Design und Implementierung eines Ontologie-Agenten auf Basis von Protégé und JADE

Reinhard Vicinus

27. März 2002

Betreuer:
Dipl.-Inform. Ludger van Elst
Dipl.-Inform. Michael Sintek

Verantwortlicher Professor:
Prof. Dr. Andreas Dengel

Zusammenfassung

Ontologie-Agenten stellen anderen Agenten Funktionen zur Verfügung, mit denen Ontologien geändert, erweitert und befragt werden können. Der hier vorgestellte Ontologie-Agent wurde in JADE, einer FIPA-konformen Agentenplattform implementiert und verwendet als Backend Protégé zum Verwalten der Ontologien. Hierfür wurde Protégé um eine Schnittstelle erweitert, über die der Ontologie-Agent auf die Funktionalität von Protégé zugreifen kann. Die Sprechakte dieser Schnittstelle stellt der Ontologie-Agent anderen Agenten zur Verfügung. Die Sprechakte, die diese Schnittstelle verwendet, sind durch ein RDF-Schema definiert.

Nach einer Einführung wird anhand eines Beispiels der Aufbau der Sprechakte dargestellt und erklärt, wie sie zu verwenden sind. Weiterhin werden die Implementationen des Ontologie-Agenten in JADE und der Schnittstelle in Protégé vorgestellt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Im Projekt FRODO[2][1] werden Methoden und Werkzeuge zur Konstruktion und Pflege von verteilten Unternehmensgedächtnissen (Organizational Memories[3]) entwickelt. Insbesondere wird ein agentenbasiertes Framework für den Aufbau solcher verteilten Organizational Memories zur Verfügung gestellt. Eine prototypische Implementierung erfolgt auf der Basis der FIPA-konformen Agentenplattform JADE[5].

Eine wichtige Form der Wissensrepräsentation in Organizational Memories sind Ontologien[6]. Eine Ontologie ist eine formale Spezifikation einer Konzeptualisierung eines Bereiches[7]. Allgemein ausgedrückt bilden Ontologien die Begriffswelt einer Domäne auf ein Modell ab, dass von den verschiedenen, menschlichen oder maschinellen Akteuren einer Informationslandschaft geteilt, d.h. gleich verstanden wird. In Organizational Memories werden Ontologien dazu verwendet, Wissensquellen im Kontext der Domäne zu betrachten, um das Wissen besser analysieren und verstehen zu können. Typische Anwendungsgebiete sind Diskussionsgruppen, Informationsfilterung, Suche nach nicht textbasierten Informationen und Expert-User Kommunikation[11]. Insbesondere werden Ontologien dazu genutzt, um die in diesen Informationsquellen enthaltenen Informationen genauer beschreiben und damit verstehen zu können, welche Informationen in welchen Quellen liegen und ob die Informationen in einem bestimmten Kontext relevant sind. Aufbau und Wartung von Ontologien sind sehr aufwendig. Der Aufwand nimmt außerdem stark zu, je umfangreicher Ontologien sind. Auch sind der Aufbau und die Pflege mit erheblichen Kosten verbunden, da viele der Aufgaben nicht automatisierbar sind. Deshalb werden Ontologien meistens nur für spezielle Aufgaben eingesetzt, wo der Kosten-Nutzen-Faktor hoch ist[3]. Darüberhinaus wird in FRODO die Annahme von *zentralen* Ontologien fallengelassen, so dass der Wartungsprozess für verteilte Ontologien geeignet sein muss. Diese verteilten Ontologien sollen mit Hilfe von Ontologie-Agenten in Organizational Memories genutzt werden. Das Projekt FRODO verwendet ein framebasiertes Wissensrepräsentationssystem für Ontologien, nämlich Protégé[10].

Ziel dieser Arbeit ist es, JADE-Agenten verfügbar zu machen, die Ontologien mittels Protégé verwalten können, d.h. Sprechakte für Aufbau und Wartung und zur Abfrage von Ontologien zur Verfügung zu stellen. Diese JADE-Agenten werden im folgenden Jade Protégé Agenten (JPA) genannt. Die generelle Architektur der Jade Protégé Agenten ist die folgende: Die Jade Protégé Agenten bestehen aus zwei Teilen, einem JADE-Agenten mit dem Namen *Protégé Agent* und

einem Plugin für Protégé, das *RDF-Tab* bezeichnet wird. Der Protégé Agent ist für die Kommunikation mit den anderen JADE-Agenten im FRODO-Framework zuständig, er leitet Anfragen anderer JADE-Agenten an das RDF-Tab weiter und sendet die Antworten des RDF-Tab an die entsprechenden JADE-Agenten zurück. Das RDF-Tab ist für die Verarbeitung der Anfragen zuständig.

In den folgenden Kapiteln wird ein Überblick über die Jade Protégé Agenten gegeben. Kapitel 2 zeigt anhand eines kleinen Beispiels die grundlegende Funktionsweise der Jade Protégé Agenten. In Kapitel 3 werden die Sprechakte genauer betrachtet, und in Kapitel 4 wird ein grober Überblick über die Implementierung des Protégé Agenten und des RDF-Tab gegeben. Kapitel 5 gibt einen Ausblick auf zukünftig angedachte Erweiterungen der Jade Protégé Agenten. Eine genaue Beschreibung aller Sprechakte ist im Anhang zu finden.

Kapitel 2

Einführendes Beispiel

Das Ziel der Jade Protégé Agenten ist es, anderen JADE-Agenten Mittel zum Aufbau und Wartung und zur Abfrage von Ontologien zur Verfügung zu stellen. In diesem Kapitel wird eine einfache Ontologie beschrieben, mit deren Hilfe im weiteren Verlauf die Funktionsweise der Jade Protégé Agenten erläutert werden soll.

Wie schon in der Einleitung erwähnt, wird Protégé[10] zur Verwaltung der Ontologien eingesetzt. Protégé verwendet ein framebasiertes Modell zur Repräsentation von Ontologien, d.h Frames sind die Hauptbausteine des Modells. Eine Protégé Ontologie besteht aus Klassen, Slots, Facetten und Axiomen, wobei Klassen Begriffe aus der betrachteten Domäne sind. Slots Eigenschaften oder Merkmale von Klassen beschreiben, Facetten Eigenschaften von Slots beschreiben und Axiome zusätzliche Nebenbedingungen spezifizieren. Das Protégé Modell kann zusätzlich zur eigentlichen Ontologie noch Instanzen von Klassen mit Werten in den Slots verwalten.

Die Beispielontologie *Fahrzeuge* ist in Abbildung 2.1 abgebildet. Die Ontologie besteht nur aus den Klassen *Kraftfahrzeug*, *PKW*, *LKW* und *Sportwagen* sowie den Slots *bezeichnung*, *hoechstgeschwindigkeit* und *sitzplaetze*. Die oberste Klasse ist *Kraftfahrzeug*, an ihr hängen die beiden Slots *bezeichnung* und *hoechstgeschwindigkeit*. Die Klasse *Kraftfahrzeug* hat zwei direkte Unterklassen, *PKW* und *LKW*. Die Klassen *PKW* und *LKW* erben die Slots ihrer Oberklasse *Kraftfahrzeug*. Zusätzlich besitzt die Klasse *PKW* noch den Slot *sitzplaetze* und hat noch eine direkte Unterklasse, nämlich *Sportwagen*.

Diese kleine Ontologie soll nun der Ausgangspunkt für unser erstes Beispiel sein: Die Ontologie *Fahrzeuge* soll um die Klasse *Pickup* erweitert werden. Die Klasse *Pickup* soll dabei eine direkte Unterklasse der Klasse *PKW* werden. Zusätzlich soll die Klasse *Pickup* den Slot *ladeflaeche* erhalten, und die Eigenschaften des geerbten Slots *sitzplaetze* sollen geändert werden. In Abbildung 2.2 ist die Ontologie abgebildet, wie sie nach der Änderung aussehen soll.

Der in dieser Arbeit verfolgte Ansatz ist, solche Änderungen in Elementaroperationen zu zerlegen und aus diesen Elementaroperationen Ketten (einfach verkettete Listen) zu bilden, die dann von den Jade Protégé Agenten verarbeitet werden können. Wenn man nochmal die geplante Änderung betrachtet, so kann man sich folgende Elementaroperationen vorstellen: Klasse erzeugen, Oberklasse anhängen, Slot erzeugen, Slot anhängen und Slot-Eigenschaften überschreiben. Diese Elementaroperationen können natürlich nicht in beliebiger Reihenfolge

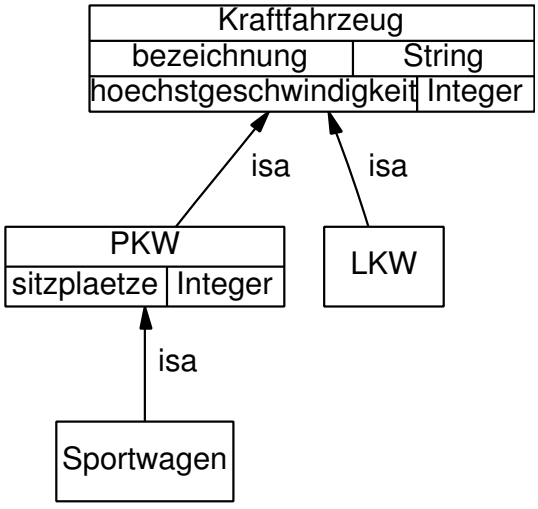

Abbildung 2.1: Beispielontologie *Fahrzeuge* vor der Änderung.

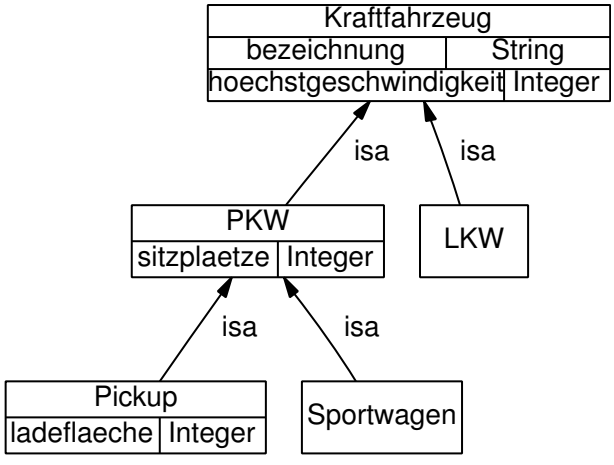

Abbildung 2.2: Beispielontologie *Fahrzeuge* nach der Änderung.

durchgeführt werden. Aber durch die einfache Verkettung der Elementaroperationen wird eine Ordnung auf den Operationen definiert und somit eine Reihenfolge der Elementaroperationen festgelegt.

Die Semantik der Sprechakte und insbesondere der Elementaroperationen und Verarbeitungsketten werden im folgenden Kapitel genauer beschrieben, wobei die Elementaroperationen auch als Aktionen bezeichnet werden.

Kapitel 3

Definition von Verarbeitungsketten in RDF-Schema

Die Sprechakte der Jade Protégé Agenten sind in einem RDF-Schema[15][16] modelliert worden. Die Elementaroperationen wurden dabei auf RDF-Klassen abgebildet, und Instanzen der RDF-Klassen bilden die konkreten Verarbeitungsketten. Da die kompletten Sprechakte der Jade Protégé Agenten in dem RDF-Schema definiert sind, sind auch die Antworten auf Anfragen in dem Schema definiert. Deshalb gliedert sich das Top-Level des RDF-Schemas in zwei Bereiche: Die Klasse `Action` und ihre Unterklassen bilden die Elementaroperationen ab und definieren somit Aktionen zur Änderung und Wartung und zur Abfrage von Ontologien. Die Klassen `Ok`, `Error` und `Warning` und deren Unterklassen definieren die Antworten der Jade Protégé Agenten auf eine Anfrage. Zusätzlich gibt es noch Hilfskonstrukte, die von gewissen Aktionen verwendet werden: Die Klasse `ValueType` fast die Eigenschaften von Werten in Slots zusammen und wird von den Aktionen `createSlot` und `overrideSlot` verwendet, und die Klasse `Querys` und ihre Unterklassen werden von manchen Unterklassen der Klasse `Query` verwendet, um Anfragen zu beantworten.

In Abbildung 3.1 ist der zentrale Teil des RDF-Schemas, die Klasse `Action` und ein Teil ihrer Unterklassen, dargestellt. Die Klasse `Action` und ihre Unterklassen definieren die Elementaroperationen. Die Klasse `Action` definiert selbst nur ein Feld `nextAction`, das auf die nächste Aktion verweist. Somit lassen sich Ketten von Aktionen bilden. Unter der Klasse `Action` hängen drei Unterklassen: `Change`, `Query` und `Start`. Die Klasse `Change` ist die Oberklasse alle Elementaroperationen zum Aufbau und zur Wartung von Ontologien, während die Klasse `Query` die Elementaroperationen zur Abfrage von Ontologien zusammenfasst. Die Klasse `Start` dient zum markieren des Anfangs von Verarbeitungsketten. Außerdem verwaltet die Klasse `Start` Einstellungen, die für die ganze Verarbeitung einer Kette gültig sein sollen. Unterklassen dieser drei Klassen werden je nach Oberklasse im folgenden auch als `Change`-, `Query`- oder `Start`-Aktion genannt. Genauso werden Instanzen dieser Aktionen auch `Change`-, `Query`- oder `Start`-Instanzen bezeichnet werden.

In einer Verarbeitungskette dürfen nie `Change`-Instanzen und `Query`-Instan-

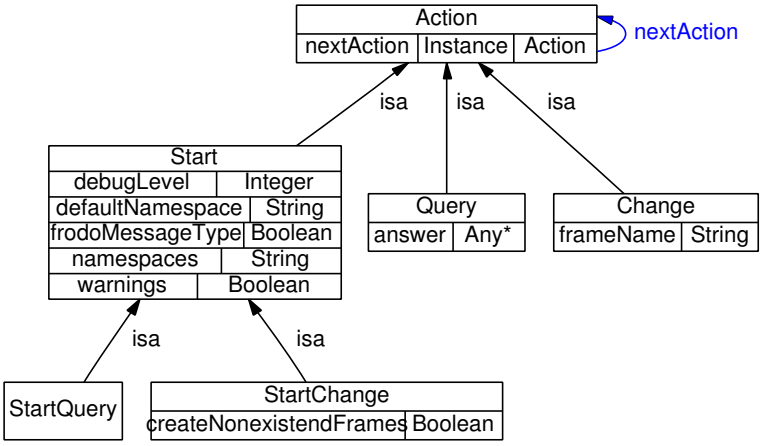

Abbildung 3.1: Überblick über die Toplevel des RDF-Schemas der Sprechakte.

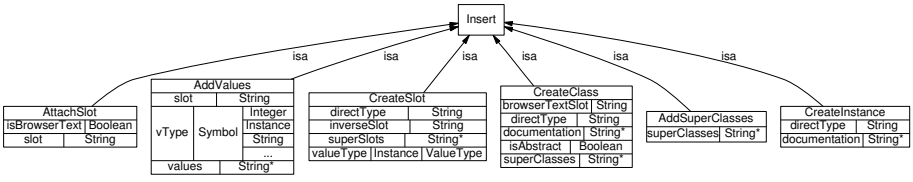

Abbildung 3.2: Überblick über den Zweig `Insert` des RDF-Schemas.

zen gleichzeitig vorkommen. Eine Kette besteht entweder nur aus `Change`-Instanzen oder nur aus `Query`-Instanzen. Die einzige Ausnahme davon ist, das eine Instanz einer `Start`-Aktion den Anfang der Kette markiert. Die Verarbeitungsketten werden auch entsprechend der Art von Aktionen die sie enthalten als `Change`- oder `Query`-Ketten bezeichnet.

Von den `Start`-Aktionen zum Markieren des Kettenanfangs gibt es genau zwei, nämlich `StartChange` und `StartQuery`. `StartChange` markiert den Anfang von `Change`-Ketten und `StartQuery` markiert den Anfang von `Query`-Ketten.

Die `Change`-Aktionen sind nochmals in drei Klassen aufgeteilt: `Insert`-, `Update`- und `Delete`-Aktionen.

Die Klasse `Insert` fasst die Aktionen zum Erweitern von Ontologien zusammen. `Insert`-Aktionen erweitern Ontologien, indem sie neue Klassen, Slots oder Instanzen erzeugen bzw. Slots an Klassen und Werte an Instanzen binden. Einen kurzen, grafischen Überblick über die Klasse `Insert` und die darunter zusammengefassten Aktionen gibt die Abbildung 3.2.

Unter der Klasse `Update` sind die Aktionen zusammengefasst, die Werte von bestehenden Klassen, Slots oder Instanzen ändern. Einen kurzen Überblick über die Klasse `Update` und deren Unterklassen gibt Abbildung 3.3.

Die Klasse `Delete` fasst die Aktionen zusammen, die Frames löschen und die Verbindungen zwischen Slots und Klassen und Instanzen und Werten trennen. Die Unterklassen sind invers zu den Unterklassen von `Insert`. Der einzige

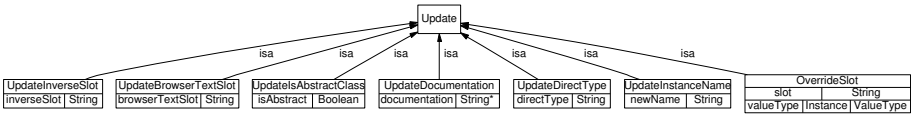

Abbildung 3.3: Überblick über den Zweig `Update` des RDF-Schemas.

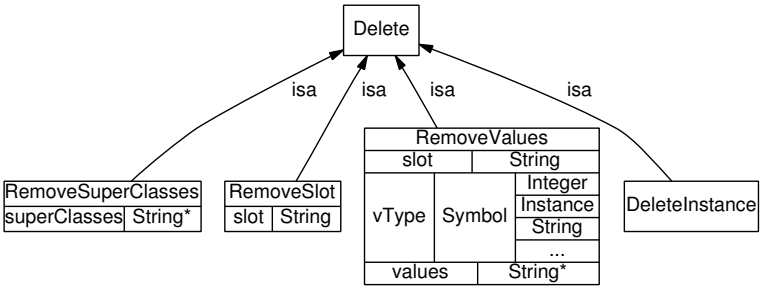

Abbildung 3.4: Überblick über den Zweig `Delete` des RDF-Schemas.

Unterschied ist, dass `DeleteInstance` auch Klassen und Slots löscht und somit invers zu `CreateClass`, `CreateSlot` und `CreateInstance` ist. Ansonsten ist `RemoveSlot` die inverse Aktion zu `AttachSlot`, `RemoveSuperClasses` zu `AddSuperClasses` und `RemoveValues` zu `AddValues`. Abbildung 3.4 gibt einen kurzen Überblick über die Klasse `Delete` und deren Unterklassen.

Im folgenden werden nun Anhand der Beispielontologie *Fahrzeuge* (vgl. Abbildung 2.1 und Abbildung 2.2) der Aufbau und die Verwendung von `Change`- und `Query`-Ketten erläutert. Im ersten Teil wird eine `Change`-Kette aufgebaut, die das neue Konzept *Pickup* in die Beispielontologie einfügt, und im zweiten Teil wird eine `Query`-Kette aufgebaut, die einige Anfragen an eine leicht modifizierte Ontologie *Fahrzeuge* stellt.

3.1 Änderungsketten

Die in der Einführung beschriebene Ontologie *Fahrzeuge* und das einzufügende Konzept *Pickup* sollen in diesem Kapitel als Grundlage dienen, konkret zu zeigen, wie eine Änderungsanfrage auszusehen hat. Dazu muss eine Änderungskette aufgebaut werden, wie sie in Abbildung 3.5 zu sehen ist. Die Abbildung gibt einen Überblick über die Änderungskette und zeigt wie die einzelnen Aktionen verbunden sind. Im folgenden wird nun Schritt für Schritt diese Anfrage aufgebaut und erklärt. Zu beachten ist, dass in diesem Beispiel geschildert wird, wie man mit Protégé Änderungsketten erzeugt. Dies geschieht zum besseren Verständnis und da noch keine Implementierung zur einfachen maschinellen Erstellung von Aktionsketten existiert.

3.1.1 Kopf der Änderungskette erzeugen

Der erste Schritt bei der Erstellung einer Änderungskette ist, eine Instanz der Klasse `StartChange` zu erzeugen. Die Instanz bildet den Kopf der Änderungsket-

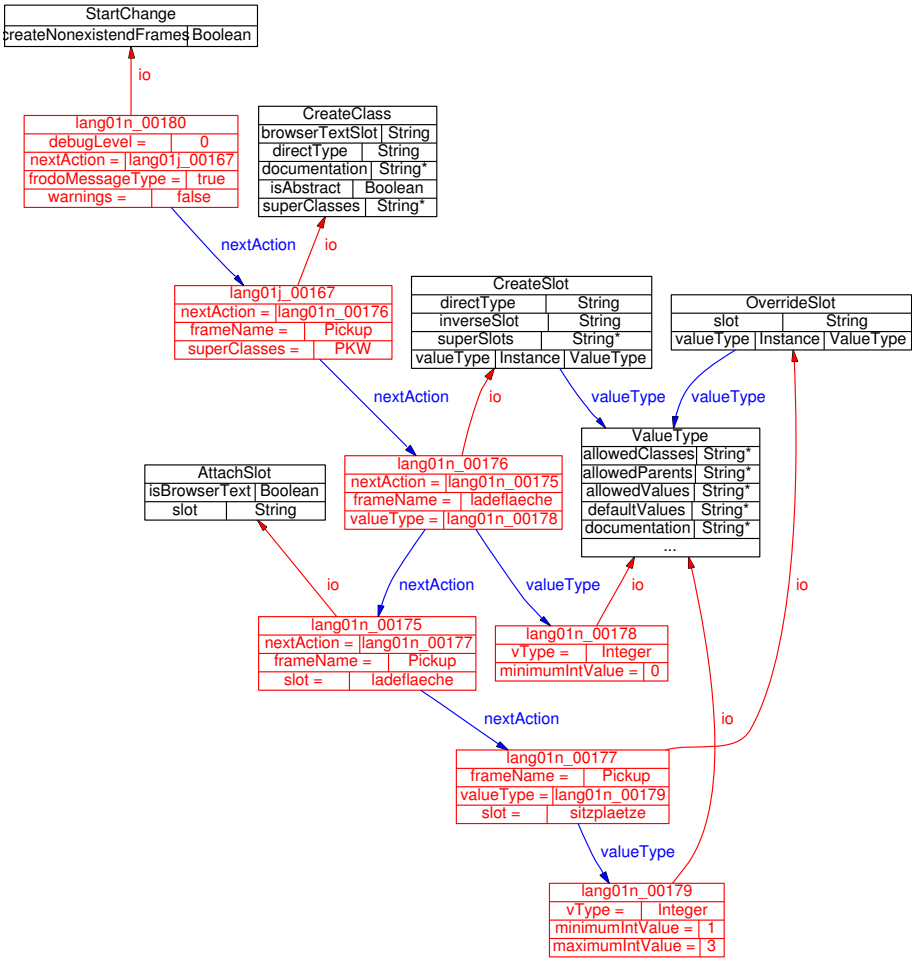

Abbildung 3.5: Darstellung der `Change`-Kette.

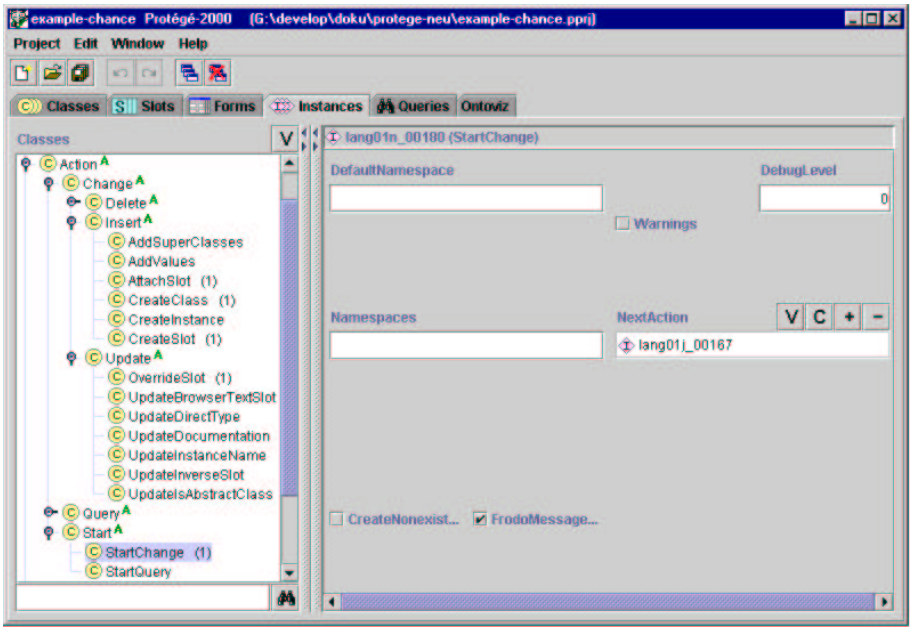

Abbildung 3.6: `StartChange`-Aktion beim Erstellen in Protégé.

te. Eine Änderungskette ist eine einfach verkettete Liste von `Change`-Aktionen. Das Feld `nextAction` ist dabei der Zeiger der immer auf die nächste `Change`-Aktion zeigt. Desweiteren werden in der `StartChange`-Instanz Einstellungen, die für die ganze Kette gelten sollen, gespeichert. Eine Aufzählung aller möglichen Optionen mit ausführlicher Erklärung wird im Anhang A.1 Start-Aktionen gegeben.

Zu Anschauungszwecken wurde die Instanz mit Protégé erzeugt und ist in Abbildung 3.6 mit allen nötigen Eintragungen abgebildet. Wie in der Abbildung zu sehen ist, ist nur die Option `FrodoMessageType` aktiviert und in das Feld `nextAction` etwas eingetragen. Die Option `FrodoMessageType` muss aktiviert werden, da dies von dem JADE-Agenten erwartet wird. Das Feld `nextAction` ist ausgefüllt, da die Änderungskette aus mehr als einer Aktion besteht. Natürlich kann das Feld `nextAction` erst ausgefüllt werden, wenn die nächste Aktion erzeugt worden ist.

3.1.2 Klassen erzeugen

Die nächste Aktion muss die Klasse *Pickup* erzeugen, da die anderen Aktionen auf diese Klasse Bezug nehmen. Die Aktion `CreateClass` ist für das Erzeugen von Klassen zuständig. Also muss eine Instanz der Klasse `CreateClass` erzeugt werden. Nachdem dies geschehen ist, sollte der Name der Instanz in das Feld `nextAction` der Instanz der Klasse `ChangeAction` eingetragen werden, damit die ersten beiden Glieder der Änderungskette miteinander verbunden sind. In der erzeugten Instanz muss das Feld `frameName` ausgefüllt werden, da diese Angabe zum Erzeugen der Klasse in jeden Fall benötigt wird. Der Name der Klasse, die in unserem Beispiel erzeugt werden soll, ist *Pickup* und somit muss dieser

Name in das Feld `frameName` eingetragen werden. Weiterhin kann der Typ der Klasse in dem Feld `directType` angegeben werden. Wird hier nichts angegeben, so wird der Defaultwert *:STANDARD-CLASS* genommen. Mit der Option `isAbstract` kann eine Klasse als abstrakt markiert werden, d.h. von dieser Klasse können keine Instanzen angelegt werden. Die Option `superClasses` erlaubt die Oberklassen anzugeben. In dem Beispiel ist das *PKW*. Wird nichts angegeben, so wird erwartet, dass diese Angabe noch später erfolgt, da in Protégé jede Klasse außer :THING eine Oberklasse haben muss. Am Ende der Abarbeitung einer `Change`-Kette werden alle Klassen überprüft, ob sie eine Oberklasse haben, falls nicht werden sie automatisch an :THING gehängt. In das Feld `documentation` kann, wie der Name schon andeutet, eine eventuell vorhandene Dokumentation zu der erzeugten Klasse eintragen werden. Da bei allen anderen Feldern relativ klar ist welche Funktion sich hinter dem Namen verbirgt, hier nur eine kurze Erklärung zu der Option `BrowserTextSlot`: BrowserTextSlots sind nicht für die Ontologie wichtig, sondern werden nur für die Darstellung der Ontologie in Protégé benötigt. Der Wert des in dem Feld BrowserTextSlot angegebenen Slots wird beim Anschauen einer Instanz dieser Klasse anstatt des Namens der Instanz angezeigt. Da im Normalfall zum Zeitpunkt der Erzeugung einer Klasse noch keine Slots existieren, sollte das Feld `BrowserTextSlot` nicht verwendet werden und stattdessen nach dem Attachen der Slots die Aktion `UpdateBrowserText` Verwendung finden.

3.1.3 Slots erzeugen und mit Klassen verbinden

Der nächste Schritt ist, den Slot *ladeflaeche* zu erzeugen. Die Aktion `Create-Slot` ist zum Erzeugen von Slots da. Deshalb wird eine Instanz der Klasse `CreateSlot` erzeugt. Auch diese Instanz muss wieder in die Änderungskette eingegliedert werden, in dem sie in das Feld `nextAction` der vorherigen Instanz eingetragen wird. Wie auch beim Erzeugen von Klassen ein eindeutiger Klassenname benötigt wird, so wird auch beim Erzeugen von Slots ein eindeutiger Slotname benötigt. Dieser muss in das Feld `frameName` der Instanz eingetragen werden. In unserem Beispiel hat der Slot den Namen *ladeflaeche*, und somit muss *ladeflaeche* in das Feld `frameName` eingetragen werden. Die weiteren Angaben sind optional, das heißt, entweder werden sie nicht benötigt oder es gibt Defaultwerte. So kann in das Feld `directType` der Typ des Slots eingetragen. Wenn nichts angegeben wurde, ist der Defaultwert *:STANDARD-SLOT*. In das Feld `inverseSlot` kann ein eventuell vorhandener inverser Slot eintragen werden. Das Feld `superSlots` ist für Superslots. Superslots funktionieren genauso für Slots, wie Oberklassen für Klassen, nur muss nicht jeder Slot einen Superslot haben. Das Feld `valueType` verweist auf eine Instanz der Klasse `ValueType`. In der Klasse sind Einstellungen für das Speichern von Werten in Slots zusammengefasst. Wird nichts in das Feld `valueType` eingetragen, so wird der Slot mit den Standardwerten von Protégé erzeugt.

Da in unserem Beispiel aber der Typ des Wertes, der in dem Slot *ladeflaeche* gespeichert werden soll, Integer ist, muss also auf jeden Fall ein Defaultwert von Protégé geändert werden. Nach Erzeugung einer Instanz der Klasse `ValueType` (siehe Abbildung 3.5) und Eintragen der Instanz in das Feld `valueType` der Instanz der Klasse `CreateSlot`, kann im Feld `vType` der Typ der Werte auf Integer geändert werden. Da die Ladefläche immer eine positive Größe ist, wird weiterhin in das Feld `minimumIntValue` der Wert Null eingetragen. Eine ausführliche

Erklärung der Klasse `ValueType` und der einstellbaren Werte ist im Anhang A.4 Utility Klassen zu finden.

Als nächstes soll der neu erzeugte Slot *ladeflaeche* an die neu erzeugte Klasse *Pickup* gehängt. Dafür ist die Aktion `AttachSlot` zuständig. Also wird eine Instanz der Klasse `AttachSlot` erzeugt. Die Aktion benötigt natürlich immer die Angaben welcher Slot an welche Klasse gehängt werden soll. In unserem Beispiel ist der Name der Klasse *Pickup*, und somit wird *Pickup* in das Feld `FrameName` eingetragen. Der Name des Slots ist *ladeflaeche* und wird in das Feld `slot` eingetragen. Außerdem muss die Aktion auch wieder in das Feld `nextAction` der vorherigen Aktion, in diesem Fall der `CreateSlot`-Aktion, eingetragen werden. Soll der Wert des Slots als Browsertext der Instanz benutzt werden, so kann dies durch Aktivieren der Option `isBrowserText` veranlasst werden.

3.1.4 Überschreiben von Slots

Jetzt fehlt nur noch das Ändern der Wertebereiche des geerbten Slots *sitzplaetze*. Diese Aktion wird auch als überschreiben des Slots bezeichnet. Dabei ist zu beachten, dass der Wertebereich des überschriebenen Slots immer eine Teilmenge des Wertebereichs des originalen Slots sein muss, damit gilt, dass die Unterklasse eine Spezialisierung der Oberklasse ist. Dies überprüft aber weder das RDF-Tab noch Protégé und liegt somit in der Verantwortung des Nutzers. Mit der Aktion `OverrideSlot` können die Wertebereiche eines Slots überschrieben werden. Zwei Angaben sind dafür immer nötig: Der Name der Klasse, an der der Slot hängt, und der Name des Slots. In dem Beispiel ist der Name der Klasse *Pickup* und der Name des Slots *sitzplaetze*, und somit wird *Pickup* in das Feld `frameName` und *ladeflaeche* in das Feld `slot` eingetragen. Außerdem darf wieder nicht vergessen werden, die Aktion in die Kette einzugliedern, indem die Aktion in das Feld `nextAction` der vorherigen Aktion der Kette eingetragen wird. Als letztes müssen dann noch die Wertebereiche angegeben werden, die überschrieben werden sollen. Dies geschieht - wie bei der Aktion `CreateSlot` - indem eine Instanz der Klasse `ValueType` erzeugt wird und darin die zu überschreibenden Werte des Slots angegeben werden. Diese Instanz wird dann in das Feld `valueType` eingetragen. Wurde nichts in das Feld `valueType`, eingetragen so werden alle überschriebenen Werte des angegebenen Slots an der angegebenen Klasse gelöscht. Überschriebene Eigenschaften des Slots, die an Oberklassen überschrieben wurden, werden dadurch nicht gelöscht.

In der Instanz der Klasse `ValueType` müssen nur die Werte angegeben, die überschrieben werden sollen. In dem Beispiel soll die Anzahl der Sitzplätze eines Pickups auf maximal drei Sitzplätze beschränkt werden. Also wird in das Feld `MaximumIntValue` eine Drei eingetragen. Auch an dieser Stelle sei auf den Anhang A.4 Utility Klassen verwiesen, wo eine ausführliche Erklärung der Klasse `ValueType` und der einstellbaren Werte zu finden ist.

Damit ist die RDF-Message fertig erstellt. Im Anhang A.2 Change-Aktionen sind alle Aktionen zum Ändern und Erweitern von Ontologien kurz beschrieben.

3.1.5 Abarbeitung und Fehlerbehandlung

Wenn die RDF-Message mit Protégé erstellt wurde, muss nur das Projekt gespeichert werden, und die dabei erzeugte RDF-Datei ist eine korrekte RDF-Message. Diese kann dann z.B. mit Hilfe des *testclients* versendet werden. Die Antwort des

RDF-Tab auf diese RDF-Message sollte eine Instanz der Klasse `Ok` sein, und in dem Feld `completed_chains` sollte der Name der `StartChange`-Instanz stehen. Das Feld `completed_chains` enthält immer die Namen der vollständig abgearbeiteten `Change`-Ketten, so dass für den Fall, dass mehrere `Change`-Ketten gesendet wurden, schnell ersichtbar ist, welche Ketten erfolgreich abgearbeitet wurden. Tritt ein Fehler während der Verarbeitung einer `Change`-Kette auf, wird eine Fehlermeldung erstellt, indem eine Instanz einer Unterklasse der Klasse `Error` erzeugt wird.

Einige allgemeine Bemerkungen noch zu Fehlermeldungen des RDF-Tab. Eine vollständige Auflistung aller Fehlermeldungen ist im Anhang A.5 zu finden. Allen Fehlerklassen ist das Feld `error` gemeinsam, in dem eine menschenlesbare Form des Fehlers abgelegt ist. Die Unterklassen der Klasse `Error` teilen sich in zwei Gruppen auf. In der einen Gruppe sind die Fehler, die beim Parsen des RDF-Requests auftreten. Diese Fehler haben immer die Ursache, dass sich die RDF-Message nicht an das RDF-Schema gehalten hat, der Request also nicht das korrekte Format hatte. Tritt ein solcher Fehler auf, bedeutet das immer, dass die Verarbeitung des Requests nicht begonnen werden konnte, da er nicht verstanden wurde. Die andere Gruppe beinhaltet die Fehler, die beim Verarbeiten einer Aktionsinstanz auftreten. Da diese Fehler einer Instanz zugeordnet sind, haben die entsprechenden Fehlerklassen zusätzlich zu dem Feld `error` noch die Felder `instance` und `chainName`, wobei das Feld `instance` den Namen der Instanz und das Feld `chainName` den Namen der Start-Instanz der Kette, in der der Fehler aufgetreten ist, enthält. Bei einem solchem Fehler während der Verarbeitung einer Kette wird die weitere Verarbeitung dieser Kette abgebrochen und eventuelle Veränderungen durch die fehlerhafte Aktion rückgängig gemacht.

Der Zustand der Zielontologie nach einem Fehler in einer Aktion ist der, dass alle Änderungen durch Aktionen, die vor der fehlerhaften Aktion waren, ausgeführt wurden und alle Änderungen durch Aktionen, die nach der fehlerhaften (einschließlich der fehlerhaften) Aktion waren, nicht durchgeführt worden sind.

3.2 Abfrageketten

Nachdem im letzten Abschnitt dargestellt wurde, wie Ontologien mit Jade Protégé Agenten aufgebaut und verändert werden können, folgt jetzt ein Beispiel für die Abfrage von Ontologien. Dabei wird insbesondere auf die Unterschiede zwischen Änderungsketten und Abfrageketten eingegangen. Als Beispiel dient wieder die Ontologie *Fahrzeuge*. Zusätzlich zu dem Konzept *Pickup*, das im vorherigen Beispiel in die Ontologie eingefügt wurde, ist die Ontologie um zwei Instanzen, *Volkswagen* und *Mercedes*, der Klasse *Pickup* erweitert. In Abbildung 3.7 ist die verwendete Ontologie einschließlich aller Erweiterungen abgebildet.

Die Abfragekette soll folgende Aufgaben lösen: Es soll nach geprüft werden, ob die Klasse *Pickup* auch wirklich erzeugt wurde und welchen Typ sie hat. Desweiteren sollen alle Instanzen der Klasse *Pickup*, die genau 3 Sitzplätze haben, gesucht werden, und ein Teil der Ontologie soll zurückgeliefert werden.

In Abbildung 3.8 ist die komplette Anfragekette grafisch dargestellt. Anders als bei `Change`-Aktionen werden bei `Query`-Aktionen, zusätzlich zu den Ok- und Fehlermeldungen, die kompletten Anfragen mit ausgefüllten `answer` Feldern zurückgesendet (siehe Abbildung 3.9).

Ansonsten unterscheiden sich die Abfrageketten nicht wesentlich von den

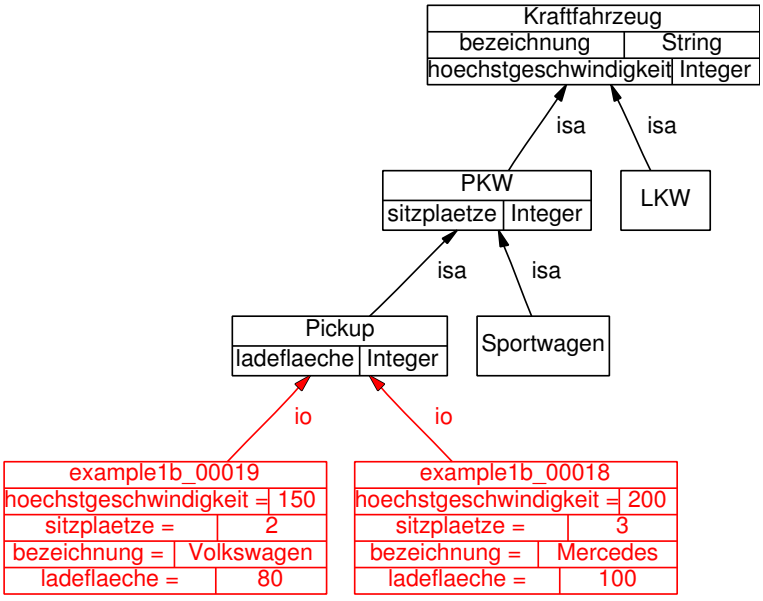

Abbildung 3.7: Beispielontologie für die `Query`-Aktionen.

Änderungsketten. Genauso wie die Aktionen der Änderungsketten, werden die Aktionen der Abfrageketten mit dem Feld `nextAction` miteinander verbunden, und auch die Aktion `StartQuery` zum Markieren des Anfangs einer Abfragekette hat fast die selben Einstellungsmöglichkeiten wie die Aktion `StartChange`.

3.2.1 Erzeugen einfacher Abfrageketten

Der erste Schritt beim Erstellen einer Abfrageketten ist - wie auch beim Erstellen von Änderungsketten - das Erzeugen einer Instanz der entsprechenden `Start`-Aktion. Bei Abfrageketten ist die Klasse `StartQuery` zum Markieren des Anfangs der Kette da. Die möglichen Optionen sind im Anhang A.1 Start-Aktionen beschrieben. Wenn das RDF-Tab nicht alleine benutzt werden soll, muss auf jeden Fall die Option `frodoMessageType` aktiviert sein, da sonst der Protégé Agent die Antwort des RDF-Tab nicht versteht.

Die einfachste `Query`-Action ist die Frage, ob ein Frame mit einem bestimmten Namen in der Zielontologie existiert. Dies wird erreicht, indem eine Instanz der Klasse `ExistsFrame` erzeugt wird und in das Feld `frameName` der Name des zu suchenden Frames eingetragen wird. So können aber nur Frames gefunden werden, die im default Namespace der Ontologie liegen. Ist das nicht der Fall, so muss auch die entsprechende Abrevation, im Format: Abrevation:Framename, angegeben werden. In dem Beispiel soll erfragt werden, ob die Klasse *Pickup* auch wirklich erzeugt wurde. Also wird in das Feld `frameName` der Name der gesuchten Klasse, *Pickup* eingetragen. Das RDF-Tab wird dann bei der Verarbeitung in das Feld `answer` eintragen, ob ein Frame mit dem Namen *Pickup* existiert.

Der Typ der Klasse *Pickup* kann mit Hilfe der Aktion `getDirectType` erfragt werden. Die Verwendung dieser Aktion erfolgt nach dem gleichen Schema wie

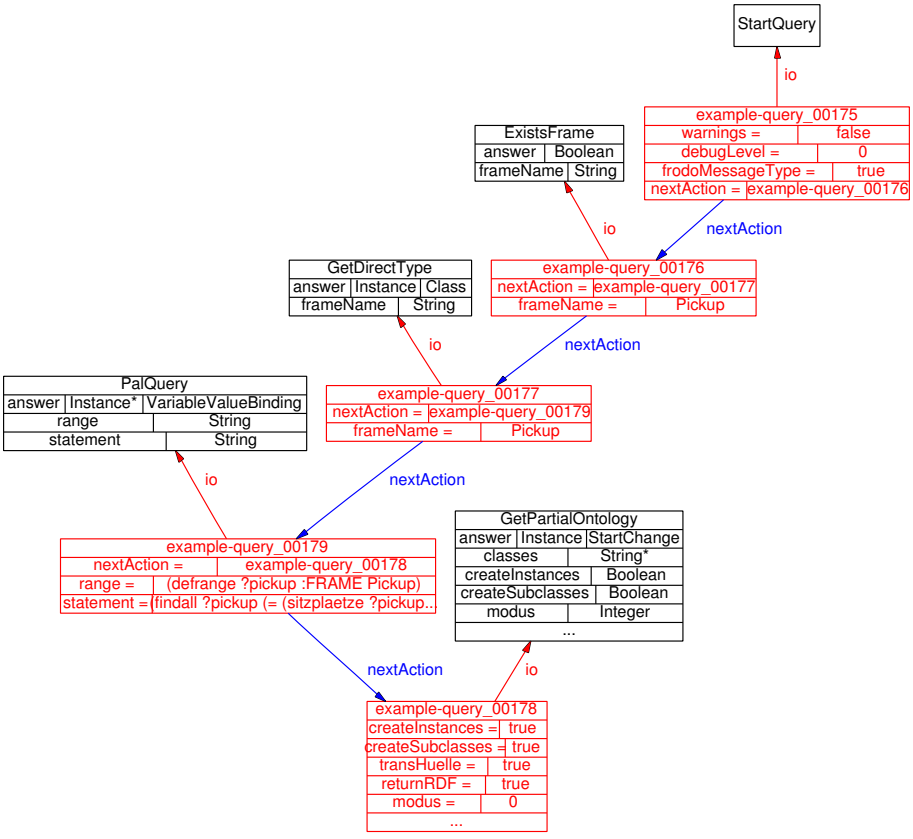

Abbildung 3.8: Darstellung der `Query`-Kette.

bisher. Eine Instanz der Klasse `GetDirectType` wird erzeugt, und in das Feld `frameName` wird der Name des Frames eingeben, dessen Typ gesucht wird, also *Pickup*. Das RDF-Tab wird bei der Verarbeitung der Aktion in das Feld `answer` den Typ des angegebenen Frames eintragen. Da der Typ eines Frames immer eine Klasse ist, wird natürlich immer der in das Feld `answer` eingetragene Wert eine Klasse sein. Aber nicht alle Aktionen liefern immer nur einen Art von Frames (Klassen, Slots oder Instanzen) zurück. Um immer entscheiden zu können, welche Art von Frames gemeint sind, schreibt das RDF-Tab nicht den Namen des Frames in das Feld `answer`, sondern es erzeugt eine entsprechende Instanz der Klassen `Class`, `Slot` oder `Instance` und trägt den Namen des Frames in das Feld `name` ein. Die Instanz wird dann anstelle des Namens des Frames in das Feld `answer` vom RDF-Tab geschrieben, so das einfach die Art des Frames bestimmt werden kann. In dem Beispiel ist der Typ der Klasse *Pickup* die Klasse *:STANDARD-CLASS*. Das RDF-Tab erzeugt also eine Instanz der Klasse `Class` und trägt in das Feld `name` der Instanz *:STANDARD-CLASS* ein. Die Instanz wird dann in das Feld `answer` geschrieben.

3.2.2 Erzeugen komplexer Abfrageketten mit PAL-Queries

Nach diesen beiden relativ einfachen Anfragen, sind die beiden letzten Anfragen komplexerer Natur. Mit der Aktion `PalQuery` lassen sich PAL-Querys (Protégé Axiom Language) stellen. Mit PAL-Querys können Instanzen von Klassen, die bestimmte Eigenschaften haben, gesucht werden. Im Anhang A.3 Query-Aktionen unter `PalQuery` kann mehr über die interne Verarbeitung von Pal-Querys im RDF-Tab erfahren werden, für nähere Informationen über PAL sei auf die PAL-Homepage [1] verwiesen.

Um eine Pal-Query Anfrage zu stellen, wird zuerst eine Instanz der entsprechenden Aktion, hier `PalQuery`, erzeugt. Die Klasse `PalQuery` hat die Felder `answer`, `statement` und `range`. Das Feld `statement` ist für das PAL-Statement und das Feld `range` für die PAL-Range. Für Informationen über die Möglichkeiten und die Syntax von PAL-Statement und PAL-Range sei auf einen Newsgroupartikel [2] über dieses Thema verwiesen. In das Feld `range` kommt, welche Instanzen untersucht werden sollen: *(defrange ?pickup :FRAME Pickup)*, und in das Feld `statement` die eigentliche Suchbedingung: `(findall ?pickup (= (sitzplaetze ?pickup) 3))`.

Bei der Verarbeitung der Aktion wird für jeden gefundenen Wert eine Instanz der Klasse `VariableValueBinding` erzeugt und der Wert in das Feld `variableValue` eingetragen. In das Feld `name` wird der Bereich, indem gesucht wurde eingetragen. Dieser hängt von der Angabe bei PAL-Range ab. In dem Beispiel erfüllt die Instanz *Mercedes* die Bedingung und wird somit zusammen mit dem Bereich *Pickup* eingetragen. Die Instanz wird dann in das Feld `answer` eingetragen.

3.2.3 Lieferung von Teilontologien

Die Aktion `GetPartialOntology` dient zum Ausliefern von Teilen der Zielontologie. Dabei ist das Format der zurückgelieferten Antwort nicht RDF oder eine

[1] `http://protege.stanford.edu/plugins/paltabs/PAL_tabs.html`
[2] `http://protege.stanford.edu/mail_archive/msg00549.html`

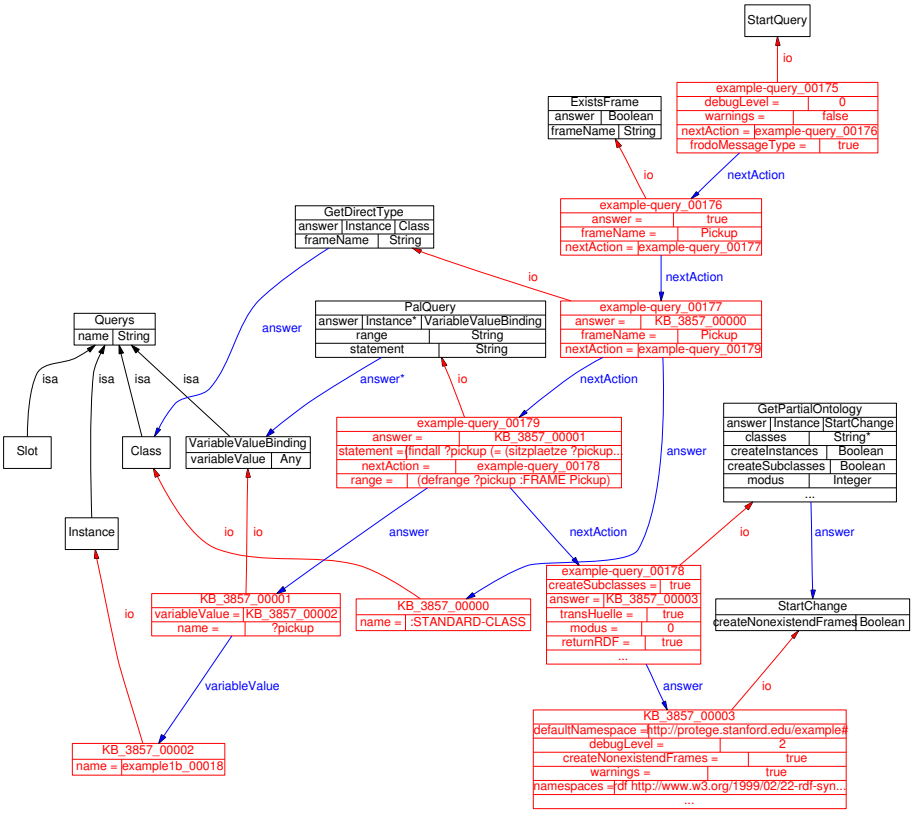

Abbildung 3.9: Darstellung Antwort des RDF-Tabauf die `Query`-Kette.

andere Ontologiebeschreibungssprache, sondern eine Kette von Änderungsaktionen, mit der ein RDF-Tab wieder die Teilontologie erzeugen kann. Dabei ist zu beachten, dass Teilontologien nicht abgeschlossen sein müssen, und deshalb Abhängigkeiten zur Zielontologie vorhanden sein können. Man kann aber mit der Option `transHuelle` einstellen, dass eine abgeschlossene Teilontologie erzeugt wird. In das Feld `classes` werden die Namen der Klassen eingetragen, die den Ausgangspunkt beim Erstellen der Teilontologie sein sollen. Ist das Feld `createSubclasses` aktiviert, so werden alle Unterklassen der Klassen mitgeliefert, andernfalls werden Unterklassen ignoriert. Ebenso werden, falls das Feld `createInstances` aktiviert ist, alle Instanzen der Klassen mitgeliefert. Das Feld `modus` regelt, wie mit direkten Oberklassen verfahren wird. Ist der Wert Null, so wird nichts getan, ist der Wert Eins dann werden Oberklassen mitgeliefert. Denkbar, aber bisher nicht Implementiert, ist ein weiterer Modus, der nur alle benötigten Slots der Oberklassen an die entsprechenden Klassen hängt. In das Feld `answer` wird vom RDF-Tab die erzeugte `StartChange` Instanz eingetragen. Wird das Feld `returnRDF` aktiviert, so wird die erzeugte *Chance*-Kette von dem RDF-Tab verarbeitet und die resultierende Ontologie als RDF und RDFS in eigenen RDF-Containern mit zurückgeschickt.

Die Aktion `GetPartialOntology` geht bei der Erzeugung der Liste, welche Teile der Zielontologie verarbeitet werden müssen, so vor: Als Ausgangspunkt dienen die Namen der Klassen, die im Feld `classes` übergeben wurden. Dann werden die Unterklassen dieser Klassen hinzugefügt, falls das Feld `createSubclasses` aktiviert ist. Danach werden, falls der Wert des Feldes `modus` Eins oder das Feld `transHuelle` aktiviert ist, die Oberklassen aller bis jetzt gefundenen Klassen der Liste hinzugefügt. Wenn das Feld `createInstance` aktiviert ist, werden dann noch alle Instanzen der bisher gefundenen Klassen in die Liste aufgenommen. Dann wird diese Liste abgearbeitet, und die Aktionen zum Erzeugen der Klassen und Instanzen und Ihrer Slots und Werte werden erstellt. Dabei werden Verweise auf fehlende Klassen, Slots oder Instanzen in einer eigenen Liste gespeichert. Falls das Feld `transHuelle` nicht aktiviert ist, werden die Aktionen noch verkettet und die `StartChange` in das Feld `answer` eingetragen. Ist aber das Feld `transHuelle` aktiviert, wird die Liste der fehlenden Klassen, Slots und Instanzen zur neuen Liste der abzuarbeitenden Frames. Dies wird solange wiederholt, bis die Liste der fehlenden Klassen, Slots und Instanzen leer ist. Dann werden alle Aktionen verkettet und die `StartChange` Aktion in das Feld `answer` eingetragen. In dem Beispiel soll die komplette Ontologie zurückgeliefert werden. Deshalb muss in das Feld `classes` nur der Name *Kraftfahrzeug* eingetragen werden, da die Klasse *Kraftfahrzeug* Oberklasse aller Klassen der Ontologie *Fahrzeuge* ist. Außerdem müssen die Optionen `createInstances`, `createSubclasses` und `transHuelle` aktiviert werden. Das Feld `modus` kann Null bleiben, da `transHuelle` aktiviert ist. In Abbildung 3.8 ist die Instanz mit ausgefüllten Feldern zu sehen. Die Antwort des RDF-Tab enthält dann zusätzlich zu den `Query`-Instanzen `Change`-Instanzen die, wenn sie wieder an ein RDF-Tab geschickt werden, die komplette Ontologie aufbauen. Unter `doku/protege-neu/example-query-answer.pprj` ist die Antwort des RDF-Tab wieder in Protégé importiert worden. Die Antwort Message enthält insgesamt 25 `Change`-Anweisungen und die entsprechende StartChange-Anweisung.

Eine vollständige Auflistung aller Aktionen zum Abfragen von Ontologien ist im Anhang A.3 Query-Aktionen zu finden.

Kapitel 4

Softwaretechnische Einbettung mit Protégé und JADE

Die Jade Protégé Agenten bestehen aus zwei Teilen, dem Protégé Agenten und dem RDF-Tab. Das RDF-Tab ist ein Protégé-Plugin[8], das für die Abarbeitung der Verarbeitungsketten zuständig ist. Es kommuniziert über eine offene Schnittstelle mit der Außenwelt. Der Protégé Agent ist ein JADE-Agent, der auf der Basis der FRODO-Agenten[9] implementiert ist. Er ist für die Kommunikation mit der FRODO-Agentenwelt zuständig und leitet Anfragen der Agenten an das RDF-Tab und Antworten des RDF-Tab an die entsprechenden Agenten weiter. In den folgenden beiden Abschnitten werden erst der Protégé Agent und dann das RDF-Tab genauer betrachtet.

4.1 Protégé Agent

Der Protégé Agent ist die Implementierung des Ontologie-Agenten. Er ist in JADE implementiert und dient als Kommunikationsrelay zwischen den anderen JADE-Agenten und Protégé. In Abbildung 4.1 ist das Kommunikationsverhalten des Protégé Agenten abgebildet. Der Protégé Agent ist nicht sehr komplex, da er nicht viel Funkionalität besitzt. Seine einzige Tätigkeit ist auf eingehende ACL-Messages von anderen JADE-Agenten zu warten. Mit ACL-Messages kommunizieren die JADE-Agenten untereinander. ACL-Messages sind ähnlich wie E-Mails aufgebaut. Sie besitzen ein eindeutiges Absender und Empfängerfeld, ein Subject-Feld in dem aber kein Freitext, sondern ein ACL-Messagecode nähere Angaben zur Nachricht macht und ein Contentfeld indem die eigentliche Nachricht abgespeichert ist. Im Projekt FRODO wurden die ACL-Messages erweitert, so das sie unteranderem mehrere Nachrichten mit Hilfe von Containern im Contentfeld speichern können. Die erweiterten ACL-Messages werden FRODO-Messages genannt.

Empfängt der Protégé Agent eine ACL-Message, so wird ein neuer Thread erzeugt. Als erstes überprüft der Protégé Agent, ob die ACL-Message auch eine FRODO-Message ist. Außerdem erwartet er, das die eingehende Nachricht

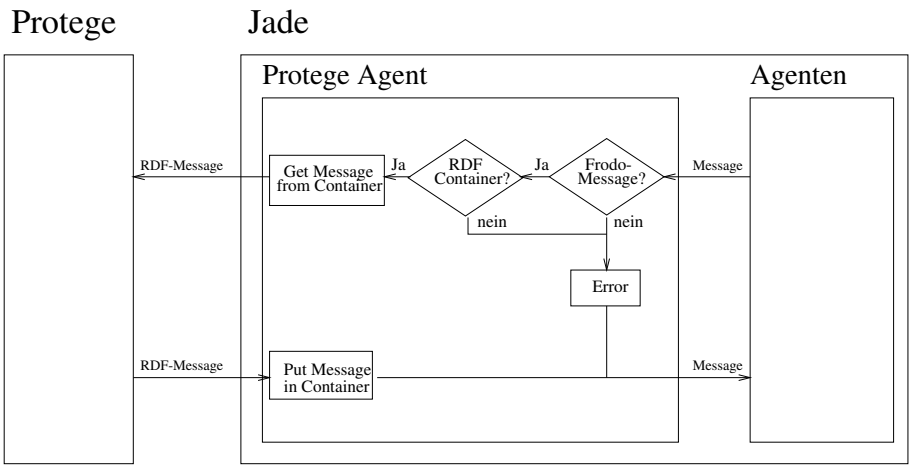

Abbildung 4.1: Interner Aufbau des Jade Protégé Agenten

einen RDF-Container mit einer bestimmten ID hat, die in dem Java-Interface *ContainerIDs* definiert ist. Dieser RDF-Container enthält per Definition die Anfrage. Ist eine dieser Vorraussetzungen nicht gegeben, so kann er mit der empfangenen Nachricht nichts anfangen. In dem Fall erzeugt er eine Antwort auf die eingegangene Nachricht und gibt in dem Subjektfeld den ACL-Messagecode NOT_UNDERSTOOD an. Die Antwort sendet er dann an den Sender der Nachricht zurück.

Handelte es sich um eine FRODO-Message mit entsprechendem RDF-Container dann holt der Protégé Agent die Anfrage aus dem RDF-Container und schickt diese an den RDF-Tab in Protégé. Wenn das RDF-Tab in Protégé die Verarbeitung der Anfrage beendet hat schickt es das Ergebnis zurück an den Protégé Agent, der das Ergebnis dann wieder in eine FRODO-Message packt und an den Sender schickt. Da die Verarbeitung der Nachrichten in einem eigenen Thread ablaufen ist der Protégé Agent während das RDF-Tab eine Anfrage abarbeitet nicht blockiert, die Serialisierung der Anfragen erfolgt im RDF-Tab.

4.2 RDF-Tab

Das RDF-Tab ist für die Verarbeitung der Requests zuständig. Eingehende Requests werden in eine Warteschlange gestellt, die nach dem FIFO Prinzip (First In First Out) arbeitet. Dadurch ist das RDF-Tab, während es eine Anfrage verarbeitet, nicht blockiert und kann weitere Anfragen entgegennehmen. Die Schnittstelle zum Empfangen von Requests und Versenden von Replys ist ein TCP-Socket. Von eingehenden Requests wird erwartet, dass sie als (Java Objekt) String kodiert sind und sich an das definierte RDF-Schema halten. Die Antworten des RDF-Tab auf Requests sind entweder auch ein (Java Objekt) String, wobei dann aber nicht der volle Funktionsumfang zur Verfügung steht, oder ein (Java Objekt) RDF-Container. Das Reply auf ein Request wird immer dann in einen RDF-Container verpackt, wenn eine der Ketten die Option `frodoMessageType` aktiviert hatte. Bei den Request gilt es zwischen `Query-`

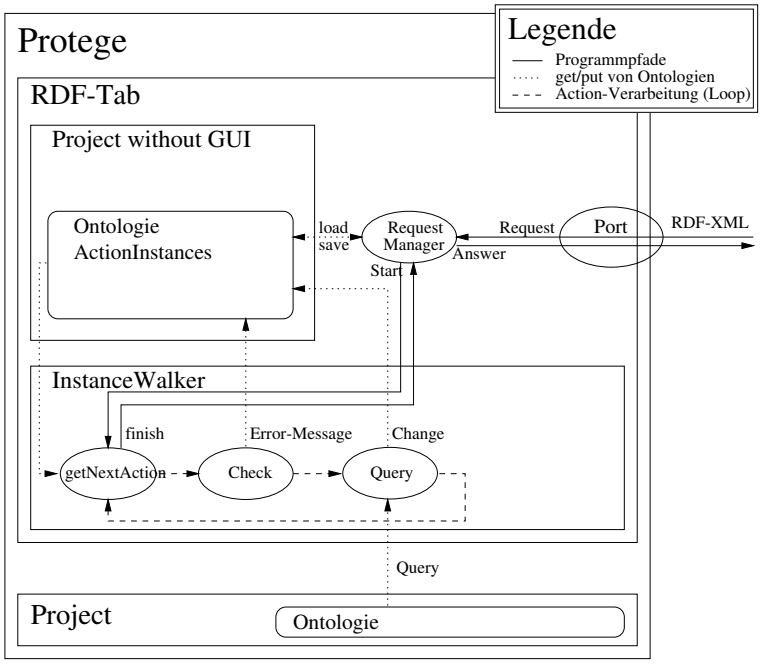

Abbildung 4.2: Verarbeitung von `Query`-Requests durch das RDF-Tab

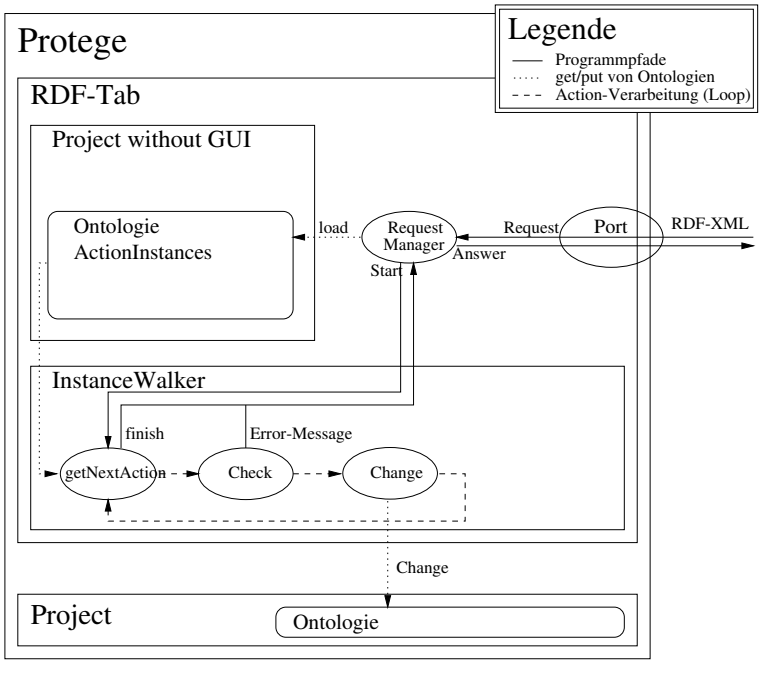

Abbildung 4.3: Verarbeitung von `Change`-Requests durch das RDF-Tab

und `Change`-Requests zu unterscheiden, da diese leicht unterschiedlich verarbeitet werden, wie auch in den Abbildungen 4.2 und 4.3 zu sehen ist. Ein Großteil der Verarbeitungsschritten von Requests ist jedoch identisch. Deshalb wird im folgenden Teil die Verarbeitung beider Requestarten gemeinsam beschrieben und nur an den Stellen, wo die Verarbeitung sich unterscheidet, kurz auf diese Unterschiede eingegangen.

Das RDF-Tab besitzt intern ein eigenes Protégé Projekt. In dieses wird der zu verarbeitende Request als erstes geladen. Das Laden übernimmt das zu Protégé gehörende RDF-Backend. Nach dem Laden des Requests in das Projekt des RDF-Tab ist der erste Schritt der Verarbeitung die Ermittlung aller Instanzen der Klassen `StartQuery` und `StartChange`, da diese den Anfang von Ketten von Anfragen markieren. Wenn keine Instanzen dieser beiden Klassen gefunden werden, so wird der Fehler `NoStartInstance` zurückgesendet und die Verarbeitung abgebrochen. Werden Instanzen beider Klassen gefunden, so wird der Fehler `ChangeAndQueryInstances` erzeugt, da in einem Request nur Ketten einer Art bearbeitet werden können. Die weitere Verarbeitung wird auch in diesem Fall abgebrochen und der Fehler zurückgesendet. Ansonsten werden die gefundenen Ketten in einer nicht definierten Reihenfolge abgearbeitet. Es wird einfach die erste gefundene `Start`-Instanz genommen, und die globalen Einstellungen für alle dazugehörigen Aktionen werden mit den Werten aus der `Start`-Instanz initialisiert. Außerdem werden die Namespaceinformationen aus der `Start`-Instanz ausgelesen und die Namespacetransformation initialisiert. Danach werden die Aktionen der Kette der Reihe nach ausgeführt. Handelt es sich bei der Kette um eine Kette mit `Query`-Aktionen so werden die Ergebnisse der Aktionen in das Feld `answer` geschrieben. Falls ein Fehler während der Ausführung einer Aktion auftritt, so wird die weitere Verarbeitung der Kette gestoppt, alle Änderungen der fehlerhaften Aktion rückgängig gemacht und die entsprechende Fehlermeldung generiert. Trat kein Fehler auf, so wird die Kette als erfolgreich abgearbeitet markiert, und die nächste Kette wird ausgeführt. Falls es keine weitere Kette gibt, wird die Antwort auf den Request erzeugt. Wurde mindestens eine Kette erfolgreich abgearbeitet, so wird eine Instanz der Klasse `Ok` erzeugt und die Namen der Start-Instanzen der erfolgreich abgearbeiteten Ketten in das Feld `completed_chains` eingetragen. Gab es in einer oder mehrerer Ketten einer Fehler, werden die erzeugten Fehler-Instanzen der Antwort hinzugefügt. Handelte es sich bei dem Request um einen Request aus `Change`-Aktionen, so ist das die Antwort die zurückgesendet wird. Waren hingegen `Query`-Aktionen in dem Request, werden der Antwort noch der gesamte Request mit ausgefüllten Answer-Feldern hinzugefügt. Anschließend werden alle Klassen und Instanzen aus dem Projekt des RDF-Tab gelöscht, und die Bearbeitung des nächsten Requests kann beginnen.

Kapitel 5

Zusammenfassung und Ausblick

In dieser Arbeit wurden das Design und die Implementierung eines Ontologie-Agenten dargestellt. Der Agent wurde auf Basis der FIPA-konformen Agentenplattform JADE implementiert. Für die Verwaltung der Ontologien wurde Protégé verwendet. Protégé ist ein sehr mächtiges Tool zum Erstellen und Verwalten von Ontologien. Um mit dem Ontologie-Agenten auf die Funktionalität von Protégé zugreifen zu können, wurde Protégé mit einem Plugin um eine Schnittstelle erweitert, mit der externe Programme mit Protégé kommunizieren können. Die Sprechakte dieser Schnittstelle sind Ketten von elementare Operationen auf Ontologien. In einem RDF-Schema sind die Sprechakte definiert, wobei die elementaren Operationen auf RDF-Klassen abgebildet sind.

Die bisherige Arbeit stellt eine Basis-Funktionalität für Ontologie-Agenten zur Verfügung. Diese sollte in zwei Richtungen erweitert werden um einen vollständigen Ontologie-Agenten zu erhalten, der in Organizational Memories verwendet werden kann: Eine Aufgabe ist, die Anfragen zu vereinfachen. Die Sprechakte des Ontologie-Agenten sind sehr elementare Operationen; diese müssen durch eine auf diesen elementaren Operationen aufbauende, Abstraktionsebene zu komplexeren Sprechakten aufgebaut werden. Der andere Bereich ist das Rollenverhalten von Ontologie-Agenten.

Neben der Verwaltung der Konzeptualisierung sollte auch der *sharing scope* der Ontologie explizit im System gehandhabt werden. In [13][14] wird dazu ein explizites Rollenmodell für *Ontology Societies* vorgeschlagen, und in [4] findet man Protokolle, mit denen Agenten die Inhalte von Ontologien verhandeln können. Eine Integration dieser beiden Konzepte in das vorgestellte System ist für die nähere Zukunft geplant.

Literaturverzeichnis

[1] A. Abecker, A. Bernardi, A. Dengel, L. van Elst, M. Malburg, M. Sintek, S. Tabor, A. Weigel, and C. Wenzel. FRODO: A Framework for Distributed Organizational Memories. Project Proposal, DFKI GmbH Kaiserslautern, 2000. URL: `http://www.dfki.uni-kl.de/frodo/`.

[2] A. Abecker, A. Bernardi, L. van Elst, A. Lauer, H. Maus, S. Schwarz, and M. Sintek. Frodo: Milestone m1. Technical report, DFKI, 2002.

[3] Andreas Abecker, Ansgar Bernardi, Knut Hinkelmann, Otto Kühn, and Michael Sintek. Toward a Technology for Organizational Memories. *IEEE Intelligent Systems*, 13(3):40–48, June 1998.

[4] S. C. Bailin and w. Truszkowski. Ontology negotiation using JESS. In *3rd Int. Conference on Enterprise Information Systems*, 2001.

[5] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. Jade - a fipa-compliant agent framework. Technical report, Centro Studi e Laboratori Telecomunicazioni (CSELT), 1999.

[6] V.R. Benjamins, D. Fensel, and A. Gómez Pérez. Knowledge management through ontologies. In *[12]*, 1998.

[7] T. R. Gruber. A translation approch to portable ontology specifications. *Knowledge Aquisition*, 5:199–220, 1993.

[8] Stanford Medical Informatics. Protégé-2000 Programming Development Kit, unknown. URL: `http://protege.stanford.edu.doc/pdk/index.shtml`.

[9] Andreas Lauer. FRODO AgentFramework, 2002. URL: `http://www.dfki.uni-kl.de/frodo/frodoianer/FrameworkOverview_1_3_02.htm`.

[10] N. F. Noy, R. W. Fergerson, and M. A. Musen. The knowledge model of Protege-2000: Combining interoperability and flexibility. In *In the Proceedings of the 12th International Conference on Knowledge Engineering and Knowledge Management (EKAW 2000)*, 2000.

[11] D. O'Leary. Using AI in Knowledge Management: Knowledge Bases and Ontologies. *IEEE Intelligent Systems*, 13(3):34–39, June 1998.

[12] U. Reimer. *PAKM-98: Practical Aspects of Knowledge Management. Proc. of the Second Int. Conference.* October 1998. URL: `http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-13/`.

[13] L. van Elst and A. Abecker. Domain Ontology Agents in Distributed Organizational Memories. In *IJCAI'2001 Working Notes of the Workshop on Knowledge Management and Organizational Memories*, pages 39–48, 2001.

[14] L. van Elst and A. Abecker. Ontology-Reletad Services in Agent-Based Distributed Information Infrastructures. In *Proceedings of the Thirteenth International Conference on Software Engineering & Knowledge Engineering*, pages 79–85, 2001.

[15] W3C. Resource Description Framework (RDF) Schema Specification 1.0, 2001. URL: `http://www.w3.org/TR/2000/CR-rdf-schema-20000327/`.

[16] W3C. Semantic Web Activity: Resource Description Framework (RDF), 2001. URL: `http://www.w3.org/RDF/`.

Anhang A

Referenz

In diesem Kapitel werden ausführlich alle in dem RDF-Schema `lang01.rdfs` definierten Aktionen beschrieben.

Bei der Tabelle der Felder eines Befehls gelten folgende Bestimmungen: Ist die Kardinalität in Klammern geschrieben, so ist das Feld optional. Ist der Typ Instanz, so ist bei Ziel die Oberklasse angegeben, die der Typ der Instanz besitzen muss.

A.1 Start-Aktionen

- `Start`
  - Felder:

Name	Typ	Kardinalität	Ziel
`defaulNamespace`	String	(single)	
`namespaces`	String	(single)	
`warnings`	Boolean	(single)	
`debugLevel`	Integer	(single)	
`frodoMessageType`	Boolean	single	
`nextAction`	Instance	(single)	`Change,Query`

  - Beschreibung:
    Den Anfang aller Ketten von Aktionen bildet eine Aktion der Klasse `Start`. Je nach Art der Kette - `Query`- oder `Change`-Kette - muss die entsprechende `Start`-Aktion - `StartChange` oder `StartQuery` - verwendet werden. Diese `Start`-Instanzen beinhalten globalen Einstellungen, die für die gesamte Verarbeitung einer Kette gelten. Die abstrakte Klasse `Start` faßt die gemeinsamen Einstellungen zusammen. Im folgenden wird nun erklärt was die einzelnen Optionen tun:
    - `FrodoMessage`: Die Option `frodoMessage` ist die einzige globale Option, d.h. ist sie in einer Verarbeitungskette aktiviert, gilt sie für die gesamte Anfrage. Ist die Option aktiviert, wird als Antwort des RDF-Tab ein FRODOContent-Objekt verschickt. Der Vorteil von dem FRODOContent-Objekten ist, das diese beliebig viele Container aufnehmen können, und so mehrere Objekte

einfach als eine Antwort versendet werden können. Die eigentliche Antwort wird dabei in einen RDF-Conatiner verpackt, der wiederum in dem FRODOContent-Objekt gespeichert wird. Daneben können noch weitere Container in dem FRODOContent-Objekt gespeichert werden. Diese Funktionalität wird eigentlich nur dann benötigt, wenn in der Aktion `getPartialOntologie` die Option `returnRDF` aktiviert ist, um das von dieser Aktion erzeugte RDF und RDFS zu übertragen. Der Protégé Agent erwartet, das die Option aktiviert ist und die Antworten immer in einem FRODOContent-Objekt verpackt sind. Wenn die Option nicht aktiviert ist wird einfach ein (Java Objekt) String als Antwort verschickt.

  * `DebugLevel`: Die Option gibt an wie viele Informationen während der Verarbeitung der Aktionen ausgegeben werden sollen. Sinnvolle sind Werte: 0, 1 und 2. Wobei 0 für keinerlei Meldungen, 1 für die Ausgabe von allen wichtigen Parametern einer Aktion steht und 2 für alle möglichen weiteren wichtigen Informationen.
  * `defaultNameSpace`: In das Feld wird falls vorhanden der default Namespace eingetragen, indem alle Frames ohne Abrivation liegen. Das sollte normalerweise eine URI sein. Wird in dieses Feld nichts eingetragen, so werden alle Framenamen ohne Abrivation in den default Namespace der Zielontologie eingetragen. Ist der eingetragene Namespace ungleich dem default Namespace in der Zielontologie, so wird der Namespace als weiterer Namespace der Zielontologie hinzugefügt, und automatisch alle Framenamen mit der entsprechenden Abrivation versehen.
  * `Namespaces`: Werden in einer Anfrage mehr als ein Namespace verwendet, so müssen die weiteren Namespaces in dieses Feld in der Form: abrivation1 Leerzeichen namespace1 Leerzeichen abrivation2 Leerzeichen namespace2 usw. eingetragen werden. Fehlende Namespaces in der Zielontologie werden automatisch eingetragen und Framenamen gegebenenfalls automatisch korrigiert.
  * `Warnings`: Ist diese Option aktiviert werden, wenn das RDF-Tab automatisch Änderungen an der Zielontologie durchführt, zusätzlich noch Warnungen über die Änderungen erzeugt. Zur Zeit ist das nur dann der Fall, wenn am Ende einer `Change`-Kette Klassen existieren die keine Oberklasse besitzen. Diese werden automatisch an die Klasse *:THING* gehängt. Ist die Option `warnings` aktiviert, wird zusätzlich noch eine Warnung erzeugt, das diese Änderung erfolgt ist.

- `StartChange`
  - Felder:

Name	Typ	Kardinalität	Ziel
`debugLevel`	Integer	(single)	
`defaulNamespace`	String	(single)	
`frodoMessageType`	Boolean	single	
`namespaces`	String	(single)	
`warnings`	Boolean	single	
`createNonexistendFrames`	Boolean	(single)	
`atErrorUndo`	Boolean	(single)	
`nextAction`	Instance	(single)	`Change`

– Beschreibung:
Die Aktion `StartChange` markiert den Anfang von `Change`-Ketten. Zusätzlich zu den von der Klasse `Start` geerbten Optionen, besitzt die Aktion `Startchange` noch die Optionen:

  * `CreateNonexistendFrames`: Die Option hat nur Auswirkungen auf `Insert`-Aktionen, `Update`- und `Delete`-Aktionen ändern ihr Verhalten durch aktivieren dieser Option nicht. Ist die Option deaktiviert wird bei einem fehlenden Frame die Verarbeitung der aktuellen Kette abgebrochen und der Fehler `MissingFrame`, mit der Aktion, dem Namen des fehlenden Frames und dem Namen der `Start`-Instanz, erzeugt. Ist die Option aber aktiviert, so wird zuerst versucht aus der Art der Aktion zu bestimmen welcher Typ (Klasse, Slot oder Instanz) der fehlende Frame hat. Wenn das gelingt wird der Frame automatisch erzeugt, falls nicht wird der Fehler `CantIdentifyType` erzeugt.
    Um etwas die Möglichkeiten und Grenzen dieser Option aufzuzeigen, eine kurze Erklärung für was sie Gedacht ist und was Probleme bereitet. Ziel ist es, das keine totale Ordnung der Aktionen nötig ist. Zum Beispiel wenn viele neue Klassen erzeugt werden sollen, die zum Teil in einer Unterklassen-Relation zu einander stehen, müssten die Aktionen zum Erzeugen dieser Klassen so angeordnet werden, das keine Abhängigkeit verletzt wird, also das die entsprechenden Oberklassen immer vor den eigentlichen Klassen erzeugt werden. Durch aktivieren dieser Option können nun die Klassen in beliebiger Reihenfolge erzeugt werden. Existiert eine Oberklasse zum Zeitpunkt der Erzeugung der Unterklasse noch nicht, wird diese einfach erzeugt und es wird vermerkt, das diese Klasse automatisch erzeugt wurde. Kommt dann später die Aktion zum Erzeugen der automatisch erzeugten Klasse, wird die Klasse nicht neu erzeugt, sondern nur geändert. Leider funktioniert das nicht überall, da z.B. an einen *any*-wertigen Slot Klassen, Slots und Instanzen gehängt werden können und somit nicht entschieden werden kann welcher Typ der fehlende Frame hat. Deshalb würde hier auch das aktivieren der Option nichts bringen. Allgemein lässt sich sagen, das die Option immer funktioniert (sollte) sofern die Aktionen gleichen Typs zusammengefasst und in folgender Reihenfolge übergeben werden: `CreateClass`, `CreateSlot`, `AttachSlot`, `OverrideSlot`, `CreateInstance` und als letztes die `AddValues` Aktionen.
  * `atErrorUndo`: Die noch experimentelle Option versucht bei ei-

nem Fehler nicht nur die Änderungen der aktuellen Aktion rückgängig zu machen, sondern auch alle Aktionen der Kette, die vorher erfolgreich ausgeführt worden sind. Das funktioniert bisher für alle `Change`-Aktionen bis auf `DeleteInstance`. Das hat den Grund, das wenn ein Frame gelöscht wird, abhängige Frames mit gelöscht werden und auch die ganzen Links auf dieses Frame gelöscht werden.

- `StartQuery`
  - Felder:

Name	Typ	Kardinalität	Ziel
`debugLevel`	Integer	(single)	
`defaulNamespace`	String	(single)	
`frodoMessageType`	Boolean	single	
`namespaces`	String	(single)	
`warnings`	Boolean	single	
`nextAction`	Instance	(single)	`Query`

  - Beschreibung:
    Die Aktion `StartQuery` markiert den Anfang von `Query`-Ketten. Sie besitzt keine weiteren Optionen als die von `Start` geerbten. Eine Besonderheit hat sie aber trotzdem, das RDF-Tab füllt die Felder `defaultNamespace` und `namespaces` in dem Reply mit den Namespaces aus die in der Zielontologie vorkommen, aber nicht in dem Request vorkommen. Das bedeutet nicht das irgendeiner dieser Namespaces auch in dem Reply verwendet wird, nur das in der Zielontologie noch folgende weitere Namespaces vorkommen und diese möglicherweise in dem Reply vorkommen. Dadurch lässt sich durch Abschicken einer leeren `StartQuery`-Instanz alle verwendeten Namespaces der Zielontologie ermitteln.

A.2 Change-Aktionen

A.2.1 Insert-Aktionen

- `AddSuperClasses`
  - Felder:

Name	Typ	Kardinalität	Ziel
`frameName`	String	single	
`superClasses`	String	multiple	
`nextAction`	Instance	(single)	`Change`

  - Beschreibung:
    Mit `AddSuperClasses` wird an eine Klasse weitere Oberklassen hinzugefügt. Der Name der Klasse an die Oberklassen hinzugefügt werden soll, kommt in der Feld `frameName`. Die Namen der Oberklassen werden in dem Feld `superClasses` angegeben.
  - Fehlermeldungen:
    * `MissingFrameName`
    * `MissingFrame`

    * `FrameHasWrongType`
    * `ClassAlreadyHasSuperclass`
  - Ideen:
    * Optional wenn das Frame bereits die Oberklasse hat nur eine Warnung ausgeben.

- `AddValues`
  - Felder:

Name	Typ	Kardinalität	Ziel
`frameName`	String	single	
`slot`	String	single	
`values`	String	multiple	
`vType`	Symbol	(single)	
`nextAction`	Instance	(single)	`Change`

  - Beschreibung:
    Der Befehl `addValues` hängt an einen Slot, der an eine Klasse gebunden ist, Werte. In das Feld `frameName` kommt der Name der Klasse, in das Feld `slot` der Name des Slots und in das Feld `values` kommen die Werte, die in dem Slot gespeichert werden sollen. Hierbei ist zu beachten, das die Werte als Strings benötigt werden. Ist der Typ des Slots *Class* und *Instance* muss der Name der Instanz angegeben werden. Ist er *ANY* so dürfen in einem `addValues`-Befehl nur Werte von einem Typ hinzugefügt werden, d.h. wenn mehrere Werte von verschiedenem Typ an einen *ANY*-wertigen Slot gehängt werden sollen, so muss das mit mehreren `addValues`-Befehlen geschehen. Außerdem muss bei *ANY*-wertigen Slots explizit im Feld `vType` der Typ der angehängten Werte angegeben werden.
  - Fehlermeldungen:
    * `MissingFrameName`
    * `MissingFrame`
    * `SlotNotAttached`
    * `MultipleValuesOnValueSlot`

- `AttachSlot`
  - Felder:

Name	Typ	Kardinalität	Ziel
`frameName`	String	single	
`slot`	String	single	
`isBrowserText`	Boolean	(single)	
`nextAction`	Instance	(single)	`Change`

  - Beschreibung:
    Mit dieser Aktion werden Slots an Klassen gehängt. Dazu wird der Name der Klasse, an die der Slot gehängt werden soll, im Feld `frameName` vermerkt. Und den Namen des Slots, der an die Klasse gehängt werden soll, wird in das Feld `slot` eingetragen. Es kann direkt angegeben werden ob der Slot *BrowserTextSlot* sein soll, indem in das Feld `isBrowserText` *true* eingetragen wird, ansonsten ist der Defaultwert *false*.

- Fehlermeldungen:
  - `MissingFrameName`
  - `MissingFrame`

- `CreateClass`
  - Felder:

Name	Typ	Kardinalität	Ziel
`frameName`	String	single	
`directType`	String	(single)	
`superClasses`	String	(multiple)	
`isAbstract`	Boolean	(single)	
`documentation`	String	(multiple)	
`browserTextSlot`	String	(single)	
`nextAction`	Instance	(single)	`Change`

  - Beschreibung:
    Der Befehl `createClass` erzeugt eine neue Klasse. Die einzige Angabe die dazu benötigt wird, ist ein eindeutiger Klassenname. Dieser muss in das Feld `frameName` eingetragen werden. Natürlich ist es sinnvoll direkt noch weitere Eigenschaften anzugeben. In das Feld `superClasses` können die Oberklassen der neuen Klasse angegeben. Ist die Option `createMissingFrames` aktiviert so müssen die Oberklassen beim Erzeugen der Klasse noch nicht existieren. Mit dem Feld `isAbstract` kann eine Klasse als abstrakt markiert werden. Beim Erzeugen von Klassen wird als Defaultwert der Standardtyp *:STANDARD-CLASS* verwendet, wenn ein anderer Typ benötigt wird, kann der Name dieses Typs in dem Feld `directType` angeben werden. Das Feld `documentation` ist zur Dokumentation der Klasse, diese wird in der Klasse gespeichert. Im Feld `browserTextSlot` kann ein entwagiger BrowserTextSlot angegeben werden, wenn er zu diesem Zeitpunkt schon bekannt ist.
  - Fehlermeldungen:
    - `MissingFrameName`
    - `MissingFrame`
    - `ClassIsAbstract`
    - `WrongDirectType`
    - `FrameHasWrongType`
    - `FrameExist`

- `CreateInstance`
  - Felder:

Name	Typ	Kardinalität	Ziel
`frameName`	String	single	
`directType`	String	single	
`documentation`	String	(multiple)	
`nextAction`	Instance	(single)	`Change`

  - Beschreibung:
    Erzeugt eine neue Instanz einer Klasse. Es können keine Klassen

oder Slots mit dieser Aktion erzeugt werden, auch wenn Klassen und Slots Instanzen der Klasse *:STANDARD-CLASS* bzw. *:STANDARD-SLOT* sind. Außerdem können keine Instancen von abstrakten Klassen erzeugt werden. In das Feld `frameName` kommt der Name der Instanz und in das Feld `directType` der Name der Klasse, von der die Instanz abgeleitet werden soll. Falls irgendwelche Dokumentation existiert kann diese in das Feld `documentation` eingetragen werden.

  - Fehlermeldungen:
    * `MissingFrameName`
    * `MissingFrame`
    * `ClassIsAbstract`
    * `FrameHasWrongType`
    * `FrameExist`

- `CreateSlot`
  - Felder:

Name	Typ	Kardinalität	Ziel
`frameName`	String	single	
`valueType`	Instance	(single)	`ValueType`
`directType`	String	(single)	
`inverseSlot`	String	(single)	
`superSlots`	String	(multiple)	
`nextAction`	Instance	(single)	`Change`

  - Beschreibung:
    Mit dem Befehl `createSlot` werden Slots erzeugt. Das einzige benötigte Argument ist der eindeutige Slotname der in das Feld `frameName` eingetragen wird. In das Feld `valueType` kann eine Instanz der Klasse `ValueType` eingetragen werden. In der Klasse ValueType sind alle Eigenschaften eines Slots zusammengefasst die überschrieben werden können. Eine nähere Beschreibung dazu, findet befindet sich bei der Klasse `ValueType`. In das Feld `inverseSlot` wird falls vorhanden der Name des Inversenslot angeben. Außerdem hat Protégé auch ein Hierachiesystem für Slots, so können in das Feld `superSlots` (vorhandene) Superslots eingetragen werden.
  - Fehlermeldungen:
    * `MissingFrameName`
    * `MissingFrame`
    * `ClassIsAbstract`
    * `WrongDirectType`
    * `FrameHasWrongType`
    * `FrameExist`

A.2.2 Update-Aktionen

- Override Slot

- Felder:

Name	Typ	Kardinalität	Ziel
`frameName`	String	single	
`slot`	String	single	
`valueType`	Instance	single	`ValueType`
`nextAction`	Instance	(single)	`Change`

- Beschreibung:
  Mit dieser Aktion werden Eigenschaften eines bereits an eine Klasse gehängter Slot überschrieben. Der Slot muss nicht direkt an die Klasse gehängt sein, sondern kann auch an einer Oberklasse hängen. In das Feld `frameName` wird der Name der Klasse eingetragen an der die Eigenschaften des Slots überschrieben werden sollen. Das Feld `slot` ist für den Namen des Slots dessen Eigenschaften überschrieben werden sollen. Und die Instanz von `ValueType`, die die zu überschreibenden Werte enthält wird in das Feld `valueType` eingetragen. Wird nichts in das Feld `valueType` eingetragen, so werden alle direkt überschriebenen Eigenschaften des Slots an dieser Klasse gelöscht. Überschriebene Eigenschaften des Slots die durch Oberklassen überschrieben wurden, sind dadurch nicht betroffen.

- Fehlermeldungen:
  * `MissingFrameName`
  * `MissingFrame`
  * `SlotNotAttached`

- UpdateBrowserTextSlot

- Felder:

Name	Typ	Kardinalität	Ziel
`frameName`	String	single	
`browserTextSlot`	String	(single)	
`nextAction`	Instanz	(single)	`Change`

- Beschreibung:
  Ein BrowserTextSlot ist ein Slot dessen Eintrag anstelle des Namens der Instanz angezeigt wird. Die Angabe eines BrowserTextSlots ist also in erster Linie nur für die GUI von Protégé interessant und wird auch nicht in den RDF/RDFS Dateien eines Projektes gespeichert. Mit der Action `UpdateBrowserTextSlot` kann man nun einen BrowserTextSlot bestimmen, indem in das Feld `frameName` der Name der Klasse, die den BrowserTextSlot bekommen soll und in `browserTextSlot` den Namen des Slots, der als BrowserTextSlot fungieren soll, eingetragen wird. Der Slot muss an die Klasse gehängt sein, für die er BrowserTextSlot sein soll!

- Fehlermeldungen:
  * `MissingFrameName`
  * `MissingFrame`
  * `SlotNotAttached`

- UpdateDirectType

- Felder:

Name	Typ	Kardinalität	Ziel
`frameName`	String	single	
`directType`	String	single	
`nextAction`	Instance	(single)	`Change`

- Beschreibung:
  Mit der Action `UpdateDirectType` kann der Typ eines Frames verändert werden. Die zugrunde liegende Java-Klasse kann leider nicht geändert werden. Deshalb muss eine Protégé Klasse immer eine Protégé Klasse, ein Protégé Slot immer ein Protégé Slot und eine Protégé Instanz immer eine Protégé Instanz bleiben, da sie auf unterschiedliche Java-Klassen abgebildet werden. Außerdem darf der neue Typ nicht abstrakt sein. In das Feld `frameName` wird der Name des Frames eingetragen von der Typ geändert werden soll und in das Feld `directType` der Name des neuen Typs.
- Fehlermeldungen:
  * `MissingFrameName`
  * `MissingFrame`
  * `SlotNotAttached`
  * `ClassIsAbstract`
  * `FrameHasWrongType`

- UpdateDocumentation
  - Felder:

Name	Typ	Kardinalität	Ziel
`frameName`	String	single	
`documentation`	String	(multiple)	
`nextAction`	Instance	(single)	`Change`

  - Beschreibung:
    Mit dieser Action kann die Dokumentation einer Klasse, Slot, Instanz geändert werden. Dafür wird in das Feld `frameName` der Name des Frames dessen Dokumentation geändert werden soll eingetragen und in das Feld `documentation` die neue Dokumentation die zu dem Frame gespeichert werden soll. Wird nichts in das Feld `documentation` eingetragen so wird die bestehende Dokumentation gelöscht.
  - Fehlermeldungen:
    * `MissingFrameName`
    * `MissingFrame`

- UpdateInstanceName
  - Felder:

Name	Typ	Kardinalität	Ziel
`frameName`	String	single	
`newName`	String	single	
`nextAction`	Instance	(single)	`Change`

  - Beschreibung:
    Mit der Action `updateInstanceName` kann der eindeutige Name eines

Frames geändert werden. Der neue Name muss auch wieder eindeutig in der Ontologie sein. Geändert wird der Name in dem in das Feld `frameName` der alte Name und in das Feld `newName` der neue Name eingetragen wird.

  - Fehlermeldungen:
    * `MissingFrameName`
    * `MissingFrame`
    * `FrameExist`

- UpdateInverseSlot
  - Felder:

Name	Typ	Kardinalität	Ziel
`frameName`	String	single	
`inverseSlot`	String	(single)	
`nextAction`	Instance	(single)	`Change`

  - Beschreibung:
    Mit der Action `UpdateInverseSlot` können Inverse Slots geupdated werden. In das Feld `frameName` wird der Name des Slots eingetragen und in das Feld `inverseSlot` der Name des inversen Slots. Zum Löschen eines inversen Slots wird einfach nichts in das Feld `inverseSlot` eingetragen. Protégé trägt automatisch im inversen Slot den anderen Slot als invers ein.
  - Fehlermeldungen:
    * `MissingFrameName`
    * `MissingFrame`

- UpdateIsAbstractClass
  - Felder:

Name	Typ	Kardinalität	Ziel
`frameName`	String	single	
`isAbstract`	Boolean	single	
`nextAction`	Instance	(single)	`Change`

  - Beschreibung:
    Mit der Action `UpdateIsAbstractClass` kann eine Klasse als abstrakt markiert oder diese Markierung wieder entfernt werden. Dafür wird in das Feld `frameName` der Name der Klasse eingetragen und in das Feld `isAbstract` muss *true* eingetragen werden wenn die Klasse abstrakt sein soll und *false* wenn nicht.
  - Fehlermeldungen:
    * `MissingFrameName`
    * `MissingFrame`

A.2.3 Delete-Aktionen

- DeleteInstance

- Felder:

Name	Typ	Kardinalität	Ziel
`frameName`	String	single	
`nextAction`	Instance	(single)	`Change`

- Beschreibung:
  Mit der Action `DeleteInstances` können Klassen, Slots und Instanzen gelöscht werden. Dazu wird in das Feld `frameName` der Name des Frames eingetragen.
- Fehlermeldungen:
  * `MissingFrameName`
  * `MissingFrame`

- RemoveSlot
  - Felder:

Name	Typ	Kardinalität	Ziel
`frameName`	String	single	
`slot`	String	single	
`nextAction`	Instance	(single)	`Change`

  - Beschreibung:
    Die Action `RemoveSlot` ist die inverse Action zu `Attachslot`. Ein einmal attachter Slot kann damit wieder entfernt werden. Dafür wird der Name der Klasse, an die der Slot attached ist in das Feld `frameName` eingetragen und der Name des Slots in das Feld `slot`.
  - Fehlermeldungen:
    * `MissingFrameName`
    * `MissingFrame`

- RemoveSuperClasses
  - Felder:

Name	Typ	Kardinalität	Ziel
`frameName`	String	single	
`superClasses`	String	multiple	
`nextAction`	Instance	(single)	`Change`

  - Beschreibung:
    Die Aktion `RemoveSuperClasses` ist die inverse Aktion zu `AddSuperClasses`. Mit ihr können Oberklassen einer Klasse entfernt werden. Sollte am Ende einer `Change`-Kette Klassen keine Oberklasse haben, so werden diese automatisch an die Klasse *:THING* gehängt. In das Feld `frameName` wird der Name der Klasse eingetragen und in das Feld `superClasses` die Namen der Oberklassen, die entfernt werden sollen.
  - Fehlermeldungen:
    * `MissingFrameName`
    * `MissingFrame`

- RemoveValues

- Felder:

Name	Typ	Kardinalität	Ziel
`frameName`	String	single	
`slot`	String	single	
`values`	String	multiple	
`vType`	Symbol	(single)	
`nextAction`	Instance	(single)	`Change`

- Beschreibung:
  Die Aktion `RemoveValues` ist die inverse Aktion zu `AddValues`. Mit dieser Aktion können Werte gelöscht werden. In das Feld `frameName` kommt der Name der Instanz. Das Feld `slot` nimmt den Namen des Slots auf und in das Feld `values` werden die Werte eingetragen. Bei Werten vom Typ *Instance*, *Slot* oder *Class* müssen die Namen der Frames eingetragen werden. Ist der Slot vom Typ *ANY*, so muss in das Feld `vType` der Typ der Werte eingetragen werden. Sollen Werte verschiedenen Typs von einen Any-wertigen Slot gelöscht werden, so muss die Aktion in mehrere *RemoveValues*-Aktionen auf gespaltet werden.

- Fehlermeldungen:
  * `MissingFrameName`
  * `MissingFrame`
  * `SlotNotAttached`

A.3 Query-Aktionen

- ExistsFrame
  - Felder:

Name	Typ	Kardinalität	Ziel
`answer`	Boolean	(single)	
`frameName`	String	single	
`nextAction`	Instance	(single)	`Query`

  - Beschreibung:
    Die Aktion `ExistsFrame` testet ob ein Frame, mit dem Namen der in dem Feld `frameName` eingetragen ist, existiert. Ist dies der Fall so wird in das Feld Answer *true* eingetragen, ansonsten *false*.
  - Fehlermeldungen:
    * `MissingFrameName`

- GetClses
  - Felder:

Name	Typ	Kardinalität	Ziel
`answer`	Instance	(multiple)	`Class`
`nextAction`	Instance	(single)	`Query`

  - Beschreibung:
    Die Aktion `GetClses` sucht alle Klassen in der Zielontologie und trägt die Namen der Klassen in das Feld `answer` ein.

- Fehlermeldungen:
  - Keine.

- GetDirectInstancesOfClass
  - Felder:

Name	Typ	Kardinalität	Ziel
`answer`	Instance	(multiple)	`Class, Instance`
`frameName`	String	single	
`nextAction`	Instance	(single)	`query`

  - Beschreibung:
    Die Aktion `GetDirectInstancesOfClass` sucht alle direkten Instanzen einer Klasse. Der Name der Klasse, von der die direkten Instanzen gesucht sind, wird in das Feld `frameName` eingetragen. Das RDF-Tab trägt in das Feld `answer` die Namen aller Instanzen der angegebenen Klasse ein.
  - Fehlermeldungen:
    - `MissingFrameName`
    - `MissingFrame`

- GetDirectSubClses
  - Felder:

Name	Typ	Kardinalität	Ziel
`answer`	Instance	(multiple)	`Class`
`frameName`	String	single	
`nextAction`	Instance	(single)	`Query`

  - Beschreibung:
    Die Aktion `GetDirectSubClses` sucht die direkten Unterklassen einer Klasse. Der Name der Klasse, von der die direkten Unterklassen gesucht sind, muss in das Feld `frameName` eingetragen werden. Die Namen der gefundenen Unterklassen werden vom RDF-Tab in das Feld `answer` eingetragen.
  - Fehlermeldungen:
    - `MissingFrameName`
    - `MissingFrame`

- GetDirectSuperClses
  - Felder:

Name	Typ	Kardinalität	Ziel
`answer`	Instance	(multiple)	`Class`
`frameName`	String	single	
`nextAction`	Instance	(single)	`Query`

  - Beschreibung:
    Die Aktion `GetDirectSuperClses` sucht die direkten Oberklassen einer Klasse. Der Name der Klasse, von der die direkten Oberklassen gesucht sind, muss in das Feld `frameName` eingetragen werden. In das Feld `answer` werden die Namen der vom RDF-Tab gefundenen Oberklassen eingetragen.

  - Fehlermeldungen:
    * `MissingFrameName`
    * `MissingFrame`

- GetDirectType
  - Felder:

Name	Typ	Kardinalität	Ziel
`answer`	Instance	(single)	`Class`
`frameName`	String	single	
`nextAction`	Instance	(single)	`Query`

  - Beschreibung:
    Die Aktion `GetDirectType` ermittelt den Typ eines Frames, also von welcher Klasse der Frame Instanz ist. Der Name des Frames, dessen Typ ermittelt werden soll, kommt in das Feld `frameName`. Der Name des Typs wird von dem RDF-Tab dann später in das Feld `answer` eingetragen.
  - Fehlermeldungen:
    * `MissingFrameName`
    * `MissingFrame`

- GetInstances
  - Felder:

Name	Typ	Kardinalität	Ziel
`answer`	Instance	(multiple)	`Class, Slot, Instance`
`nextAction`	Instance	(single)	`Query`

  - Beschreibung:
    Die Aktion `GetInstances` liefert alle Instanzen zurück. Wie schon öfter erwähnt sind auch Klassen und Slots in Protégé Instanzen so genannter Metaklassen und werden deshalb auch alle zurückgeliefert. Das RDF-Tab trägt die Namen aller Instanzen in das Feld `answer` ein.
  - Fehlermeldungen:
    * `MissingFrameName`
    * `MissingFrame`

- GetOwnSlots
  - Felder:

Name	Typ	Kardinalität	Ziel
`answer`	Instance	(multiple)	`Slot`
`frameName`	String	single	
`nextAction`	Instance	(single)	`Query`

  - Beschreibung:
    Die Aktion `GetOwnSlots` ermittelt alle Slots die direkt an einer Klasse hängen. Der Name der Klasse muss in das Feld `frameName` eingetragen werden. Das RDF-Tab trägt dann in das Feld `answer` die Namen der gefundenen Slots.

- Fehlermeldungen:
  - `MissingFrameName`
  - `MissingFrame`

- GetPartialOntology
  - Felder:

Name	Typ	Kardinalität	Ziel
`answer`	Instance	(single)	`Startchange`
`classes`	String	multiple	
`createInstances`	Boolean	single	
`includeSubclasses`	Boolean	single	
`modus`	Integer	single	
`returnRDF`	Boolean	single	
`transHuelle`	Boolean	single	
`nextAction`	Instance	(single)	`Query`

  - Beschreibung:
    Mit der Aktion `GetPartialOntology` können Teile oder auch komplette Ontologien übertragen werden. `GetPartialOntology` erzeugt eine Kette von `Change` Anweisungen, die wieder einem RDF-Tab übergeben, die Teilontologie aufbauen. In das Feld `classes` werden die Namen der erwünschten Klassen geschrieben. Die Felder `createSubclasses`, `createInstances` und `modus` geben an, ob Unterklassen, Instanzen und Oberklassen auch mit einbezogen werden sollen. Ist die Option `transHuelle` aktiviert, so wird die resultierende Ontologie abgeschlossen sein, d.h. sie wird alle Klassen, Slots und Instanzen enthalten auf die verwiesen wird. Von abgeschlossenen Ontologien kann auch ein RDF Darstellung zusätzlich erzeugt werden, wenn die Option `returnRDF` aktiviert ist. In das Feld `answer` wird vom RDF-Tab die `StartChange` Instanz der `Change`-Kette geschrieben.
  - Fehlermeldungen:
    - `MissingFrameName`
    - `MissingFrame`

- GetSlotValues
  - Felder:

Name	Typ	Kardinalität	Ziel
`answer`	ANY	(multiple)	
`frameName`	String	single	
`slot`	String	single	
`nextAction`	Instance	(single)	`Query`

  - Beschreibung:
    Die Aktion `GetSlotValues` ist zum Auslesen von Werten einer Instanz. In das Feld `frameName` kommt der Name der Instanz. Da Werte immer in Slots gespeichert werden, muss in das Feld `slot` der Name des Slots eingetragen werden. Das RDF-Tab trägt in das Feld `answer` die gefundenen Werte ein.
  - Fehlermeldungen:

    * `MissingFrameName`
    * `MissingFrame`
    * `SlotNotAttached`

- GetSlots
  - Felder:

Name	Typ	Kardinalität	Ziel
`answer`	Instance	multiple	`Slot`
`nextAction`	Instance	(single)	`Query`

  - Beschreibung:
    Die Aktion `GetSlots` sucht alle Slots der Ontologie. Die Namen der Slots werden von dem RDF-Tab in das Feld `answer` eingetragen.
  - Fehlermeldungen:
    * Keine.

- GetTemplateSlots
  - Felder:

Name	Typ	Kardinalität	Ziel
`answer`	Instance	multiple	`Slot`
`frameName`	String	single	
`nextAction`	Instance	(single)	`Query`

  - Beschreibung:
    Die Aktion `GetTemplateSlots` liefert alle Slots die an einer Klasse hängen. In das Feld `frameName` wird der Name der Klasse, deren Slots erfragt werden sollen, eingetragen. Das RDF-Tab schreibt dann in das Feld `answer` die Namen der Slots.
  - Fehlermeldungen:
    * `MissingFrameName`
    * `MissingFrame`

- PalQuery
  - Felder:

Name	Typ	Kardinalität	Ziel
`answer`	Instance	(multiple)	`VariableValueBinding`
`statement`	String	single	
`range`	String	(single)	
`nextAction`	Instance	(single)	`Query`

  - Beschreibung:
    Mit der Aktion `PalQuery` können PAL-Anfragen gestellt werden. Mit PAL-Anfragen werden Instanzen von Klassen die bestimmte Eigenschaften haben gesucht. Dies geschieht in Protégé indem eine Instanz der Klasse *PAL-QUERY* angelegt wird und diese Instanz dem PAL-QueryTab als Argument übergeben wird. Die Instanz besitzt die Felder: *:PAL-NAME*, *:PAL-DESCRIPTION*, *:PAL-RANGE* und *:PAL-STATEMENT*. Das Feld *:PAL-NAME* soll die Anfrage eindeutig identifizieren, in der Antwort wird als Bezug der Name der Anfrage angegeben. Das Feld *:PAL-DESCRIPTION* ist optional für eine

genauere Beschreibung der Funktion der Anfrage. In der Feld *:PAL-RANGE* wird der Bereich in dem gesucht werden soll angegeben und in das Feld *:PAL-STATEMENT* die Suchbedingungen [1] .

Die Aktion `PalQuery` benötigt nur die Angabe des PAL-Statement und der PAL-Range. Die Instanz der Klasse *PAL-QUERY* wird vom RDF-Tab erstellt und ein eindeutiger Name ausgesucht. Da die Instanz nach der Abarbeitung der Aktion vom RDF-Tab gelöscht wird ist die Angabe einer Dokumentation nicht nötig. Das PAL-Statement kommt in das Feld `statement` und die PAL-Range in das Feld `range`. Es findet keine Überprüfung auf (syntaktische) Korrektheit statt. In das Feld `answer` wird bei einer erfolgreichen Suche Instanzen der Klasse `VariableValueBinding` eingetragen. Diese Klasse ist eine Nachbildung der Java-Klasse *VariableValueBinding*, die die PAL-QueryEngine auf Anfragen zurückgeliefert. Die Klasse `VariableValueBinding` besitzt zwei Felder, das Feld `name` in dem der Name der Variable/Feldes angegeben wird, der in Range spezifiziert wurde und das Feld `variableValue` in dem die gefundenen Instanz steht. Sollte die PAL-QueryEngine Fehler melden, so werden diese in der Fehlerklasse `PALError` zurückgeliefert. Zur Zeit wird auch bei einem `PALError` die Verarbeitung abgebrochen, obwohl das nicht nötig wäre.

Um PAL-Anfragen stellen zu können muss auf jedenfall in der Zielontologie das Projekt *pal_query.pprj* includet sein. Dieses ist im Protégé Verzeichnis im Unterverzeichnis *projects/pal_query* zu finden.

– Fehlermeldungen:

  * `PALError`

A.4 Utility Klassen

- ValueType

  – Felder:

Name	Typ	Kardinalität
`allowedClasses`	String	(multiple)
`allowedParents`	String	(multiple)
`allowedValues`	String	(multiple)
`defaultValues`	String	(multiple)
`documentation`	String	(multiple)
`maximumCardinality`	Integer	(single)
`minimumCardinality`	Integer	(single)
`maximumIntValue`	Integer	(single)
`minimumIntValue`	Integer	(single)
`maximumFloatValue`	Float	(single)
`minimumFloatValue`	Float	(single)
`vType`	Symbol	(single)

[1] Eine ausführlichere Beschreibung der Formatierung der Bereichs- und Suchbedingung bei PAL-Querys kann unter folgender URL gefunden werden:
`http://www.smi.stanford.edu/projects/protege/mail_archive/msg00549.html`

– Beschreibung:
Die Klasse `ValueType` fasst alle Eigenschaften der Slots bezüglich den in ihnen speicherbaren Werte zusammen. Verwendet wird die Klasse `ValueType` deshalb in den Aktionen `CreateSlot` und `OverrideSlot` die diese Eigenschaften ändern. Es gibt zwei Gruppen von Eigenschaften, einmal die Eigenschaften die für alle Typen von Werten gelten und anderseits die Eigenschaften die nur dann gelten wenn ein bestimmter Typ von Wert in dem Slot gespeichert wird. Zur ersten Gruppe gehören `vType` der angibt welcher Typ von Wert in dem Slot gespeichert wird, `maximumCardinality` und `minimumCardinality` die die Kardinalität der Werte verwalten, `documentation` das für Dokumentation, über den Slot und die zu speichernden Werte, da ist und `defaultValues` das default Werte für den Slot speichert. Zur zweiten Gruppe gehören `allowedParents`, das nur Sinn macht wenn der Typ der Werte *Class* ist und angibt welche Oberklasse eine, in so einem Slot gespeicherte, Klasse haben muss, `allowedClasses`, das nur Verwendung findet wenn der `vType Instances` ist und angibt welche Oberklasse der Typ einer gespeicherten Instanz haben muss, `allowedValues`, der wenn der `vType` *Symbol* ist die möglichen Symbols angibt, `maximumIntValue` und `minimumIntValue`, die wenn der `vType` *Integer* ist die maximal und minimal möglichen Werte aufnehmen und `maximumFloatValue` und `minimumFloatValue`, die die gleiche Funktionalität, wie `maximumIntValue` und `minimumIntValue` für den Typ *Integer* zur Verfügung stellen, für Werte des Typs *Float* bereitstellen.

A.5 Ok-, Fehler- und Warnungs-Messages

A.5.1 Ok Message

- Felder:

Name	Typ	Kardinalität
`completed_chains`	String	multiple

- Beschreibung:
Die Antwort auf eine Anfrage enthaelt immer dann eine Instanz dieser Klasse wenn mindestens eine Abfragekette der Anfrage bei der Verarbeitung keinen Fehler erzeugte. Die Namen der Startinstanzen der fehlerlosen Abfrageketten werden in das Feld `completed_chains` eingetragen.

- Ideen:

A.5.2 Fehler beim Parsen der Anfrage

In dieser Klasse sind alle Fehler zusammengefasst, die beim Parsen der RDF-Message auftreten können. Alle Fehler dieser Klasse haben nur das Feld `error`. In diesem Feld wird eine menschenlesbare Form der Fehlermeldung abgelegt. Die Verarbeitung der RDF-Message wird direkt abgebrochen und es wird nichts an der Zielontologie geändert.

- ChangeAndQueryInstances

  - Meldung:
    Change- and Querystartinstances mismatch!
  - Beschreibung:
    Der Fehler tritt auf wenn in einer RDF-Message gleichzeitig Change- und Querystartinstances sind.

- NoStartInstance
  - Meldung:
    No Change- or Querystartinstances!
  - Beschreibung:
    Der Fehler tritt auf wenn in einer RDF-Message keine `Start`-Instance vorhanden ist.

- RDFError
  - Meldung:
    Die Fehlermeldungen des RDF-Parsers von Protégé werden direkt in das Feld kopiert.
  - Beschreibung:
    Alle Fehler des RDF-Parsers von Protégé werden in diese Klasse verpackt.

A.5.3 Fehler der Elementaroperationen

Alle `InstancesErrors` haben die Felder `error`, `instance` und `chainName` gemeinsam. In dem Feld `error` ist eine menschenlesbare Form des Fehlers gespeichert, das Feld `instance` speichert den Namen der Instanz, in der der Fehler auftrat und das Feld `chainName` enthält den Namen der `Start`-Instanz der fehlerhaften Aktion, so das einfach ersichtlich ist, in welcher Kette der Fehler auftrat.

- CantIdentifyType
  - Felder:

Name	Typ	Kardinalität
`error`	String	multiple
`instance`	String	single
`chainName`	String	single
`frameName`	String	single
`instance`	String	single

  - Meldung:
    Can't identify Type of Instance: `frameName`.
  - Beschreibung:
    Diese Fehlermeldung tritt nur in dem speziellen Fall auf, das die Option `createMissingFrames` aktiviert ist und versucht wird ein fehlendes Frame zu erzeugen, von dem der Typ nicht ermittelt werden kann. Dies ist zur Zeit der Fall, wenn an einen Slot vom Typ *Instance*, der keinerlei Einschränkung bei den erlaubten Parents hat, eine Instanz gehängt werden soll, die zu dem Zeitpunkt noch nicht erzeugt

ist. Denn dann kann zu diesem Zeitpunkt nicht entschieden werden, ob eine Instanz vom Typ Klasse, Slot oder Instanz erzeugt werden muss. In das Feld `frameName` wird der Name der Instanz, von der der Typ nicht bestimmt werden konnte, eingetragen.

- ClassIsAbstract
  - Felder:

Name	Typ	Kardinalität
`error`	String	multiple
`instance`	String	single
`chainName`	String	single
`frameName`	String	single
`directType`	String	single

  - Meldung:
    Class `directType` is Abstract! Couldn't create Instance: `frameName`.
  - Beschreibung:
    Der Fehler kommt, wenn versucht wird von einer als abstrakt definierten Klasse eine Instanz zu erzeugen. In das Feld `frameName` wird der Name der Instanz und in das Feld `directType` der Name der Klasse, von der die Instanz erzeugt werden soll, eingetragen.

- FrameExist
  - Felder:

Name	Typ	Kardinalität
`error`	String	multiple
`instance`	String	single
`chainName`	String	single
`frameName`	String	single
`directType`	String	single

  - Meldung:
    Frame does already exists: `frameName`
  - Beschreibung:
    Der Fehler wird erzeugt wenn versucht wurde ein Frame mit einem bereits vorhandenen Namen zu erzeugen.

- FrameHasWrongType
  - Felder:

Name	Typ	Kardinalität
`error`	String	multiple
`instance`	String	single
`chainName`	String	single
`frameName`	String	single
`frameDirectType`	String	(single)
`directType`	String	(single)

  - Meldung:
    Frame: `frameName` ist not of Type: `directType`.

  - Beschreibung:
    Dieser Fehler tritt dann auf, wenn das übergebene Frame nicht den Typ hat den die Aktion erwartet. Wenn z. B. der Aktion `createSlot` im Feld `inverseSlot` kein Slot, sondern eine Klasse übergeben wird, so wird dieser Fehler erzeugt. In das Feld `frameName` wird der Name des Frames eingetragen. Das Feld `frameDirectType` enthält den Namen des Typ des Frames und das Feld `directType` den Namen der Oberklasse, die der Typ des Frames eigentlich besitzen müßte.

- MissingFrame
  - Felder:

Name	Typ	Kardinalität
`error`	String	multiple
`instance`	String	single
`chainName`	String	single
`frameName`	String	single

  - Meldung:
    Frame: `frameName` does not exist!
  - Beschreibung:
    Existiert kein Frame mit dem angegebenen Namen, so wird dieser Fehler erzeugt. In das Feld `frameName` kommt der Name des Frames, das nicht gefunden werden konnte.

- MultipleValuesOnValueSlot
  - Felder:

Name	Typ	Kardinalität
`error`	String	multiple
`instance`	String	single
`chainName`	String	single
`frameName`	String	single
`slot`	String	single
`values`	String	multiple

  - Meldung:
    Multiple Values: `values` on one Value Slot: `slotName` at Instance: `frameName`
  - Beschreibung:
    Dieser Fehler wird nur von der Aktion `AddValues` erzeugt. Und zwar dann, wenn versucht wird an einen Slot, der nur einen Wert speichern kann, mehrere Werte zu speichern. Der Name der Instanz wird in das Feld `frameName` geschrieben. Der Name des Slots in das Feld `slot` und eine Stringrepräsentation der Werte in das Feld `values`.

- PALError
  - Felder:

Name	Typ	Kardinalität
`error`	String	multiple
`instance`	String	single
`chainName`	String	single

  - Meldung:
    Die Fehlermeldungen des PAL-Plugins werden direkt in das Feld `error` eingetragen.
  - Beschreibung:
    Dieser Fehler wird nur von der Aktion `PalQuery` erzeugt. Alle Fehler, die die PAL-QueryEngine meldet, werden dieser Klasse übergeben.

- SlotNotAttached
  - Felder:

Name	Typ	Kardinalität
`error`	String	multiple
`instance`	String	single
`chainName`	String	single
`frameName`	String	single
`slot`	String	single

  - Meldung:
    Instance: `frameName` has not Slot: `slotName`.
  - Beschreibung:
    Wenn eine Aktion abgearbeitet wird, bei der erwartet wird das der Slot `slot` an die Instanz `frameName` gebunden ist (z.B. `AddValues`), dies aber nicht der Fall ist, wird dieser Fehler erzeugt.

- WrongDirectType
  - Felder:

Name	Typ	Kardinalität
`error`	String	multiple
`instance`	String	single
`chainName`	String	single
`frameName`	String	single
`requiredDirectType`	String	single

  - Meldung:
    DirectType: `directTypeName` has not Superclass: `requiredDirectType`!
  - Beschreibung:
    Dieser Fehler wird nur von den Aktionen `CreateClass` und `CreateSlot` erzeugt. Der Fehler besagt, das der übergebene Typ keine Metaklasse beziehungsweise Metaslot ist. In das Feld `frameName` wird der Name des übergebenen Typ eingetragen und in das Feld `requiredDirectType` die Art des benötigten Typ eingetragen.

- SlotAlreadyAttached
  - Felder:

Name	Typ	Kardinalität
`error`	String	multiple
`instance`	String	single
`chainName`	String	single
`frameName`	String	single
`slot`	String	single

  - Meldung:
    Slot: `slot` already attached at Class: `frameName`.
  - Beschreibung:
    Der Fehler kommt, wenn die Aktion `AttachSlot` einen Slot an eine Klasse attachen soll, an die der Slot schon attached ist. Der Name der Klasse wird im Feld `framename` übergeben und der Name des Slots im Feld `slot`.

- SlotAlreadyAttached
  - Felder:

Name	Typ	Kardinalität
`error`	String	multiple
`instance`	String	single
`chainName`	String	single
`frameName`	String	single
`superclass`	String	single

  - Meldung:
    Class: `frameName` already has Superclass: `superclass`.
  - Beschreibung:
    Der Fehler tritt dann auf, wenn eine Aktion versucht einer Klasse eine Oberklasse hinzuzufügen, die diese bereits besitzt. Der Name der Klasse wird im Feld `framename` übergeben und der Name der Oberklasse im Feld `superclass`.

A.5.4 Fehler im RDF-Code

RDF-Fehler sind Fehler im RDF-Code der Message, die nicht vom RDF-Parser erkannt werden.

- MissingFrameName
  - Felder:

Name	Typ	Kardinalität
`error`	String	multiple
`instance`	String	single
`chainName`	String	single
`missingFrameName`	String	single

  - Meldung:
    `missingFrameName` is null in `instance`. Although required!
  - Beschreibung:
    Dieser Fehler kommt immer dann, wenn kein Wert in ein Feld, das als *required* markiert ist, eingetragen ist. In das Feld `missingFrameName` wird dabei der Name des Slots eingetragen der keinen Wert enthält.

- UnknownNextInstance
  - Felder:

Name	Typ	Kardinalität
`error`	String	multiple
`instance`	String	single
`chainName`	String	single
`frameName`	String	single

  - Meldung:
    Don't know type of Instance: `instance`.
  - Beschreibung:
    Dieser Fehler bedeutet, das in das Feld `nextInstance` eine unbekannte Instanz eingetragen worden ist. Mit unbekannter Instanz ist jede Instanz gemeint, deren Typ keine Unterklasse der Klasse `Action` in der Sprachdefinition der RDF-Message ist. In das Feld `frameName` wird der Name der Instanz geschrieben.

- WrongNextInstance
  - Felder:

Name	Typ	Kardinalität
`error`	String	multiple
`instance`	String	single
`chainName`	String	single
`type`	String	single

  - Meldung:
    Instance: `instance` must be of Type `type`.
  - Beschreibung:
    Dieser Fehler tritt auf wenn in einer `Change`-Kette eine `Query`-Aktion vorkommt oder umgekehrt in einer `Query`-Kette eine `Change`-Aktion. In das Feld `type` wird der Typ der aktuell bearbeiteten Aktionskette eingetragen.

A.5.5 Warnungen

Warnungen werden nur ausgegeben wenn die Option `warnings` aktiviert ist.

- ClassHasNoSuperClass
  - Felder:

Name	Typ	Kardinalität
`warning`	String	multiple
`classes`	String	multiple

  - Meldung:
    Classes: `classes` have no Superclass(es)!
  - Beschreibung:
    Die Warnung wird ausgegeben wenn nach Abarbeitung einer `Change`-Kette mindestens eine Klasse keine Oberklasse hat. Die Namen der Klassen ohne Oberklasse werden in das Feld `classes` eingetragen. Ausserdem werden die Klassen automatisch an die Klasse `:THING` gehängt.