert sich natürlich die Menge der Abonnenten von „Mickey Mouse“, und somit sollte das gecachte Methodenresultat für den Aufruf `mouse.getSubscriberList()` invalidiert werden. Genau dies erzwingt das oben diskutierte Cache-Modell (aus Abbildung 15), denn der Methodenaufruf aus Abbildung 17 a) las alle Elemente des Index `Subscriber` (durch die Funktion `all` in Zeile 8 des Cache-Modells), während der Aufruf `dan.addSubscription(mouse)` ein Indexelement von `Subscriber` schreibt (Zeile 21 von Abbildung 15). Damit stellt der Cache eine Modellabhängigkeit der beiden Aufrufe fest und entfernt das entsprechende Methodenresultat {`joe`} aus seinem Speicher. Die Veränderungen sind in Abbildung 17 c) durch den fehlenden gestrichelten Pfeil und die zusätzliche Kante von Dan nach „Superman“ angedeutet.

Ein weiterer Aufruf von `mouse.getSubscriberList()` (Abbildung 17 d)) erreicht nun also wieder den Applikationsserver und liefert das Resultat {`joe,dan`}. Das neue Resultat des Methodenaufrufs wird abermals gecacht, was durch die gestrichelten Pfeile in Abbildung 17 d) angedeutet ist.

Abbildung 18 zeigt ein weiteres korrektes Cache-Model für den Abonnement-Dienst mit dem zugehörigen Cache-Modell-Graphen in Abbildung 19.



a) Das Resultat des Methodenaufrufs wird auf dem Server berechnet und gecacht.

b) Das Resultat des wiederholten Methodenaufrufs kann aus dem Cache geholt werden.

c) Der Aufruf der Schreibmethode invalidiert das zuvor gecachte Resultat.

d) Das veränderte Resultat des Methodenaufrufs wird wiederum auf dem Server berechnet und gecacht.

Abbildung 17: Beispiel für eine Sequenz von Methodenaufrufen zur Dientschnittstelle aus Abbildung 14. Beziehungen zwischen Abonnenten (`Subscriber`) und Abonnements (`Subscription`) sind symbolisch dargestellt.

Im Gegensatz zu dem Modell aus Abbildung 15 besitzt es zwei Indizes, die mit `Subscription2` und `Subscriber2` benannt sind. Der Index `Subscriber2` erfüllt hier eine ähnliche Aufgabe wie der Index `Subscriber` aus Abbildung 15. Der zusätzliche Index `Subscription2` dient dazu, die Menge der `Subscription`-Objekte (auf der Serverseite) über ihren Titel zu repräsentieren. Aus diesem Grund liefert die entsprechende Indexfunktion `$this$.getTitle()` gerade den Titel eines Abonnements. Das Cache-Modell aus Abbildung 18 bildet die Abhängigkeiten zwischen der Methode `getSubscriberList()` und den Methoden `addSubscription()` bzw. `removeSubscription()` nun über den Index `Subscription2` ab, was wesentlich präziser ist als die Alternative aus Abbildung 15. Denn während ein Aufruf von `addSubscription()` (oder `removeSubscription()` gemäß dem Cache-Modell aus Abbildung 15 immer *alle* gecachten Methodenresultate zur Methode `getSubscriberList()` invalidiert, geschieht dies im Modell von Abbildung 18 nur für ein einziges Methodenresultat zu `getSubscriberList()`. Es wird dann genau jenes Methodenresultat invalidiert, welches sich auf das `Subscription`-Objekt bezieht,

```
1   <cachemodel>
2    <index name="Subscription2"/>
3    <index name="Subscriber2"/>
4
5    <interface name="Subscription">
6     <method name="getTitle" access="r">
7      <model index="Subscription2" ifun="$this$.getTitle()"/>
8     </method>
9     <method name="getSubscriberList" access="r"
10            mapstrategy="collection" containedclass="Subscriber">
11     <model index="Subscription2" ifun="$this$.getTitle()"/>
12    </method>
13   </interface>
14
15   <interface name="Subscriber">
16    <method name="getID" access="w"/>
17    <method name="getSubscriptionList" access="r"
18            mapstrategy="collection" containedclass="Subscription">
19     <model index="Subscriber2" ifun="$this$.getID()"/>
20    </method>
21    <method name="addSubscription" access="w">
22     <param name="subscription"/>
23     <model index="Subscriber2" ifun="$this$.getID()"/>
24     <model index="Subscription2" ifun="$subscription$.getTitle()"/>
25    </method>
26    <method name="removeSubscription" access="w">
27     <param name="subscription"/>
28     <model index="Subscriber2" ifun="$this$.getID()"/>
29     <model index="Subscription2" ifun="$subscription$.getTitle()"/>
30    </method>
31   </interface>
32  </cachemodel>
```

Abbildung 18: Ein weiteres korrektes Cache-Modell für den Abonnementdienst

das in dem betreffenden `addSubscription()`- bzw. `removeSubscription()`-Aufruf als Argument übergeben wurde.

Das Cache-Modell aus Abbildung 18 ist aber insgesamt weniger präzise als jenes aus Abbildung 15, da es die Methoden `getID()` in Zeile 16 (unnötiger Weise) als Modell-Schreibmethode deklariert.

Abbildung 20 zeigt die bereinigte verfeinerte Vereinigung der Cache-Modelle aus den Abbildungen 18 und 15. (Das Modell aus Abbildung 18 diente als erster Operand des Vereinigungsoperators.) Wie man sieht, verbindet das entstehende Modell die Vorteile der Ausgangsmodelle: die Methoden `getTitle()` und `getID()` sind als nicht invalidierbare Modell-Lesemethoden deklariert, und die Abhängigkeit zwischen `getSubscriberList()` und `addSubscription()` bzw. `removeSubscription()` wird so behandelt wie in Abbildung 18. Hätte man die verfeinerte Vereinigung in der Form „$15 \uplus 18$“ vorgenommen, wäre der zweite Vorteil leider nicht eingetreten. Der Cache-Modell-Graph zu „$15 \uplus 18$“ ist in Abbildung 21 dargestellt.

Zum Abschluss wird in Abbildung 22 noch das (bereinigte) Ergebnis der Produktvereinigung „$15 \sqcup 18$“ vorgestellt, wie es bezüglich der Java-Implementierung des Cache-Systems zu Stande kommt. Den zugehörigen Cache-Modell-Graphen illustriert Abbildung 23.

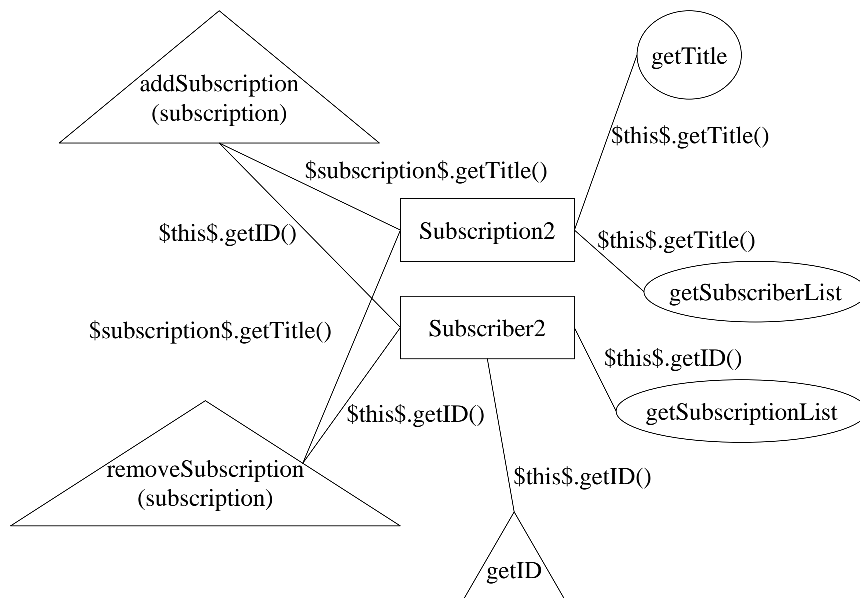


Abbildung 19: Der Cache-Modell-Graph zum Cache-Modell aus Abbildung 18

Das Ergebnis der Produktvereinigung ist genauso präzise wir das Cache-Modell aus Abbildung 20, aber, wie bereits in Abschnitt 5 angesprochen, wirkt es durch die Produktbildung wesentlich komplexer. Der große Vorteil ist hier, dass die Reihenfolge der Operandenmodelle auf die Präzision keine Auswirkung hat. Die Produktfunktionen bzw. die Elemente der Produktindizes werden durch die Erzeugung von Tupelobjekten der Klasse `Pair` hergestellt. Die Klasse `Pair` ist mit geeigneten Eigenschaften für die Vergleichsmethode `Pair.equals()` als gegeben vorausgesetzt. Durch die Konkatenation der ursprünglichen Indizes entstehen die neuen Indizes `Subscriber_Subscription2` und `Subscriber_Subscription2`. Hier gibt es noch automatisierbare Optimierungsspielräume bei der Berechnung der Produktvereinigung, denn offensichtlich könnte man statt `Subscriber_Subscriber2` wieder einen verkürzten Index `Subscriber` einführen und die Zeilen 18, 23 sowie 30 dann durch `<model index="Subscriber" ifun="$this$.getID()"/>` ersetzen. Ein entsprechender Optimierer könnte dies leicht über die Gleichheit der entsprechenden Indexfunktionen in den Operandenmodellen erkennen.

8 Implementierungs- und Anwendungsaspekte

In diesem Abschnitt wird eine Liste noch ausstehender, praktischer Probleme abgehandelt, die beim Einsatz eines Methoden-Caches bzw. von Cache-Modellen auftreten.

8.1 Korrektheitstests für Cache-Modelle

Ein wichtiges Problem ist die Zusicherung der Korrektheit eines Cache-Modells und damit die Konsistenz des Methoden-Caches zur Laufzeit. Um die Korrektheit zu gewährleisten, bietet der implementierte Methoden-Cache-Prototyp einen Korrektheitsprüfmodus an, in dem der Cache

```
1   <cachemodel>
2    <index name="Subscription2"/>
3    <index name="Subscriber2"/>
4
5    <interface name="Subscription">
6     <method name="getTitle" access="r"/>
7     <method name="getSubscriberList" access="r"
8            mapstrategy="collection" containedclass="Subscriber">
9      <model index="Subscription2" ifun="$this$.getTitle()"/>
10    </method>
11   </interface>
12
13   <interface name="Subscriber">
14    <method name="getID" access="r"/>
15    <method name="getSubscriptionList" access="r"
16           mapstrategy="collection" containedclass="Subscription">
17     <model index="Subscriber2" ifun="$this$.getID()"/>
18    </method>
19    <method name="addSubscription" access="w">
20     <param name="subscription"/>
21     <model index="Subscriber2" ifun="$this$.getID()"/>
22     <model index="Subscription2" ifun="$subscription$.getTitle()"/>
23    </method>
24    <method name="removeSubscription" access="w">
25     <param name="subscription"/>
26     <model index="Subscriber2" ifun="$this$.getID()"/>
27     <model index="Subscription2" ifun="$subscription$.getTitle()"/>
28    </method>
29   </interface>
30  </cachemodel>
```

Abbildung 20: Bereinigte verfeinerte Vereinigung der Cache-Modelle aus Abbildung 18 und 15 ($(18 \uplus 15)^\star$)

wie gewohnt arbeitet, aber im Falle eines Cache-Treffers den Aufruf der entsprechenden Lesemethode *auch* zum Applikationsserver weiterleitet. Danach werden das Resultat, das vom Applikationsserver stammt und das gecachte Pendant verglichen. Wenn Sie sich unterscheiden, ist die Cache-Konsistenz verletzt; somit muss das Cache-Modell inkorrekt sein und angepasst werden.

Die Bestimmung der Gleichheit eines gecachten Methodenresultats und dem entsprechenden Ergebnis des Applikationsservers ist bei komplexen, objektwertigen Resultaten nicht trivial. Standardmäßig verwendet der Korrektheitsprüfmodus des Prototyps hierfür die Vergleichsmethode `java.lang.Object.equals()`, deren Implementierung in diesem Zusammenhang aber nicht immer geeignet ist. Deshalb kann man in einem Cache-Modell pro annotierter Methode mit dem XML-Attribut `compare` eine eigene Vergleichsmethode angeben.

Wie bei gewöhnlichen Testverfahren für Programme lässt die Abwesenheit entsprechender Inkonsistenten leider keine direkten (logischen) Schlüsse zu. Man hat dann aber ein höheres Zutrauen in die Korrektheit des erstellten Cache-Modells.

Eine andere eher theoretische Option wäre es, die Korrekheit des Cache-Modells *zu beweisen*. Im Allgemeinen ist diese Aufgabe aber extrem schwierig und nicht automatisierbar auf Grund der Ergebnisse der Berechenbarkeitstheorie zu verwandten Problemen (wie etwa dem Halteproblem). Ein typischer Anwendungsprogrammierer wäre mit der Beweisaufgabe überfordert. Man

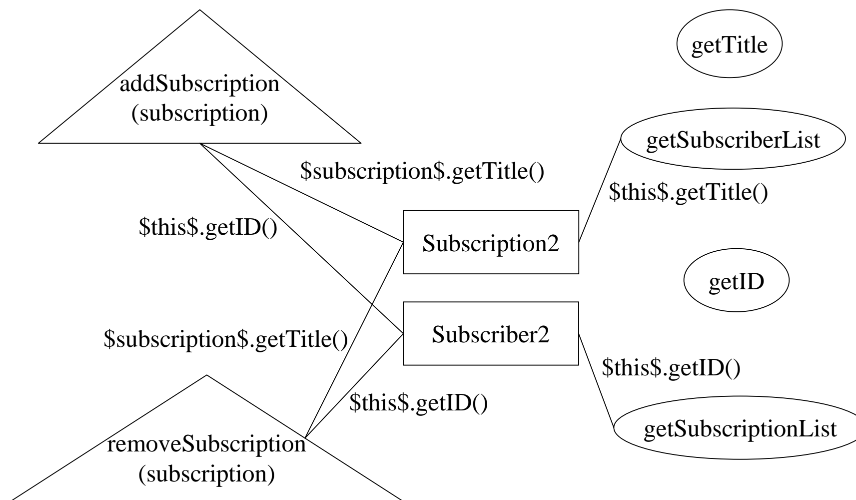


Abbildung 21: Der Cache-Modell-Graph zum Cache-Modell aus Abbildung 20

müsste hierzu evtl. den *gesamten* Quellcode analysieren, der über die Dienstschnittstellenaufrufe potentiell ausgeführt wird (inklusive der Implementierung des Applikationsserver-Systems selbst).

Wenn der Methoden-Cache zur Performanzverbesserung eines existierenden Systems eingesetzt wird, gibt es häufig schon umfangreiche Blackbox-Tests und Belastungstests (Load Tests) bezüglich der Dienstschnittstelle oder der Präsentationsschicht des Systems. Diese Tests können dann eventuell ohne oder mit nur wenigen Änderungen übernommen werden, um eine Korrektheitsprüfung, wie oben beschrieben, durchzuführen. Gegenüber normalen Programmtests, für die meist besondere Testtreiber entwickelt werden müssen, ist die vorgeschlagenen Korrektheitsprüfung ebenso wie der Methoden-Cache selbst transparent, was die Wiederverwendung existierender Testprogramme ermöglicht. Dieser Ansatz zum Korrektheitstest eines Cache-Modells wurde übrigens für die Experimente aus [7] erfolgreich angewendet. Da auch dort bereits Systembelastungstests existierten, musste kein zusätzlicher Aufwand zur Durchführung der Korrektheitsprüfung getrieben werden.

8.2 Cache-Kohärenz bei *n* Klienten

Bei der Anwendung von korrekten Cache-Modellen zum konsistenten Methoden-Caching wurde bisher unterstellt, dass alle Aufrufe von Dienstmethoden zuerst den Methoden-Cache passieren. Denn nur dadurch können alle Schreibmethodenaufrufe, die gecachte Methodenresultate invalidieren sollen, erfasst werden. Diese Annahme ist erfüllt, wenn es genau einen Klienten gibt, der überhaupt Dienstmethoden bei einem Server aufruft. In diesem Fall kann also der Methoden-Cache, wie bereits in Abbildung 2 angedeutet, direkt beim Klienten installiert sein.

Falls, wie häufig üblich, eine unbekannte Zahl von Klienten existiert und wenn man nicht sicher sein kann, dass ein Klient den Methoden-Cache lokal verwendet, empfiehlt es sich, den Cache gleich komplett auf der Applikationsserver-Seite zu installieren. Dadurch ergibt sich ein Systemaufbau entsprechend Abbildung 24. Natürlich stellen sich dann in Bezug auf die Klienten keine Cache-Kohärenzprobleme, da nur eine Instanz des Methoden-Caches existiert, welche beim (einzigen) Server installiert ist. Der Methoden-Cache führt dann bei Cache-Treffern vor-

```
1   <cachemodel>
2    <index name="Subscriber_Subscription2"/>
3    <index name="Subscriber_Subscriber2"/>
4
5    <interface name="Subscription">
6     <method name="getTitle" access="r"/>
7     <method name="getSubscriberList" access="r"
8             mapstrategy="collection" containedclass="Subscriber">
9      <model index="Subscriber_Subscription2"
10            ifun="new Pair($all$, $this$.getTitle())"/>
11    </method>
12   </interface>
13
14   <interface name="Subscriber">
15    <method name="getID" access="r"/>
16    <method name="getSubscriptionList" access="r"
17            mapstrategy="collection" containedclass="Subscription">
18     <model index="Subscriber_Subscriber2"
19            ifun="new Pair($this$.getID(), $this$.getID())"/>
20    </method>
21    <method name="addSubscription" access="w">
22     <param name="subscription"/>
23     <model index="Subscriber_Subscriber2"
24            ifun="new Pair($this$.getID(), $this$.getID())"/>
25     <model index="Subscriber_Subscription2"
26            ifun="new Pair(this$.getID(), $subscription$.getTitle())"/>
27    </method>
28    <method name="removeSubscription" access="w">
29     <param name="subscription"/>
30     <model index="Subscriber_Subscriber2"
31            ifun="new Pair($this$.getID(), $this$.getID())"/>
32     <model index="Subscriber_Subscription2"
33            ifun="new Pair(this$.getID(), $subscription$.getTitle())"/>
34    </method>
35   </interface>
36  </cachemodel>
```

Abbildung 22: Bereinigte Produktvereinigung der Cache-Modelle aus Abbildung 15 und 18 ($(15 \sqcup 18)\star$)

aussichtlich immer noch zu großen Ressourceneinsparungen, da der interne Applikationsserver-Zugriff und auch ein dadurch ausgelöster Datenbankzugriff vermieden werden. Die Kosten von entfernten Methodenaurufen zwischen Klient und Applikationsserver bleiben aber bestehen.

Eine anderer häufiger Fall tritt ein, wenn jeder einzelne Klient mit einem Methoden-Cache ausgerüstet werden kann. Ein Beispiel hierfür ist ein Cluster von Servlet-fähigen Webservern, die alle auf einen Applikationsserver zugreifen. Alternativ können sogar mehrere geclusterte Applikationsserver existieren, auf die die Webserver zugreifen.

In dieser Situation muss die Kohärenz der verschiedenen Methoden-Caches sichergestellt werden, so dass diese also *zueinander* konsistent sind. Dazu bietet sich eine Architektur an, wie sie in Abbildung 25 dargestellt ist: Neben den mit Methoden-Caches ausgerüsteten Klienten und dem (potentiellen) Cluster von Applikationsservern existiert noch ein so genannter *Invali-*

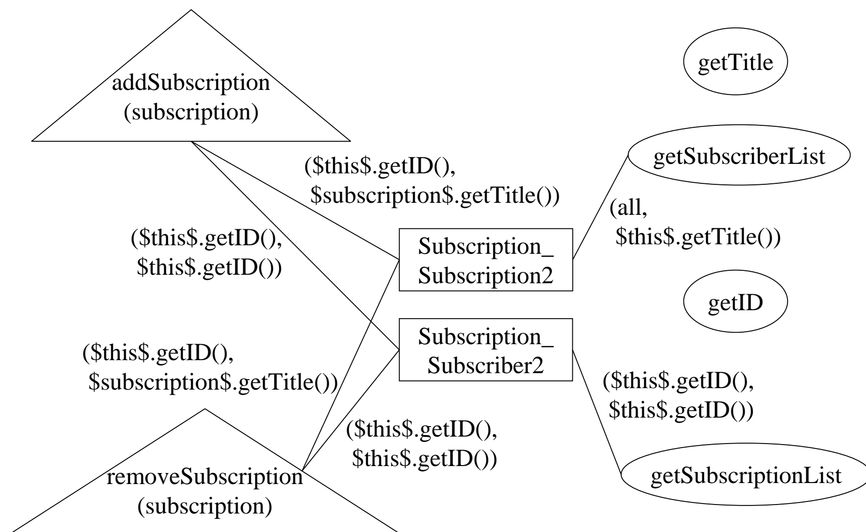


Abbildung 23: Der Cache-Modell-Graph zum Cache-Modell aus Abbildung 22

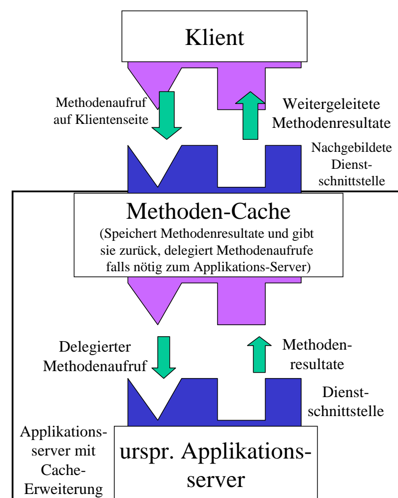


Abbildung 24: Interaktion von Applikationsserver und Klient bei Einführung eines Methoden-Caches auf der Serverseite

dierungsserver, an den *Invalidierungsnachrichten* von den Methoden-Caches aus gesendet werden. Die Invalidierungsnachrichten sind dabei sehr einfach strukturierte Nachrichten in der Tupelform $(Index, Indexelement)$. Ein Methoden-Cache gibt ein solches Tupel an den Invalidierungsserver immer dann weiter, wenn über den Cache eine Schreibmethode ausgeführt wurde, die auf dem entsprechenden Index bzw. Indexelement des Cache-Modells schreibt. Man beachte, dass hierbei alle Klienten das gleiche Cache-Modell verwenden, da sie auch auf der gleichen Dienstschnittstelle operieren.

Der Invalidierungsserver leitet seinerseits das empfangene Tupel an alle (ihm bekannten) Kli-

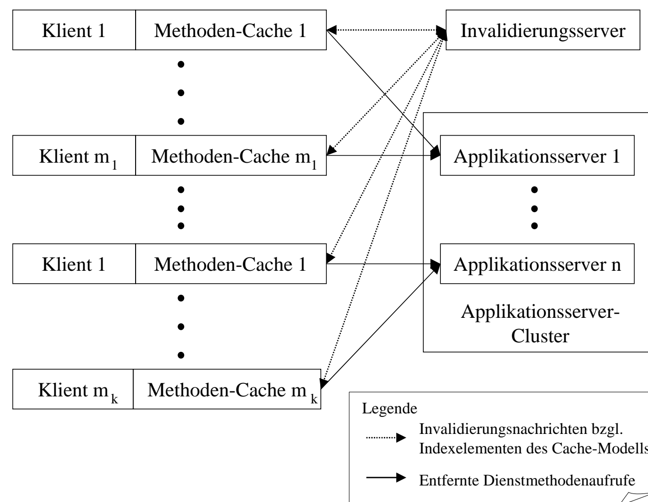


Abbildung 25: Architektur zur Kohärenz von Klient-seitigen Methoden-Caches durch Versenden von Invalidierungsnachrichten über einen Invalidierungsserver

enten als Nachricht weiter außer an den Klienten, von dem das Tupel ursprünglich stammt. Dies kann zum Beispiel als Broadcast im lokalen Netz des Systems geschehen; alternativ kann der Invalidierungsserver aber auch die Klienten spezifisch addressieren, sofern sie sich zuvor bei ihm registriert haben. Sobald ein Methoden-Cache ein entsprechendes Tupel vom Invalidierungsserver empfängt, nutzt der Cache diese Information, um Invalidierungen auf lokal gecachten Methodenresultaten durchzuführen. Dies geschieht so, als ob der ursprüngliche Schreibmethodenaufruf, durch den die Invalidierungsnachricht entstand, bei dem entsprechenden Methoden-Cache lokal stattgefunden hätte.

Durch die Architektur ist sichergestellt, dass stets alle Methoden-Caches alle Invalidierungsnachrichten erhalten, sofern es keine Netzwerkausfälle gibt. Somit ist Cache-Kohärenz gewährleistet; lediglich durch die geringe Latenzzeit zwischen dem Versand einer Invalidierungsnachricht und der Verarbeitung bei einem empfangenden Methoden-Cache kann es vorkommen, dass ein Cache bereits ungültig gewordene Methodenresultate an den Klientencode weitergibt. Entsprechende Verzögerungen sind aber extrem kurz, und daher ist das Problem in der Praxis meist vernachlässigbar.

Die Verwendung eines Invalidierungsservers orientiert sich an einem Architekturmodell aus [1], bei dem ein so genannter globaler Cache-Manager (Global Cache Manager, GCM) für ähnliche Aufgaben in allgemeineren Caching-Szenarien vorgesehen ist. Verwendet man, wie im vorigen Abschnitt bereits angedeutet, einen Broadcasting-Ansatz zur Übermittlung von Invalidierungsnachrichten, kann der Invalidierungsserver auch komplett aus der Architektur entfernt werden. Voraussetzung ist dann, dass ein Broadcast auf Basis der Netzwerktopologie alle Klienten (bzw. deren Methoden-Caches) erreicht und das Netzwerk nicht in kritischer Weise belastet.

8.3 Umgehung des Methoden-Caches

Unter einer *Umgehung* des Methoden-Caches wird in diesem Bericht ein Ereignis verstanden, das den Zustand des Applikationsservers verändert, ohne dass der Methoden-Cache darüber benachrichtigt wird. Im vorigen Abschnitt wurde bereits der Fall abgehandelt, dass ein Klient

die Dienstschnittstelle des Applikationsservers aufruft, ohne dass dieser Aufruf den Methoden-Cache passiert. Das Problem taucht aber auch auf, wenn der Zustand des Applikationsservers oder eines seiner Subsysteme nicht über die dem Klienten bekannte Dienstschnittstelle, sondern über eine andere Systemschnittstelle verändert wird. Wenn der Applikationsserver etwa einen Teil seines Zustands in einer relationalen Datenbank speichert, dann könnte es noch andere Prozesse geben, die auf der Datenbank arbeiten und somit den Applikationsserver indirekt beeinflussen. Dies kann, wie bereits besprochen, zu Inkonsistenzen bezüglich gecachter Methodenresultate führen.

Der wichtigste Ansatz, um dieses Problem zu behandeln, ist die sogenannte *programmierte Invalidierung*. Im Falle des Methoden-Cachings sollte hierzu eine Programmierschnittstelle existieren, über die der Methoden-Cache direkt über die entsprechende Zustandsveränderung benachrichtigt werden kann. Damit dies funktioniert, muss aber der Programmcode des Systemteils, in dem die Zustandsveränderung stattfindet, entsprechend angepasst werden. Für das Datenbankbeispiel von oben könnten etwa Datenbanktrigger bei einem Schreibzugriff auf relevante Tabellendaten die Invalidierungsschnittstelle des Methoden-Caches aufrufen.

Die Struktur der Invidierungschnittstelle ist dabei sehr einfach: ähnlich wie in Abschnitt 8.2 müssen die Aufrufer hierzu Invalidierungstupel der Form $(Index, Indexelement)$ an eine entsprechende Methode übergeben. Es ist zu beachten, dass der Aufrufer hierbei also die lokale Zustandsveränderung selbständig auf die Indexstruktur des Cache-Modells abbilden muss.

Zur Vermeidung aufwändiger Systemanpassungen, wie sie bei der programmierten Invalidierung notwendig werden, lassen sich natürlich auch zeitgesteuerte Invalidierungsansätze für das Methoden-Caching verwirklichen. Gecachte Methodenresultate werden nach einer gewissen Zeit automatisch aus dem Cache gelöscht. Dadurch können allerdings veraltetete, also inkonsistente Resultate an die Klienten der Dientschnittstelle weitergegeben werden.

Häufig ist es sinnvoll, eine gemischte Invalidierungstrategie anzuwenden, das heißt ein Teil der Methoden der Dienstschnittstelle liefert Resultate, für die man starke Konsistenz erwartet, für einen anderen Teil genügen aber schwächere Konsistenzgarantien. Dem entsprechend setzt man eine Invalidierungsstrategie mittels Cache-Modellen ein, um die volle Konsistenz dort zu erreichen, wo sie gefordert ist, und wendet für die übrigen Dienstmethoden etwa eine zeitgesteuerte Invalidierung an.

Darüber hinaus gibt es Lesemethoden, für die das Caching von Resultaten nicht anwendbar ist. Ein einfaches Beispiel ist eine Methode, die die aktuelle Uhrzeit liefert. Für solche Methoden sollte das Caching von Methodenresultaten komplett abgeschaltet werden.

Literatur

[1] P. Bodorik and D. Jutla. Qos architecture for caching in middleware. In *International Conference on Advanced Infrastructures in e-Business, Education, Science, Medicine, Mobile Technologies, on the Internet*, Aquila, Italy, 2003.

[2] S. J. Caughey, G. D. Parrington, and S. K. Shrivastava. Shadows - A flexible support system for objects in distributed systems. In *Proc. Third Int. Workshop on Object Orientation and Operating Systems*, pages 73–82, Asheville, NC (USA), Dec. 1993.

[3] G. V. Chockler, D. Dolev, R. Friedman, and R. Vitenberg. Implementing a caching service for distributed cobra objects. In *IFIP/ACM International Conference on Distributed systems platforms*, pages 1–23. Springer-Verlag New York, Inc., 2000.

[4] Dynamic proxy classes. `http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html`.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[6] P. Martin, V. Callaghan, and A. Clark. High performance distributed objects using caching proxies for large scale applications. In *International Symposium on Distributed Objects and Applications*, pages 110–119, 1999.

[7] D. Pfeifer and H. Jakschitsch. Method-based caching in multi-tiered server applications. Technical Report 2003-11, Universität Karlsruhe, 2003.

[8] D. Pfeifer and H. Jakschitsch. Method-based caching in multi-tiered server applications. In *Proceedings of OTM Conferérated International Conferences CoopIS, DOA, and ODBASE 2003*, Sicily, Italy, November 2003. Springer.

[9] Z. Tari, H. Hamidjaja, and Q. T. Lin. Cache management in CORBA distributed object systems. *IEEE Concurrency*, 8(3):48–55, July/Sept. 2000.

[10] M. van Steen, P. Homburg, and A. S. Tanenbaum. Globe: A wide-area distributed system. *IEEE Concurrency*, 7(1):70–78, Jan.-Mar. 1999. `http://www.cs.vu.nl/pub/papers/globe/ieeeconc.99.org.ps.Z`.