Von objektorientierter Programmierung über das Metaobjekt Protokoll zur Repräsentation von Konfigurierungswissen - Ein weiterer Schritt -*

Lothar Hotz

Universität Hamburg
Fachbereich Informatik
Bodenstedtstr. 16
22765 Hamburg
e-mail: hotz@fbihh.informatik.uni-hamburg.de

Kurzfassung

Wir zeigen auf, wie das Metaobjekt Protokoll der objektorientierten Sprache Common Lisp Object System (CLOS) verwendet werden kann, um CLOS zu erweiteren. Dazu untersuchen wir, welche Anforderungen durch eine Wissensrepräsentation (in unserem Fall speziell Konfigurierungswissen) an eine Implementationssprache gestellt werden. Eine solche Anforderung bildet die Erweiterung von Konzeptbeschreibungen während des Zugriffs auf Slots (sie ist bei uns u.a. für die Anbindung eines Constraintsystems sinnvoll). Eine andere liegt in der Versionsverwaltung mehrerer Eigenschaftswerte vor (sie ist für Backtracking-mechanismen einsetzbar). Diese Sprachmerkmale sind zunächst nicht in CLOS enthalten. Wir zeigen wie diese Grenze der Sprache mittels des Metaobjekt Protokolls überschritten werden kann.

1 Einleitung

In diesem Beitrag beschreiben wir, wie die objektorientierte Programmiersprache Common Lisp Object System (CLOS) [8] erweitert werden kann, um mit ihrer Hilfe eine Sprache für die Repräsentation von Konfigurierungswissen zu implementieren. Eine solche Sprache ist speziell durch die Darstellung von Konzepten (von uns für die Konfigurierung auch "Domänenobjekte" genannt), ihren Eigenschaften und ihren Beziehungen untereinander geprägt. Neben solchem Konzeptwissen sind Constraints zur Realisierung von Randbedingungen und Kontrollwissen zur Steuerung des Konfigurierungsprozesses in der Konfigurierung notwendig. Wir konzentrieren uns hier auf das Konzeptwissen und auf einen Ausschnitt der Anbindung von Constraints.

*Das diesem Bericht zugrundeliegende Forschungsvorhaben wird mit Mitteln des Bundesministers für Forschung und Technologie (Förderkennzeichen ITW9101A6, Verbundvorhaben PROKON) gefördert. Die Verantwortung für den Inhalt dieser Veröffentlichung liegt bei dem Autor.

Zwei wesentliche Relationen bilden die Spezialisierungshierarchie und die Zerlegungshierarchie. In einer Spezialisierungshierarchie (*is-a*-Relation) muß jedes Unterkonzept die Eigenschaften seines Oberkonzepts einschränken, indem die Wertebereiche verkleinert oder beibehalten werden. Eine Zerlegungshierarchie beschreibt, in welche Teilkonzepte ein Konzept zerlegt werden kann und damit auch aus welchen Teilkomponenten eine reale Komponente besteht (*has-parts*-Relation). Wir gehen näher auf die Anforderungen ein, die von einer Wissensrepräsentationssprache im Zusammenhang mit der Verfolgung mehrerer Lösungspfade und mit der Erweiterung der Eigenschaftsstruktur (Slotschema) von Konzepten an die Implementationssprache gestellt werden. Mehrere Lösungspfade werden durch die Einführung einer Versionsverwaltung von Slotwerten realisiert, die Erweiterung von Konzepten steht im Zusammenhang mit der Repräsentation von Constraints, die eine Redefinition von Konzepten während des Modellierungsprozesses nötig macht.

In objektorientierten Programmiersprachen (OOP) ist es zwar möglich, Klassenhierarchien zu erstellen, grundsätzlich ist jedoch eine Trennung zwischen dieser mehr datenorientierten Ebene und der Ebene der Wissensrepräsentation zu ziehen [2]. Diese Unterscheidung ist sinnvoll, da Datenobjekte andere Anforderungen genügen müssen als Wissensobjekte. Durch Einführung von verschiedenen Ebenen kann die unterschiedliche Funktionalität separat beschrieben werden. Obiges Beispiel der einschränkenden Vererbung von Eigenschaften illustriert bereits, daß auf der Wissensebene andere Vererbungstechniken zum Einsatz kommen als in OOP üblich.

Eine Möglichkeit für die Implementation der Wissensebene bildet die Anpassung einer OOP auf die Anforderungen dieser Ebene. Diese Anpassung kann dann möglich sein, wenn die Implementation der OOP verändert werden kann, z.B. indem der vorgegebene Vererbungsmechanismus geändert wird. (Dies zu prüfen, ist Gegenstand unserer aktuellen Arbeit im Projekt PROKON [7], in dessen Zusammenhang das konfigurierungsspezifische Werkzeug KONWERK mittels CLOS implementiert werden soll.) CLOS ist mit Hilfe des Metaobjekt Protokolls (MOP) [9] implementiert. Mit diesem Protokoll ist eine Öffnung von CLOS gegeben, die es erlaubt, das Verhalten von CLOS (z.B. bei einem Slotzugriff) gegebenen Wünschen anzupassen und portabel zu ändern. Im folgenden beschreiben wir, wie eine dynamische Erweiterung von Konzepten und Instanzen bei Zugriffen auf vorher nicht definierte Eigenschaften erfolgen kann. Ist ein Metaobjekt Protokoll nicht vorhanden (wie etwa in C++ oder Smalltalk), muß diese Funktionalität, die durch das MOP Teil der Sprache wird, simuliert werden.

Eine andere Alternative der Implementation von Wissensrepäsentationssprachen besteht darin, die Konstrukte der Wissensebene nicht direkt auf Konstrukte einer OOP abzubilden, d.h. Konzepte, Vererbungsmechanismen, Verhaltensbeschreibungen von Instanzen (Methoden) etc. nicht auf die analogen Konstrukte einer OOP abzubilden, sondern diese lediglich zur Unterstützung der Implementation zu verwenden. Bei diesem Ansatz muß die ähnliche Funktionalität von z.B. Klassen- und Konzepthierarchien parallel implementiert werden.

Statt die Spezifikation eines neuen Protokolls (d.h. von Klassen und generische Funktionen und ihren Zusammenhängen) für die Implementation einer Wissensrepräsentationssprache einzuführen, erweitern wir das genau spezifizierte Protokoll von CLOS. Speziell die Verständlichkeit und Wartbarkeit wird so erleichtert, da statt eines vollständigen neuen Protokolls ein bekanntes erweitert wird. Für unsere Entwicklung von KONWERK erscheint eine Anpassung einer OOP an die genannten Anforderungen eine mögliche,

schneller zu realisierende, genauer spezifizierte und wartbarere (da übersichtlichere) Alternative zu sein.

Wir wollen im folgenden zunächst die notwendigen Elemente und Methoden einer Wissensrepräsentation für die Konfigurierung formulieren und die sich daraus ergebenden Anforderungen an eine OOP ableiten (Abschnitt 2). Danach beschreiben wir kurz das MOP (Abschnitt 3). In Abschnitt 4 wird die Realisierung der Verwaltungen von Domänenobjekten und Eigenschaften, wie sie im System KONWERK mit Hilfe des MOP erfolgt, vorgestellt. In Abschnitt 5 diskutieren wir kurz den vorgestellten Ansatz.

2 Anforderungen der Wissensrepräsentation an eine Basissprache

Im folgenden listen wir Elemente und Methoden einer Wissensrepräsentationssprache sowie die daraus abzuleitenden Anforderungen an eine objektorientierte Implementationssprache auf (vgl. auch [2]):

a) Domänenobjekte, Eigenschaften und Werte einer Wissensrepräsentationssprache können grob mit Klassen, Slots und Slotwerten einer OOP identifiziert werden.

b) Um zusätzliche Informationen über Werte eines Slots (etwa Evidenzwerte, Wertbestimmungsfunktionen, erlaubte Wertebereiche, Defaultwerte) zu repräsentieren, müssen Facetten einem Slot zugeordnet werden können.

c) Zerlegungshierarchien (*has-parts, part-of*) werden über Relationen zwischen Objekten dargestellt. Dabei werden Relationen als spezialisierbare Klassen repräsentiert, um Erweiterungen zu ermöglichen. Diese Klassen sorgen z.B. für die gleichzeitige Erzeugung einer inversen Relation (falls möglich) und die Verwaltung der Verweise.

d) Um die Konsistenz von Slotwerten zu gewährleisten, müssen Tests beim Eintragen von Werten in Slots ermöglicht werden ("procedural attachment").

e) Um verschiedene Pfade eines Suchbaums darzustellen (z.B. für Backtracking), sind Versionsverwaltungen von Slotwerten und Objekten gefordert.

f) Die Erweiterbarkeit von Domänenobjekten während der Systementwicklung und für die Definition von Constraints erfordert die dynamische Erweiterung von Klassen, d.h. das Hinzufügen von Slots zu Instanzen oder Klassen während des Zugriffs auf einen unbekannten Slot.

Wir betrachten die letzten beiden Punkte genauer. Im Konfigurierungssystem KONWERK sind Versionsverwaltungsmechanismen für die (quasi) parallele Verwaltung von mehreren Eigenschaftswerten von Eigenschaften notwendig. Die Konfigurierung erfolgt schrittweise, durch aufeinanderfolgende Teilkonfigurationen (TK), die eine immer spezieller werdende Konfiguration beschreiben (vgl. auch [6]). In jeder TK werden ein oder mehrere Eigenschaftswerte festgelegt oder Wertebreiche eingeschränkt. Technisch gesprochen haben die zugehörigen Slots in jeder TK einen anderen Wert. Es werden also keine

Kopien von Instanzen pro TK erzeugt, sondern aus Effizienzgründen nur die Änderungen von TK zu TK gespeichert. Da es möglich sein soll, während der Konfigurierung zwischen diesen TKs zu wechseln (z.B. wenn eine TK inkonsistent wird, d.h. eingetragene Werte widersprechen sich, wird gemäß unterschiedlicher Backtracking-Mechanismen eine vorhergehende TK ausgewählt), müssen alle Eigenschaftswerte pro TK verwaltet werden. Diese Verwaltung wird realisiert, indem jede TK einen *Slotkontext* (auch *viewpoint*) bildet (vgl. [3]). Während des Zugriffs auf einen Slot wird der aktuelle Kontext berücksichtigt und der passende Wert geholt (vgl. Abschnitt 4). Statt eines Slotwertes wird eine Liste von Werten und Kontexten verwaltet. Neben den gerade genannten Backtracking-Verfahren sind speziellere Versionsverwaltungen für komplexere Varianten von KONWERK (etwa mit einem ATMS [4]) ähnlich einzugliedern.

Der Punkt der Erweiterbarkeit von Slotstrukturen bietet Entwicklern anderer Module, etwa für die Verwaltung von Constraints oder zur Steuerung des Konfigurierungsprozesses, die Möglichkeit, zusätzliche Slots Konzepten hinzuzufügen. Ein Beispiel bietet die Anbindung von Constraints. Randbedingungen oder Abhängigkeiten, wie etwa die Bedingung, daß der Speicher eines zu konfigurierenden PC's der Summe der Größen der Speicherblöcke entspricht, werden in KONWERK mit "konzeptuellen Constraints" dargestellt. Im Beispiel wird das Konzept "Speicher" mit dem Konzept "Speicherblock" verbunden, daher "konzeptuelle" Constraints. Während der Konfigurierungsphase werden die konzeptuellen Constraints auf spezielle Wissensinstanzen (etwa *Speicher-1, Speicherblock-37* etc.) angewandt. Es werden Constraintinstanzen, bestehend aus Wissensinstanzen und einem Vermerk der Abhängigkeit (im Beispiel bezüglich der Speichergröße), erzeugt. Aufgrund der Vielzahl von konzeptuellen Constraints und Wissensinstanzen entsteht ein Constraintnetz. Dabei werden Verwaltungsinformationen, die beschreiben, in welcher Constraintinstanz eine Wissensinstanz vorkommt, nötig. Diese werden in einem Slot der Wissensinstanz gespeichert, welches dynamisch dem Konzept hinzugeführt wird. Während der Definition von konzeptuellen Constraints (d.h. nach der Konzeptdefinition) wird daher den in konzeptuellen Constraints vorkommenden Konzepten ein neuer Slot hinzugefügt. Da nicht jedes Konzept in einem Constraint vorkommt, wird der Slot nicht in einer Oberklasse eingeführt, sondern dynamisch ergänzt.

Die Redefinierbarkeit bietet während des Entwicklungsprozesses von KONWERK weiterhin die Möglichkeit, kurzzeitig neue Slots Konzepten hinzuzufügen. Dies ist sinnvoll, da sich herausgestellt hat, daß eine Slotstruktur nicht von vornherein fest spezifiziert werden kann, sondern sich während der Entwicklung eines objektorientierten Programms ändert. Hat sie sich zum Ende des Entwicklungsprozesses gefestigt, können die notwendig gewordenen Slots einer Oberklasse (die etwa von einem anderen Entwickler definiert wurde) hinzugefügt werden, so daß die dynamische Redefinition nicht mehr notwendig wird.

Ein weiterer in KONWERK bisher jedoch nicht genutzter Vorteil der Redefinierbarkeit von Konzepten wird dem Benutzer (dem Modellierer der Domäne) gegeben. Er kann während der Konfigurierungsphase einem von ihm vorher definiertem Konzept zusätzliche Slots hinzufügen. Durch eine entsprechende Verwaltung (in CLOS durch ein spezielles Protokoll realisiert) werden bereits erzeugte Instanzen angepaßt. Für eine Diskussion von derartigen *Extended Instances* vergleiche auch [2], [13] und [12].

Abbildung 1: Klassen und Metaobjektklassen

3 Kurzer Überblick über das Metaobjekt Protokoll

Im MOP (für eine ausführliche Behandlung vgl. [9] bzw. [5]) werden für die grundlegenden Konzepte von CLOS Beschreibungsmittel eingeführt, die die Struktur und das Verhalten dieser Konzepte verdeutlichen. Sie werden "Metaobjekte" genannt, da sie Informationen *über* die Objekte von CLOS enthalten. Metaobjekte werden wiederum mit Hilfe der CLOS-Konzepte (Klassen, Methoden usw.), die sie beschreiben, implementiert, d.h. CLOS wird mit CLOS implementiert. Diese Eigenschaft wird *metazirkulär* genannt (vgl. auch [11]). Metaobjekte sind so Instanzen von Metaobjektklassen. Klassen-Metaobjekte beschreiben die Struktur von Klassen und deren Instanzen, generische Funktions-Metaobjekte beschreiben generische Funktionen und Methoden-Metaobjekte beschreiben Methoden. Einige vordefinierte Metaobjektklassen sind *standard-class, standard-generic-function, standard-method*; spezialisierte Metaobjektklassen können unter diesen eingeordnet werden. Eine Klassen-Metaobjektklasse (M) ist einerseits dadurch gekennzeichnet, daß ihre Instanzen (I) wiederum Klassen (K) sind. Andererseits ist aus der Sicht der Instanzen von K M die Klasse der Klasse K.[1]

Für die Metaobjektklassen werden generische Funktionen definiert, die ihr Verhalten und damit das Verhalten von CLOS beschreiben. Durch das Einhängen weiterer Methoden bzgl. spezialisierter Metaobjektklassen zu den gegebenen generischen Funktionen wird das Verhalten von CLOS modifiziert. Eine solche generische Funktion ist mit *(setf slot-value-using-class)* gegeben, welche für einen Slot einer Instanz einen neuen Slotwert einträgt.

In [9] wird gezeigt wie CLOS, um Konstrukte, wie Introspektion von Klassen und Methoden, Facetten, effiziente Slotzugriffe per Hashtable, diverse Vererbungsmechanismen u.v.m. mit Hilfe des MOP implementiert werden können. Im weiteren wollen wir zeigen, wie dynamische Slots mittels des MOP realisiert werden können.

4 Domänenobjekt- und Eigenschaftsverwaltung in der objektorientierten Basis von KONWERK

Eine Anforderung besteht in dem Wunsch, dynamische Slots bearbeiten zu können.

[1] Klassen-Metaobjektklassen werden auch Metaklassen genannt. Um jedoch die Unterscheidung zu den andern genannten Metaobjektklassen zu bekommen, sollte nach [9] dieser Begriff nicht mehr verwendet werden.

Darunter verstehen wir hier, die Möglichkeit auf Eigenschaften zuzugreifen, welche nicht vorher definiert wurden und sie zu setzen. Dies soll auf Domänenobjekte (d.h. Beschreibungen von realen Dingen) und ihren Instanzen (d.h. das programmtechnische Pendant eines realen Gegenstandes) möglich sein. Da wir Domänenobjekte auf CLOS-Klassen abbilden, bedeutet dies, daß eine Klasse unter Hinzunahme des neuen Slots neudefiniert wird. Dazu wird der Slotzugriff, der im Standardfall nur auf existierende (d.h. in der Klassendefinition vorkommende) Slots möglich ist und sonst einen Fehler hervorruft, angepaßt. Wir unterscheiden hier den Slotzugriff auf eine Klasse und auf eine Instanz. Wird auf eine Klasse mit einem unbekannten Slot zugegriffen (dies ist unten mit dem Test *is-property-p* angedeutet), bedeutet dies, daß die Instanzen der Klasse diesen Slot haben sollen. Die Klasse muß sich selbst redefinieren. Anders bei Instanzen: greift man dort auf einen nicht existierenden Slot zu, muß die Klasse der Instanz neu definiert werden.[2]

Slotzugriffe werden von Klassen-Metaobjektklassen verwaltet. Wir haben also die Situation, wie sie in Abbildung 1 gezeigt ist. (Sei *DO-object* die Oberklasse aller für ein bestimmtes Konfigurierungsproblem notwendigende Domänenobjekte.) *Add-slot-class* ist Klassen-Metaobjektklasse von *DO-object*. Eine Instanz ist die Klasse *DO-class*. Instanzen beider Klassen (*Add-slot-class* und *Do-class*) sind CLOS-Klassen (gekennzeichnet durch die Oberklasse *Standard-class*). Eine vorgegebene generische Funktion von *Add-slot-class* ist *(setf slot-value-using-class)*, diese wurde von *Standard-class* geerbt. Spezialisiert man diese Funktion für *Add-slot-class* (gekennzeichnet durch *(class add-slot-class)*, wobei *class* eine Variable bezeichnet), kann die geforderte Funktionalität der dynamischen Erweiterbarkeit folgendermaßen realisiert werden:

```
(defmethod (setf slot-value-using-class)
           (new-value (class add-slot-class) do-object slot)
   (when (not (is-property-p do-object slot))
         (add-property do-object slot nil))
   (call-next-method))
```

Ist der Existenztest des Slots (hier mit *is-property-p* vereinfacht dargestellt) negativ, wird die Klasse durch eine neue Eigenschaft (Slot der durch die Klasse *do-object* beschriebenen Instanzen) erweitert (*add-property*), sie erhält als Defaultwert *nil*. Dies wird über *reinitialize-instance* realisiert (hier nicht gezeigt). Nach der Erweiterung der Klassenbeschreibung kann der vordefinierte Slotzugriff angestoßen werden (*call-next-method*). CLOS unterstützt mit der Funktion *update-instance-for-redefined-class* die automatische Anpassung von Instanzen, nach der Neudefinition einer Klasse. Durch dieses Protokoll für die Bearbeitung von Instanzen von neudefinierten Klassen, kann vom Programmierer entschieden werden, wie alte, nicht zu der neuen Slotstruktur passende Instanzen verändert werden sollen.

Add-slot-class verwaltet damit CLOS Klassen und sorgt dafür, daß diese Klassen redefiniert werden, wenn sie mit einem neuen Slot angesprochen werden. Da Domänenobjekte mit CLOS Klassen dargestellt werden, verwaltet *Add-slot-class* Domänenobjekte. *Add-slot-class* entscheidet damit, wie auf Domänenobjekte zugegriffen wird. *DO-class* verwaltet Instanzen der Domänenobjekte, d.h. sie entscheidet, wie auf Instanzen zugegriffen

[2]Man beachte, daß die hier vorgestellte Erweiterung von Instanzen bisher nicht in KONWERK verwendet wird und daher auch in Abschnitt 2 nicht diskutiert wird.

Abbildung 2: Die Einordnung eines Domänenobjekts und seiner Instanz

wird. In Abbildung 2 ist die um das spezielle Domänenobjekt *PC* und eine Instanz (*PC-1*) erweiterte Hierarchie gezeigt. Man erhält dadurch statt drei Stufen (Instanz, Klasse, Metaobjektklasse), vier Stufen: Instanz *PC-1*, Klasse *PC*, Metaobjektklasse *Do-class* und deren Klasse ("Metametaobjektklasse") *Add-slot-class*.

Um auch für Instanzen von Domänenobjekten eine Erweiterung zu erlauben, wird eine ähnliche Methode für *(setf slot-value-using-class)* für *DO-class* definiert:

```
(defmethod (setf slot-value-using-class)
           (new-value (do-object do-class) instance slot)
   (when (not (slot-exists-p instance slot))
         (add-property do-object slot nil))
   (call-next-method))
```

Eigenschaften einer Instanz entsprechen CLOS-Slots, daher der Test *slot-exists-p*. *add-property* wird auf die Klasse der Instanz angewandt und damit dieselbe Funktion ausgeführt, wie in der Methode für *Add-slot-class*.

Kontextabhängige Eigenschaftsverwaltung von Instanzen erfordert die Speicherung mehrerer Slotwerte statt eines Slotwertes. Die Einbindung dieser Funktionalität erfolgt durch Einführung der Klasse *check-property-class* (Abbildung 3). *slot-value-using-class* (der lesende Zugriff auf Slotwerte), für *check-property-class* spezialisiert, führt eine Überprüfung des Kontextes durch (hier in *check-instance* und *check-value* nicht gezeigt):

```
(defmethod slot-value-using-class
           ((class check-property-class) instance slot)
   (cond ((check-instance instance)
          (check-value (call-next-method)))
         (t (instance-validity-error instance))))
```

Der Slotwert wird mit der von *standard-class* geerbten Methode ermittelt (*call-next-method*). Sind die Tests erfüllt, wird der Wert von *check-value* zurückgegeben. Eine ähnliche Methode setzt bei erfolgreichem Test einen neuen Slotwert, ohne die vorhergehenden Werte zu löschen, so daß mehrere Slotwerte verwaltet werden können.

Die vorgestellte Änderungen des Slotzugriffs und die notwendige Modifizierung der Instanzen bewirkt einen Effizienzverlust. Dieser könnte wie bereits erwähnt durch spätere Hinzunahme (z.B. nach Festigung der Slotstruktur) der neuen Slots bei der ersten Definition der Klasse umgangen werden. Bei der beschriebenen Einbindung der Constraints

Abbildung 3: Einführung der Klasse *check-property-class*

müßten jedoch die Verwaltungsslots auf Verdacht allen Konzepten hinzugefügt werden, eine Abwägung zwischen Speicher und Zugriffszeit ist hier notwendig. Man bedenke jedoch, daß die eigentliche Redefinition der Klasse nur einmal pro neuem Slot durchgeführt wird und dies auch nur während des meist nicht zeitintensiven Modellierungsprozesses.

5 Diskussion

Der vorgestellten Erweiterungen von CLOS koppeln das Protokoll, welches für die Redefinition von Klassen, von CLOS zur Verfügung gestellt wird, mit Funktionen, die im MOP definiert werden. Das Redefinitionsprotokoll bietet ein vorgegebenes Verhalten bei der Aktualisierung von Instanzen, wenn Klassendefinitionen geändert werden. So werden gelöschte und neu zu einer Klasse hinzugekommenen Slots beim Zugriff auf eine bereits bestehende Instanz so berücksichtigt, als ob die Redefinition die Erstdefinition der Klasse ist. (Für eine genauere Beschreibung dieses Protokolls wird auf [8] verwiesen.) Dieses Protokoll ist durchaus mit Ansätzen in anderen Sprachen (z.B. ORION [1], ein Überblick liefert [10]), welche Schema Evolution hervorheben (also die Änderung der Menge von Klassendefinitionen, wie z.B. Änderung der Klassendefinitionen durch Hinzufügen, Löschen oder Ändern von Slots (Instanzvariablen) und Methoden, der Wechsel der Oberklassen einer Klasse und das Hinzufügen und Löschen von Klassen usw.) zu vergleichen. Weiterhin liefert es die Möglichkeit die im Protokoll spezifizierten Funktionen (z.B. *update-instance-for-redefined-class*, welche die gelöschten und neu hinzugekommenen Slots einer redefinierten Klasse und eine noch nicht geänderte Instanz zur Bearbeitung dem Benutzer anbietet) zu spezialisieren. Dadurch wird ein weites Feld von möglichen Verhaltensweisen einer Klasse bei Redefinition erschlossen.

Durch unsere Kopplung der Redefinition mit dem MOP wird der Zeitpunkt festgelegt, wann eine Redefinition stattfinden soll. Statt dem Benutzer die Evaluierung einer neuen Klassendefinition (die er sich aus der bisherigen Definition extrahieren müßte) aufzubürden, wird dies beim Zugriff auf einen nicht definierten Slot automatisch erledigt. Dazu wird nicht die bestehende Klasse gelöscht und eine erweiterte definiert, sondern die bestehende Klasse durch neue Initialisierung erweitert. Dabei kommt sehr zum Tragen, daß eine CLOS-Klasse wiederum ein Bestandteil eines objektorientierten Programms ist, nämlich die Instanz einer weiteren Klasse. Ein solches Metaobjekt kann daher genau so

behandelt werden, wie eine vom Benutzer definierte Instanz. Das zeigt sich daran, daß für die erwähnte neue Initialisierung das von CLOS spezifizierte Initialisierungsprotokoll (siehe [8]) für die Redefinition verwendet wird (nämlich die kurz erwähnte Funktion *reinitialize-instance*).

Ein solches Zusammenspiel verschiedener Protokolle, welches in den unterschiedlichen Teilprotokollen des MOP gipfelt, liefert letztendlich die Stärke objektorientierter Programme. Durch die Öffnung der Implementation von CLOS durch das MOP ist zusätzlich eine Erweiterung von CLOS meist durch Änderung weniger Funktionen leicht möglich. Damit wird nicht mehr eine spezielle Sprache beschrieben, sondern ein Bereich von Sprachen, der die erweiterten Grenzen seiner Einsetzbarkeit verwischen läßt.

6 Zusammenfassung

Es wurde anhand zweier kurzer Beispiele, die die Erweiterung von Klassen- und Instanzenverwaltung und die Versionsverwaltung von Slotwerten betreffen, gezeigt, wie das Metaobjekt Protokoll verwendet werden kann, um Anforderungen einer Wissensrepräsentationssprache für Konfigurierungsaufgaben zu erfüllen. Dabei wird die objektorientierte Sprache CLOS so angepaßt, daß die Sprache selbst die geforderte Funktionalität erhält. Speziell durch die vorgestellte Erweiterbarkeit der Slotstruktur einer Klasse beim Zugriff auf einen bisher unbekannten Slot, wird die starre Slotstruktur, wie sie in anderen OO-Sprachen vorgegeben ist, aufgehoben. Dies wird in unserer Konfigurationsanwendung dann notwendig, wenn Konzepte (dargestellt durch Klassen) durch Verwaltungsinformationen erweitert werden sollen (z.B. zur Kennzeichnung von in Constraints vorkommenden Konzepten). Diese Information ist nicht in Oberklassen bündelbar, da zur Zeit der Konzeptdefinition nicht bekannt sein muß, welche Constraints definiert werden.

Es zeigte sich, daß die Implementation einer Sprache mittels eines Metaobjekt Protokolls nützlich ist, um die Sprache schnell und portabel zu erweitern. Eine solche offene Architektur einer objektorientierten Sprache ermöglicht es, die Grenzen, die durch das vorgegebene Verhalten der OOP gegeben sind, zu überschreiten.

Literatur

[1] J. Banerjee, W. Kim, K. Kim und H. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In „Proc. ACM Special Interest Group on Management of Data“, ACM SIGMOD, S. 311–322, San Francisco (1987).

[2] R. Cunis. „Das 3-stufige Frame-Repräsentationsschema - eine mehrdimensionale modulare Basis für die Entwicklung von Expertensystemkernen“. DISKI Reihe 15. infix, St. Augustin (1992).

[3] R. Cunis, A. Günter und H. Strecker (Hrsg.). „Das Plakon-Buch“. Nr. 266 in Informatik Fachberichte. Springer (1991).

[4] J. DeKleer. An Assumption-Based TMS. *Artificial Intelligence* **28**(2), 127–162 (1986).

[5] R. P. Gabriel. Book Review, G. Kiczales, Jim des Rivieres and D. G. Bobrow, The Art of the Metaobject Protocol. *Artificial Intelligence* (61), 331–342 (1993).

[6] A. Günter. „Flexible Kontrolle in Expertensystemen für Planungs- und Konfigurierungsaufgaben“. DISKI Reihe 3. infix, St. Augustin (1992).

[7] A. Günter, H. Dörner, H. Gläser, B. Neumann, C. Posthoff und H.-J. Sebastian. „Das Projekt PROKON: Problemspezifische Werkzeuge für die wissensbasierte Konfigurierung“. PROKON-Bericht, Nr. 1 (1991).

[8] S. E. Keene. „Object-Oriented Programming in Common Lisp“. Addison-Wesley Publishing Company (1989).

[9] G. Kiczales, D. G. Bobrow und J. des Rivieres. „The Art of the Metaobject Protocol“. MIT Press, Cambridge, MA (1991).

[10] G. Nguyen und D. Rieu. Schema Evolution in Object-Oriented Database Systems. Bericht 947, Programme 4, Groupe de Recherche, Grenoble (1988).

[11] A. Paepcke. Pclos: Stress Testing CLOS, Experiencing the Metaobject Protocol. In „Proc. OOPLSA '90“, ACM SIGPLAN, S. 194 – 211 (1990).

[12] G. Rydqvist, M. Patel und L. Larsson. Types in Space: Towards Flexible High-Speed Object Oriented Data Managment Tools of the Future. Bericht LiTH-IDA-R-88-43, CADLAB Research Report Universität Linköping, Schweden (1988).

[13] L. Stein. A Unified Methodology for Object-Oriented Programming. In M. L. et al. (Hrsg.), „Inheritance Hierarchies in Knowledge Representation and Programming Languages“, S. 211–222. John Wiley & Sons (1991).