Formale Spezifikation und Prototyping im Sprachentwurf: Eine Fallstudie*

Wilhelm Hasselbring

Software Engineering, Fachbereich Mathematik und Informatik
Universität Gesamthochschule Essen, 45117 Essen
willi@informatik.uni-essen.de

Zusammenfassung

Spezifikationen spielen eine zentrale Rolle in der Software-Entwicklung. Eigenschaften formaler Spezifikationen wie Widerspruchsfreiheit und Eindeutigkeit verbessern die Kommunikation zwischen Auftraggebern und Entwicklern und beeinflussen entscheidend die Qualität der danach konstruierten Programme. Zu einer formalen Spezifikation kann ein Prototyp erzeugt werden, der dann durch Ausführung getestet und vorgeführt werden kann. Auf diese Weise kann mit dem zu entwickelnden System bereits frühzeitig experimentiert werden, um die Angemessenheit einer Spezifikation beurteilen und gegebenenfalls korrigieren zu können.

Der Sprachentwurf eignet sich besonders gut zum Einsatz formaler Spezifikationen, da eine Programmiersprache selbst eine formale Sprache ist. Die Grammatik wird üblicherweise mithilfe von Backus-Naur-Form angegeben. Eine Backus-Naur-Form definiert zwar die Syntax einer Programmiersprache, jedoch nicht deren Semantik. Zur Definition der Semantik ist eine formale Spezifikationssprache hilfreich.

Unser Ausgangspunkt ist, eine Sprache zum Prototyping paralleler Algorithmen zu entwickeln. Wir entwickeln und implementieren das Programmiersystem für diese Sprache auf eine eher unkonventionelle Art: die informelle Spezifikation wird durch eine formale Spezifikation ergänzt und für die formale Spezifikation wird dann ein ablauffähiger Prototyp konstruiert, bevor das System in einer Produktionssprache implementiert wird.

Stichworte: Sprachentwurf, Formale Spezifikation, Prototyping, Z, Object-Z, ProSet, Linda

1 Einleitung

Es ist eine wohlbekannte Tatsache, daß die Kosten zur Beseitigung eines Entwurfsfehlers in einem Software-System drastisch steigen, je weiter der Entwicklungsprozeß vorangeschritten ist. Fehler in der Anforderungsdefinition und im Entwurf sind nur sehr schwer zu korrigieren, wenn sie z.B. erst nach der Installation entdeckt werden. Formale Spezifikationen und Prototyping zielen darauf hin, solche Fehler so früh wie möglich zu entdecken und zu beseitigen.

Spezifikationen dienen dazu, Systemanforderungen präzise, eindeutig und vollständig zu erfassen. Sie sind Gegenstand der Kommunikation und vertraglicher Vereinbarungen zwischen Auftraggebern und Entwicklern. Sie bilden Vorgaben für Entwurfs-, Programmier- und Validierungsaufgaben und tragen zur systematischen Wiederverwendbarkeit von Software-Komponenten sowie zur evolutionären Weiterentwicklung (Wartung) im Einsatz befindlicher Software-Systeme bei.

Um diesen Anforderungen gerecht zu werden, sollten Spezifikationen abstrakte, problemnahe Begriffsbildungen unterstützen, präzise sein, einen modularen Aufbau aufweisen und einen weitgehenden Rechnereinsatz bei ihrer Konstruktion und Validierung ermöglichen. *Abstraktion* verbessert die Verständlichkeit von Spezifikationen und läßt Raum für Entwurfsentscheidungen und alternative Implementierungen. *Präzision* ist eine notwendige Voraussetzung, um Fehlinterpretationen zu vermeiden sowie Mehrdeutigkeiten oder Widersprüche auszuschließen.

In den vergangenen Jahren hat sich daher ein zunehmendes Interesse an formalen Spezifikationssprachen und Methoden entwickelt. Wesentliches Merkmal formaler Sprachen und Methoden ist die Verwendung von mathematischen Formalismen. Diese Sprachen sind mit einer wohldefinierten Menge von Symbolen und Regeln zur Konstruktion wohlgeformter Ausdrücke ausgestattet und jedem Ausdruck ist eine eindeutige Bedeutung zugeordnet. Anders als informelle Methoden erzwingen formale Methoden präzise Formulierungen und liefern objektive Kriterien, um mehrdeutige oder widersprüchliche Angaben zu erkennen.

In diesem Beitrag wollen wir anhand der Entwicklung und Implementation einer parallelen Programmiersprache den Einsatz formaler Spezifikationen und

*Akzeptiert zur Präsentation auf der GI-Fachtagung Softwaretechnik 93 am 8.–9. November 1993 in Dortmund.

des Prototyping im Sprachentwurf diskutieren. Dazu gehen wir zunächst allgemein auf das Verhältnis von informellen Spezifikationen zu formalen Spezifikationen (Abschnitt 2) und auf das Verhältnis von formalen Spezifikationen zum Prototyping ein (Abschnitt 3). Die zu spezifizierende parallele Programmiersprache wird in Abschnitt 4 kurz vorgestellt und die formale Spezifikationssprache Object-Z wird in Abschnitt 5 eingeführt. Daraufhin werden wir die formale Spezifikation selbst (Abschnitt 6) und die dafür konstruierte Prototyp-Implementation (Abschnitt 7) präsentieren. Wir beenden unseren Beitrag mit einigen Bemerkungen in Abschnitt 8.

Die Diskussion zur formalen Semantik von Linda mit Keld Kondrup Jensen, sowie die Kommentare von Stephen Gilmore zu einer vorläufigen Spezifikation waren sehr hilfreich. Dank gilt auch Ernst-Erich Doberkat, Claus Pahl und den anonymen Gutachtern für die Anmerkungen zu früheren Versionen dieses Beitrags.

2 Informelle und formale Spezifikationen

Spezifikation der Anforderungen ist die Phase im Software-Entwicklungsprozeß, in der die zu lösenden Aufgaben definiert werden. Diese Aufgaben werden meistens informell spezifiziert. Probleme, wie z.B. das Problem der dinierenden Philosophen [5], werden aufgestellt, ohne formale Notationen zu verwenden. Der Vorteil einer informellen Spezifikation ist, daß das Problem ohne den Ballast einer formalen Notation studiert werden kann. Bildhafte Beschreibungen von Philosophen, die an einem Tisch sitzen und Stäbchen austauschen, sind hilfreich, um ein Problem zu verstehen und zu lösen.

Informelle Spezifikationen haben den Vorteil, daß natürlich-sprachliche Anforderungen direkt von zukünftigen Anwendern gelesen und überarbeitet werden können. Andererseits haben informelle Spezifikationen häufig die Eigenschaft, unvollständig, inkonsistent und mehrdeutig zu sein. Diese Probleme veranlaßten Software-Entwickler dazu, informelle Spezifikationen durch formale Spezifikationen zu ergänzen, deren Konsistenz automatisch überprüft werden kann.

Ein Argument, das oft gegen formale Spezifikationen geäußert wird, ist die Nachfrage von Endbenutzern nach leicht verständlichen Dokumentationen. Dieses Argument ist das Resultat einer falschen Einordnung von formalen Spezifikationen. Endbenutzer sollten natürlich keine formalen Spezifikationen lesen müssen. Wie in anderen Ingenieurdisziplinen auch, werden in der Software-Entwicklung sowohl leicht verständliche Dokumentationen für Endbenutzer, als auch technische Entwurfsdokumente für Software-Entwickler benötigt. Der Hauptvorteil von natürlich-sprachlichen Beschreibungen ist deren Verständlichkeit. Man sollte sich bei der Verwendung natürliche Sprache auf diese Eigenschaft konzentrieren und sie nicht für Präzision und Gründlichkeit überstrapazieren.

Der Sprachentwurf eignet sich besonders gut zum Einsatz formaler Spezifikationen, da eine Programmiersprache selbst eine formale Sprache ist. Um die Vorteile formaler Spezifikationen im Sprachentwurf an einem Beispiel zu erörtern, werden wir im folgenden einen kurzen Blick auf die Entwicklung der parallelen Programmiersprache C-Linda werfen. C-Linda ist die Kombination der (sequentiellen) Programmiersprache C mit der Koordinationssprache Linda [3]. Linda ist keine vollständige Programmiersprache. Vielmehr werden Sprachkonstrukte eingeführt, die sich unter Beachtung des jeweiligen Typkonzepts in konventionelle Programmiersprachen integrieren und so parallele Programmiersprachen entstehen lassen. Bemerkenswert ist das Kommunikationsmodell, nach dem der Austausch von Daten durch einen Assoziativspeicher — den Tupelraum — vermittelt wird. Die Kommunikation und Synchronisation paralleler Prozesse basiert auf Tupeln, die durch Linda-Operationen im Tupelraum abgelegt und von dort wieder entnommen werden können. Diese Art der Kommunikation wird auch als *generative Kommunikation* bezeichnet. C-Linda gestattet die dynamische Generierung von Prozessen mithilfe der `eval`-Operation, die sogenannte *aktive* Tupel mit aktiven Prozessen als Komponenten erzeugt.

Die Definition von C-Linda wurde informell präsentiert [13] und enthielt infolgedessen Auslassungen und Mehrdeutigkeiten. In [21] werden z.B. vier verschiedene Arten beschrieben, auf die C-Linda's `eval`-Operation interpretiert und dann implementiert wurde. Folglich kann ein C-Linda-Programm mit verschiedenen C-Linda-Compilern verschiedene Ergebnisse liefern. Solch eine Situation erfordert eine präzise Definition. Trotzdem hat die informelle Spezifikation ihre Vorteile, da sie leicht zu verstehen ist. Die Popularität von Linda kann zum Teil wohl dieser Eigenschaft zugeschrieben werden.

Für die Compiler-Implementierer entstehen jedoch erhebliche Probleme durch die unklare informelle Definition. Um diese Probleme zu vermeiden, wurden von verschiedenen Autoren formale Spezifikationen von Linda geschrieben. Beispielsweise wird in [4] Linda mit Plotkin's Structural Operational Semantics (SOS), mit Milner's Calculus of Communicating Systems (CCS), mit Petri-Netzen und mit der Chemical Abstract Machine (CHAM) formal definiert.

Um die Probleme, die durch eine informelle Spezifikation entstehen können, von vornherein zu vermeiden, haben wir unseren Ansatz zur parallelen Programmierung mithilfe der formalen Spezifikationssprache Object-Z [11] definiert (siehe Abschnitt 6). Dieser Ansatz zur parallelen Programmierung kombiniert die mengen-orientierte Prototyping-Sprache

ProSet [9] mit der Koordinationssprache Linda. Der Name ProSet steht für Prototyping with Sets. Diese mengen-orientierte Prototyping-Sprache ist eine Weiterentwicklung von SETL [7, 23]. Beide Sprachen stellen beliebige endliche Mengen und Tupel als zusammengesetzte Datenstrukturen, sowie Konstrukte der Prädikatenlogik zur Verfügung. Das hohe Niveau dieser Konstrukte qualifiziert mengenorientierte Sprachen zum Prototyping. Für eine allgemeine Einführung in das Prototyping mit mengenorientierten Sprachen verweisen wir auf [7]. Eine Fallstudie zum Prototyping mit SETL ist in [20] dokumentiert. Bemerkenswerte Weiterentwicklungen in ProSet gegenüber SETL sind Persistenz und Parallelität [10].

Es sei noch angemerkt, daß wir formale Spezifikationen nicht als *Ersatz* für informelle Spezifikationen ansehen. Sie sind vielmehr eine *Ergänzung* für natürlich-sprachliche Beschreibungen, die insbesondere als Basis zur Software-Entwicklung dient. Formale Spezifikationen können auch benutzt werden, um bessere natürlich-sprachliche Beschreibungen zu erhalten, da die formale Notation den Schreiber zu Fragen führt, die in einer informellen Spezifikation gar nicht erst aufgestellt und somit auch nicht beantwortet worden wären.

3 Formale Spezifikationen und Prototyping

Was kann man mit einer formalen Spezifikation tun, nachdem sie geschrieben wurde? Selbstverständlich sollte sie die Basis für die nächsten Schritte in der Entwicklung der Software sein: Programmentwurf und Implementation.

Traditionell werden die meisten großen Software-Projekte nach sogenannten Life Cycle-Plänen durchgeführt [2]. Hierbei wird jede Phase als ein Eingabe/Ausgabe-Schritt angesehen. Nur eine minimale Unterstützung für die Rückkopplung zu früheren Phasen ist vorgesehen. Wir verweisen hier auf [1, 7] für kritische Bewertungen dieses Ansatzes. Wesentliche Probleme sind:

- Die Endbenutzer sind von der Entwicklung ausgeschlossen.
- Die Wartung ist ungeplante Software-Entwicklung.
- Life Cycle-Pläne sind ungeeignet zur praktischen Projektkontrolle.

Konsequenterweise versuchen wir erst gar nicht, direkt aus der formalen Spezifikation ein Programm in einer Produktionssprache zu schreiben, sondern konstruieren zunächst einen ausführbaren Prototyp auf hohem Niveau (siehe Abschnitt 7). Dieser Ansatz wird auch "testing formal specifications" genannt [19]. Es ist notwendig, Spezifikationen so früh wie möglich zu *testen*, um Systeme zu entwickeln, die auch tatsächlich die von den Endbenutzern gewünschten Eigenschaften erfüllen. Da die Endbenutzer im allgemeinen nicht genau wissen, was für ein System sie eigentlich benötigen, ergeben sich viele Anforderungen erst während des Testens eines Prototypen (*Validation* der Spezifikation). Es ist grundsätzlich nicht möglich, die Übereinstimmung einer formalen Spezifikation mit den informellen Anforderungen durch eine formale *Verifikation* zu überprüfen.

Zunächst hilft der Prototyp den Entwicklern die Spezifikation zu testen, und dann den Endbenutzern, die Möglichkeiten des gewünschten Systems zu erkunden. Es ist besser, wenn die Endbenutzer ihre Anforderungen früh im Entwicklungsprozeß erkennen und überprüfen, und nicht erst nachdem das System vollständig geliefert wurde. Der Prototyp eines Programms ist das ausführbare Modell eines Produktionsprogramms, an dem wesentliche Eigenschaften des fertigen Programms studiert werden können. Herausragende Aspekte des Software Prototyping sind einmal der modellhafte Charakter des Prototypen, zum anderen aber auch die Möglichkeit, ein Programm im Dialog zwischen Anwender und Entwickler entwerfen zu können. Der Prototyp kann auch die Erkenntnis liefern, daß die Realisierung des intendierten Produktionsprogramms nicht möglich oder nicht sinnvoll ist (Risikovermeidung).

4 Die zu spezifizierende Sprache

Parallele Programmierung ist konzeptionell schwieriger durchzuführen und zu verstehen als die sequentielle Programmierung, weil sich ein Programmierer bei der parallelen Programmierung oft gleichzeitig auf mehrere Prozesse konzentrieren muß. Außerdem müssen die Programmierer auf parallelen Maschinen üblicherweise auf einem sehr niedrigen Niveau programmieren, um Höchstleistungen zu erreichen — die Einfachheit in der Benutzung wird der Effizienz in der Ausführung geopfert. Aufgrund dieser Faktoren ist die Entwicklung paralleler Algorithmen im allgemeinen eine äußerst schwierige Aufgabe. Das Ziel unseres Ansatzes ist es, das Prototyping von parallelen Algorithmen durch die Bereitstellung von Konstrukten mit hohem Niveau zur expliziten Parallelprogrammierung wesentlich zu erleichtern. Wir skizzieren diesen Ansatz hier nur sehr kurz und verweisen für Details auf [18].

Für eine Prototyping-Sprache ist es notwendig, zum Prototyping paralleler Algorithmen einfache, aber mächtige Konstrukte zur dynamischen Prozeßkreation und zur Koordination paralleler Prozesse zur Verfügung zu stellen. In ProSet wird das Konzept der Prozeßkreation durch Multilisp's Futures [15] auf die mengen-orientierte Programmierung zugeschnitten und mit dem flexiblen Konzept zur Synchronisation und Kommunikation durch Linda's Tupelraum

[13] kombiniert. Dieser Tupelraum ist ein virtueller gemeinsamer Datenraum, über den die Prozesse kommunizieren. Synchronisation und Kommunikation erfolgen durch das Einfügen, Entfernen, Lesen und durch unteilbares Ändern von einzelnen Tupeln im Tupelraum. Der Zugriff erfolgt assoziativ und nicht über Adressen, so daß Hardware-Unabhängigkeit gewährleistet ist. Parallele Prozesse werden durch den `||`-Operator dynamisch gestartet. Falls der Rückgabewert eines so gestarteten Prozesses später benötigt wird, muß dann gegebenenfalls gewartet werden, bis der entsprechende Prozeß terminiert (analog zur Prozeßkreation in Multilisp).

Das parallele Programmiermodell von ProSet liefert somit eine räumlich und zeitlich ungeordnete Menge von Prozessen, die weitgehend unabhängig voneinander programmiert werden können: Prozesse, die miteinander kommunizieren möchten, müssen nicht zur gleichen Zeit laufen und sie müssen sich nicht gegenseitig kennen. Diese Eigenschaft befähigt den Programmierer, sich auf die individuellen Prozesse einzeln zu konzentrieren. Das bringt die Parallelprogrammierung konzeptionell in die gleiche Problemklasse wie die herkömmliche sequentielle Programmierung. Das Tupelraum-Management gewährleistet gegenseitigen Ausschluß und Synchronisation bzgl. gemeinsamer Daten, so daß sich der Programmierer nicht selbst darum kümmern muß. Abschließend sei noch angemerkt, daß die Integration von Linda in ProSet sehr natürlich ist, da in beiden Modellen Tupel eine zentrale Rolle spielen.

Zur Illustration der Sprache geben wir in Abbildung 1 eine Lösung des bekannten Problems der dinierenden Philosophen an [5]. Durch die Funktion `CreateTS` wird zuerst ein neuer Tupelraum erzeugt. Dann werden durch `||` die `n` parallelen Prozesse (die Philosophen) gestartet. Die Reihenfolge, in der die Philosophen gestartet werden, ist nicht signifikant. Durch die `deposit`-Operation werden vom Hauptprogramm die zur Koordination benötigten Tupel im Tupelraum abgelegt. Das sind `n` Stäbchen und `n-1` Tickets für den Tisch. Die Zuordnung von Philosophen zu ihren Sitzplätzen und zu den Stäbchen erfolgt über die entsprechenden ganzzahligen Parameter. Zunächst denken alle Philosophen. Sobald sie hungrig werden, müssen sie sich zuerst ein Ticket und dann die beiden Stäbchen neben ihrem Teller durch die `fetch`-Operation aus dem Tupelraum holen, bevor sie essen können. Nach dem Essen müssen das Ticket und die Stäbchen durch `deposit` wieder zurückgegeben werden. Zusätzlich gibt es in ProSet noch die `meet`-Operation, die ein Tupel im Tupelraum liest und auch ändern kann, in diesem einfachen Beispiel jedoch nicht benötigt wird. Die Lösung arbeitet für beliebige $n > 1$. Deadlocks werden vermieden, indem nur `n-1` Philosophen gleichzeitig am Tisch sitzen können: mindestens einer kann immer beide Stäbchen nehmen und dann essen.

```
program DiningPhilosophers;
  visible constant
    n := 5,             -- Anzahl Philosophen
    TS := CreateTS(100); -- Neuer Tupelraum
begin
  for i in [ 0 .. n-1 ] do
    || phil(i); -- Start der Philosophen
    deposit [ "chopstick", i ] at TS
    end deposit;  -- Die Staebchen
    if i /= n-1 then -- Nur n-1 Tickets
      deposit [ "table ticket" ] at TS
      end deposit;
    end if;
  end for;

  procedure phil (i);
  begin
    loop
      think ();
      -- Eintrittskarte und die Staebchen holen:
      fetch ( "table ticket" ) at TS end fetch;
      fetch ( "chopstick", i ) at TS end fetch;
      fetch ( "chopstick", (i+1) mod n ) at TS
      end fetch;
      eat ();
      -- Alles wieder Zurueckgeben:
      deposit ["chopstick", i] at TS end deposit;
      deposit ["chopstick", (i+1) mod n] at TS
      end deposit;
      deposit ["table ticket"] at TS end deposit;
    end loop;
  end phil;
end DiningPhilosophers;
```

Abbildung 1: Lösung des Problems der dinierenden Philosophen.

5 Die formale Spezifikationssprache Object-Z

Algebraische Techniken zur Spezifikation abstrakter Datentypen bilden die Grundlage für algebraische Spezifikationssprachen [12]. Sie legen funktionale Eigenschaften von Software-Komponenten unabhängig von der konkreten Darstellung von Daten und unabhängig von der konkreten Implementierung der darauf definierten Operationen fest.

In eine andere Klasse formaler Methoden gehören modell-orientierte Ansätze wie Z [25]. Sie unterscheiden sich von den eigenschafts-orientierten algebraischen Methoden dadurch, daß sie Systemverhalten direkt in der Form eines abstrakten Modells beschreiben. Diese Modelle beruhen auf bekannten mathematischen Strukturen wie Mengen, Tupeln, Folgen, Funktionen und Relationen. Es sei angemerkt, daß der Begriff *Modell* im algebraischen Kontext eine andere Bedeutung hat als im Z-Kontext.

Object-Z [11] ist eine objekt-orientierte Erweiterung der formalen Spezifikationssprache Z. Beide Sprachen basieren auf der Mengenlehre und der Prädikatenlogik. Wir verweisen auf [6, 25] für eine Einführung in Z. Object-Z wurde aus verschiedenen Gründen zur Spezifikation von ProSet-Linda gewählt. Zunächst hat Z viele Gemeinsamkeiten mit ProSet, da auch ProSet auf der Mengenlehre und der Prädikatenlogik basiert. Das erleichtert zum einen das Verständnis der Spezifikation für Leser, die ProSet bereits kennen. Zum anderen ist es so relativ leicht möglich, aus der formalen Spezifikation einen ausführbaren Prototypen mit ProSet zu erzeugen, wie wir in Abschnitt 7 sehen werden. Außerdem existieren einige Werkzeuge, die die Erstellung von Z-Spezifikationen unterstützen [22].

Wie üblich wurde auch ProSet-Linda zunächst informell definiert [16]. Eine vorläufige formale Spezifikation von ProSet-Linda mit Z wurde dann in [17] präsentiert. Die eingeschränkten Strukturierungsmöglichkeiten von Z und das Fehlen von temporaler Logik veranlaßten uns, die Spezifikation in Object-Z umzuschreiben. Die Spezifikation konnte mit Object-Z besser strukturiert und übersichtlicher präsentiert werden, als das mit Z möglich war. Insbesondere die Möglichkeiten zur Einkapselung von Information und die Wiederverwendung durch Vererbung sind hilfreich zur Strukturierung einer Spezifikation.

Object-Z- und Z-Spezifikationen bestehen aus einer Kombination von formalem Text und natürlichsprachlichen Erklärungen. Der formale Text liefert die präzise Spezifikation, während die natürliche Sprache die formalen Teile einführt und erklärt. In Object-Z besteht der formale Text aus zwei Teilen: Klassen und Schemata, die die Spezifikation strukturieren, sowie mathematischen Formeln, die die Präzision liefern. Z (bzw. Object-Z) hat keine Konstrukte zur Spezifikation von Parallelität. Trotzdem kann mit Z eine parallele Programmiersprache spezifiziert werden: Parallelität wird durch beliebige Überlagerung von Transaktionen modelliert. Unser Ziel ist nicht die Spezifikation von Parallelität, sondern die Spezifikation der Semantik von ProSet-Linda.

Object-Z erweitert Z um eine Klassenstruktur, die ein Zustandsschema zusammen mit den dazugehörigen Operationen auf diesem Zustand einkapselt. Eigenschaften einer Klasse können durch Vererbung an andere Klassen weitergereicht werden. Das Modell einer Klasse basiert auf der *Historie* von Operationen, die eine Instanz (Objekt) dieser Klasse durchläuft. In Object-Z versteht man unter der *Historie* eines Objekts die Folge von Operationen, die es durchläuft. Die Menge der möglichen Historien kann durch Prädikate mit temporaler Logik beschränkt werden. Diese Prädikate werden üblicherweise *Lebendigkeitseigenschaften*, wie z.B. Fairneß, definieren. Einer der Vorteile von Object-Z ist, daß diese Eigenschaften direkt im Modell spezifiziert werden können. In Z können solche Lebendigkeitseigenschaften nur umständlich getrennt von den Zustands- und Operationsschemata spezifiziert werden.

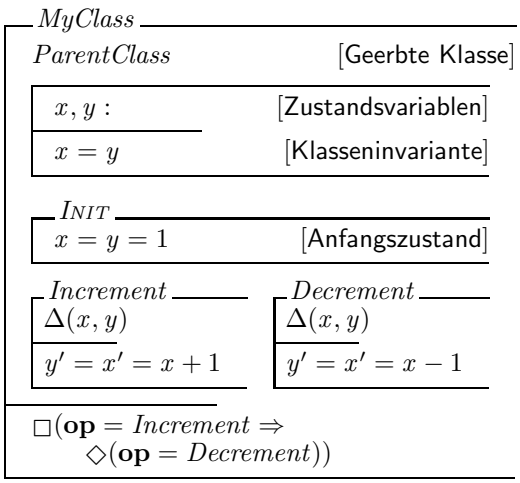


Abbildung 2: Eine einfache Klassendefinition in Object-Z.

Als einführendes Beispiel für Object-Z betrachten wir hier die Spezifikation der einfachen Klasse *MyClass* in Abbildung 2. Zunächst werden dort die Klassen angegeben, von denen geerbt wird (*ParentClass* im Beispiel). Die Zustandsvariablen x und y werden zusammen mit der Klasseninvariante im Zustandsschema definiert. Der Anfangszustand wird im *Init*-Schema spezifiziert. Die möglichen Operationen auf den Zustandsvariablen werden dann in Operationsschemata spezifiziert, die eine Relation zwischen Vor- und Nachzustand definieren (*Increment* und *Decrement* im Beispiel). Die Δ-Liste gibt dabei die Variablen an, die durch die jeweilige Operation geändert werden. Dabei bezeichnet x dann den Vorzustand und x' den Nachzustand der Variablen x bzgl. der Ausführung der Operation. Das letzte Prädikat in Abbildung 2 ist die Historieninvariante, die die erlaubten Folgen von Operationen definiert. Dabei steht **op** für eine Operation in der Historie. Hier sind auch die Operatoren der temporalen Logik erlaubt:

$\Box p$	Prädikat p gilt in jedem zukünftigen Zustand in der Historie.
$\Diamond p$	Irgendwann gibt es einen zukünftigen Zustand in der Historie, in dem p gilt.

Die Historieninvariante im Beispiel spezifiziert, daß *immer* nach einer *Increment*-Operation *irgendwann* eine *Decrement*-Operation ausgeführt werden muß.

6 Die formale Spezifikation

Das Ziel unserer Arbeit ist es nicht, eine formale Spezifikation der gesamten Sprache ProSet zu schreiben.

Wir beschränken uns darauf, generative Kommunikation in ProSet (also ProSet-Linda) zu spezifizieren. Dazu sind einige Abstraktionen notwendig:

$$[Process, Statement, Expression, LValue, Value]$$

Auf diese Art werden neue Basistypen für eine Z-Spezifikation eingeführt, deren interne Struktur an dieser Stelle keine Rolle spielt und daher verborgen bleibt. Vordefiniert sind die Basistypen und . Wir benötigen zunächst genauere Informationen über die Werte in ProSet:

$$\begin{array}{l} true, false : Value \\ boolean, integer, real, string, tuple, set, \\ \quad function, modtype, instance, atom : Value \\ om : Value \\ ValuesOfType : Value \quad Value \\ \hline ValuesOfType\ boolean = \{true, false\} \\ \mathrm{dom}\ ValuesOfType = \\ \quad \{boolean, integer, real, string, tuple, set, \\ \quad function, modtype, instance, atom\} \\ \mathrm{dom}\ ValuesOfType \subset ValuesOfType\ atom \\ (\{\ om \mapsto \{om\}\ \} \cup \{\ t : \mathrm{dom}\ ValuesOfType \bullet \\ \quad t \mapsto ValuesOfType\ t\ \})\ \mathsf{partition}\ Value \end{array}$$

Die Werte vom Typ *boolean* sind somit *true* und *false*. In ProSet gibt es die vordefinierten Typatome *boolean*, *integer* usw., die den Wertebereich (dom) der partiellen, injektiven Funktion *ValuesOfType* bilden (so definiert durch das Funktionssymbol). Diese Typatome selbst sind vom Typ *atom*. Weitere Atome können dynamisch erzeugt werden. Sie werden üblicherweise zur eindeutigen Identifikation von sonst anonymen Objekten verwendet. Der undefinierte Wert *om* zeigt undefinierte Situationen an und hat selbst keinen Typ. Jeder Wert, außer *om*, gehört in ProSet zu genau einer Typmenge (spezifiziert durch das letzte Prädikat).

ProSet ist schwach getypt, d.h. der Typ einer Variablen ist im allgemeinen erst zur Laufzeit bekannt. Der `type`-Operator liefert dann den Typ einer Variablen als vordefiniertes Typatom:

$$\begin{array}{l} Type : Value \quad Value \\ \hline Type\ true = Type\ false \\ Type\ atom = Type\ atom = Type\ boolean = Type\ set \\ \quad = Type\ integer = Type\ real = Type\ string \\ \quad = Type\ tuple = Type\ function \\ \quad = Type\ modtype = Type\ instance \\ Type\ om = Type\ om \\ \forall x : Value \mid x \neq om \bullet \\ \quad x \in ValuesOfType\ (Type\ x) \end{array}$$

Type ist somit eine totale Funktion (so definiert durch das Funktionssymbol), die jeden Wert auf sein Typatom abbildet. Die Anwendung des `type`-Operators auf `om` ist undefiniert und liefert somit den undefinierten Wert. Das letzte Prädikat gibt den Zusammenhang zwischen *Type* und *ValuesOfType* an.

Entsprechend werden Tupel und weitere Grundkonzepte formal spezifiziert. Wir werden die Spezifikation hier nur skizzieren, verweisen daher für Details auf [18], und gehen direkt zur (vereinfachten) Spezifikation eines Tupelraumes über:

TupleSpace

$$\begin{array}{l} Id : Value \\ Limit : Value \\ Tuples : \mathrm{bag}\ Tuple \\ \hline Type\ Id = atom \land Id \notin \mathrm{dom}\ ValuesOfType \\ Type\ Limit = integer \lor Type\ Limit = om \end{array}$$

Ein Tupelraum hat somit eine Identität (*Id*), die ein Atom ist, jedoch kein vordefiniertes Typatom. Das *Limit* gibt eine Grenze für die Anzahl der Tupel an, die gleichzeitig in diesem Tupelraum sein dürfen, falls es ein ganzzahliger Wert ist (Typ *integer*). Falls das *Limit* den undefinierten Wert hat, ist diese Anzahl nicht beschränkt (die Grenze ist also undefiniert). Der Tupelraum enthält eine Multimenge (bag) von Tupeln, wie es in Linda üblich ist.

Der Zustand eines ProSet-Linda-Programms besteht aus einer endlichen Menge von Tupelräumen und einer endlichen Menge von aktiven Prozessen:

ProgramState

$$\begin{array}{l} TSs : \quad TupleSpace \\ ActiveProcs : \quad Process \\ \hline \forall ts1, ts2 : TSs \bullet \\ \quad ts1 \neq ts2 \Rightarrow ts1.Id \neq ts2.Id \end{array}$$

S ist die Menge aller endlichen Teilmengen einer gegebenen Menge S. Die Klasseninvariante definiert, daß jede Tupelraumidentität eindeutig sein muß.

Init

$$\begin{array}{l} TSs = \{\} \\ \#ActiveProcs = 1 \end{array}$$

Initial gibt es noch keine Tupelräume und nur einen Prozeß (das Hauptprogramm).

ProgramTermination

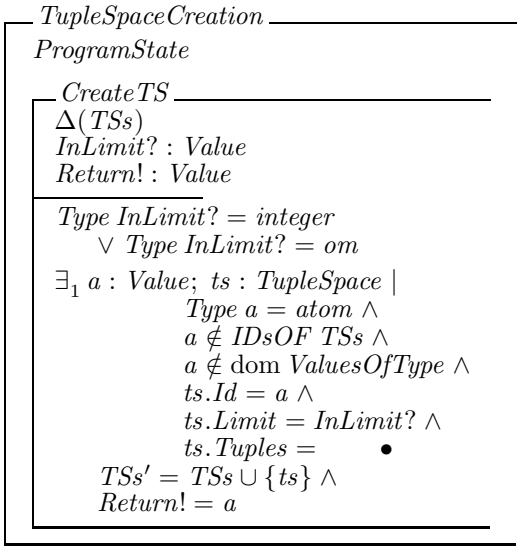
$$\begin{array}{l} \Delta(TSs, ActiveProcs) \\ \hline TSs' = \{\} \\ ActiveProcs' = \{\} \end{array}$$

Nach Programmende werden die Tupelräume wieder gelöscht, d.h. Tupelräume sind nicht persistent.

Für eine Spezifikation der gesamten Sprache ProSet sind weitere Komponenten notwendig, um den Programmzustand zu spezifizieren.

Jetzt müssen noch die anderen Operationen auf diesem Programmzustand definiert werden. Der Zugriff auf den Programmzustand wird durch Vererbung

gewährleistet. Die Funktion `CreateTS` erzeugt einen neuen Tupelraum und gibt seine Tupelraumidentität zurück:



Die Tupelraumidentität ist ein neues Atom, das dynamisch erzeugt wird. Dieses neue Atom darf weder ein Typatom sein noch bereits für einen Tupelraum als Identität eingesetzt worden sein. Die Spezifikation definiert nur die Eigenschaften der neuen Tupelraumidentität, nicht *wie* sie erzeugt wird. Die Funktion *IDsOF* liefert für eine Menge von Tupelräumen die entsprechende Menge von Tupelraumidentitäten. Die Multimenge von Tupeln ist zunächst leer ( ). Wie üblich in Z, werden die Eingabewerte durch ein Fragezeichen (*InLimit*?) und die Ausgabewerte durch ein Ausrufezeichen (*Return*!) markiert.

Die anderen Funktionen zur Verwaltung von Tupelräumen und die drei Tupelraumoperationen `deposit`, `fetch` und `meet` werden entsprechend definiert.

7 Die Prototyp-Implementation

Wie in Abschnitt 3 diskutiert wurde, sollte aus einer formalen Spezifikationen frühzeitig ein Prototyp erzeugt werden, damit die Spezifikationen durch Ausführung des Prototypen auf ihre Angemessenheit hin *getestet* werden kann.

Der ProSet-Compiler wird mithilfe des Compiler-Generators Eli [14] entwickelt und ist in [8] dokumentiert. Wir wollen hier nicht die Implementation des Compilers diskutieren, sondern die Prototyp-Implementation des Laufzeitsystems für die Tupelraumoperationen: das Tupelraum-Management. Die Prototyp-Implementation des Tupelraum-Managements wurde in ProSet selbst geschrieben. Die ProSet-Implementation wurde direkt aus der formalen Spezifikation aus Abschnitt 6 konstruiert. Abbildung 3 zeigt einen Teil dieser Implementation. Die Konstruktion des Prototypen wurde per Hand durchgeführt, war aber aufgrund der Gemeinsamkeiten von ProSet und Z sehr einfach.

```
module TupleSpaceManager
  export CreateTS, ExistsTS, ClearTS, RemoveTS,
         Deposit, Fetch, Meet;
  visible TSs, ActiveProcs;
begin
  TSs := {};
  ActiveProcs := {};

  procedure CreateTS (limit);
  begin
    newTS := [newat(), limit, {}];
    TSs := TSs + { newTS };
    return newTS(1);
  end CreateTS;

  ...
end TupleSpaceManager;
```

Abbildung 3: Ein Ausschnitt der ProSet-Implementation des Tupelraum-Managements.

Variablen, die in ProSet-Modulen durch `visible` deklariert werden, sind statische Variablen für Instanzen dieser Module [9]. Diese Variablen sind für alle Prozeduren im Modul sichtbar (und nur für die Prozeduren im Modul). Die Funktionen `ExistsTS` (Existenztest für Tupelräume), `ClearTS` (Löschen von Tupeln) und `RemoveTS` (Entfernen des Tupelraums) wurden bisher noch nicht erwähnt. Die Funktion `newat` liefert ein neues Atom. `T(i)` selektiert das $\mathtt{i}^{te}$ Element von einem Tupel. Die Indizierung startet bei `1`.

Der Programmzustand wird auch hier durch eine endliche Menge von Tupelräumen und eine endliche Menge von aktiven Prozessen modelliert. Tupelräume werden durch Tupel der Länge drei implementiert. Die drei Elemente entsprechen genau den drei Komponenten des Schemas *TupleSpace* aus Abschnitt 6. Die dritte Komponente implementiert die Multimenge von Tupeln, die, wie in Abbildung 3 gezeigt, zunächst leer ist. Da ProSet nicht direkt Multimengen zur Verfügung stellt, wird hier jedes Tupel als Paar abgespeichert. Solch ein Paar besteht aus dem Tupel und der Anzahl seines Vorkommens in der Multimenge. Wir verweisen auf [18] für eine genaue Diskussion der vollständigen Prototyp-Implementation. Zur Zeit werden aufbauend auf den Erfahrungen mit der Prototyp-Implementation des Laufzeitsystems Laufzeitsysteme in C entwickelt.

8 Abschlussbemerkungen

Die formale Spezifikation und die Prototyp-Implementation haben einige Inkonsistenzen und Auslassungen in der vorherigen informellen Spezifikation aufgezeigt. Diese wären bei einer konventionellen Entwicklungsmethode wahrscheinlich erst wesentlich später entdeckt worden.

Einige Bemerkungen zum Einsatz von Object-Z und ProSet für formale Spezifikationen und zum Prototyping ergeben sich aus unseren Erfahrungen im Sprachentwurf. Auf der negativen Seite ist anzumerken, daß die Werkzeugunterstützung für Object-Z noch nicht ausreichend ist. Wir haben unsere Spezifikation in [18] mit dem *f*uzz Typ-Checker [24] überprüft und alle Typfehler beseitigt. Als wesentlicher Nachteil ist uns dabei das Prinzip des *definition before use* aufgefallen, das von *f*uzz erzwungen wird, und für eine Top-Down-Präsentation der Spezifikation sehr hinderlich ist. Eine relativ einfache Verbesserung wäre es, wenn man *f*uzz eine zukünftige Definition irgendwie *ankündigen* könnte, bevor man das Definierte benutzt. So wäre immer noch eine Ordnung für die Definitionen gegeben und die Spezifikation könnte trotzdem top-down präsentiert werden. Auch die Erweiterungen von Object-Z (also die Klassenhierarchie) konnten nicht mit *f*uzz überprüft werden.

Auf der positiven Seite ist besonders anzumerken, daß der Übergang von der formalen Spezifikation in Object-Z zur Prototyp-Implementation in ProSet sehr natürlich und einfach war.

ProSet und Z scheinen generell eine gute Kombination für die Software-Entwicklung zu sein. In [6] werden beispielsweise aus Z-Spezifikationen Prototypen in Miranda and Prolog entwickelt. Mengenorientierte Programmiertechniken könnten hier eine bessere Wahl für die Konstruktion von Prototypen aus Z-Spezifikationen sein als funktionale oder logische Programmiertechniken. Notwendig wäre natürlich eine angemessene Werkzeugunterstützung für den Übergang von der formalen Spezifikation zur Prototyp-Implementation.

Literatur

[1] R. Budde, K. Kautz, K. Kuhlenkamp und H. Züllighoven. *Prototyping — An Approach to Evolutionary System Development*. Springer-Verlag, 1992.

[2] B.W. Boehm. Software engineering. *IEEE Transactions on Computers*, 25(12):1226–1241, 1976.

[3] N. Carriero und D. Gelernter. *How to write parallel programs*. MIT Press, 1990.

[4] P. Ciancarini, K.K. Jensen und D. Yanklevich. The semantics of a parallel language based on a shared dataspace. Technical Report 26/92, University of Pisa, Juli 1992.

[5] E.W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1:115–138, 1971.

[6] A. Diller. *Z: An introduction to formal methods*. Wiley, 1990.

[7] E.-E. Doberkat und D. Fox. *Software Prototyping mit SETL*. Teubner-Verlag, 1989.

[8] E.-E. Doberkat, W. Franke, U. Gutenbeil, W. Hasselbring, U. Lammers und C. Pahl. A First Implementation of ProSet. In U. Kastens und P. Pfahler, Hrsg., *International Workshop on Compiler Construction CC'92*, Seiten 23–27. Universität GH Paderborn, Informatik-Bericht Nr. 103, Oktober 1992.

[9] E.-E. Doberkat, W. Franke, U. Gutenbeil, W. Hasselbring, U. Lammers und C. Pahl. ProSet — A Language for Prototyping with Sets. In N. Kanopoulos, Hrsg., *Proc. Third International Workshop on Rapid System Prototyping*, Seiten 235–248, Research Triangle Park, NC, Juni 1992.

[10] E.-E. Doberkat, W. Franke und W. Hasselbring. Software Prototyping — Persistenz und Parallelität. In M. Nagl, Hrsg., *Proc. 16th European Congess Fair for Technical Communications (ONLINE'93), Congess VI (Software and Information Engineering)*, Seiten C622.01–C622.20, Hamburg, Februar 1993.

[11] R. Duke, P. King, G. Rose und G. Smith. The Object-Z Specification Language: Version 1. Technical Report 91-1, University of Queensland, Software Verification Research Center, Australia, Januar 1991.

[12] H.-D. Ehrich, M. Gogolla und U.W. Lipeck. *Algebraische Spezifikation abstrakter Datentypen*. Teubner-Verlag, 1988.

[13] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[14] R.W. Gray, V.P. Heuring, S.P. Levi, A.M. Sloane und W.M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–131, Februar 1992.

[15] R.H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[16] W. Hasselbring. Combining SETL/E with Linda. In G. Wilson, Hrsg., *Proc. Workshop on Linda-Like Systems and Their Implementation*, Seiten 84–99, Edinburgh, UK, Juni 1991. Edinburgh Parallel Computing Centre TR91-13.

[17] W. Hasselbring. A Formal Z Specification of ProSet-Linda. Informatik-Bericht 04-92, Universität GH Essen, September 1992.

[18] W. Hasselbring. *Prototyping Parallel Algorithms in a Set-Oriented Language*. Dissertation, Universität GH Essen, 1993. (in Vorbereitung).

[19] R. Kemmerer. Testing formal specifications to detect design errors. *IEEE Transactions on Software Engineering*, 11(1):32–43, 1985.

[20] P. Kruchten, E. Schonberg und J. Schwartz. Software prototyping using the SETL programming language. *IEEE Software*, Seiten 66–75, Oktober 1984.

[21] J.E. Narem. An informal operational semantics of C-Linda V2.3.5. Technical Report 839, Yale University, New Haven, CT, Dezember 1989.

[22] C.E. Parker. Z tools catalogue. Technical Report, British Aerospace, Warton, UK, Mai 1991.

[23] J.T. Schwartz, R.B.K. Dewar, E. Dubinsky und E. Schonberg. *Programming with Sets – An Introduction to SETL*. Springer-Verlag, 1986.

[24] J.M. Spivey. *The fuzz Manual*. Computing Science Consultancy, Oxford, UK, 2nd edition, Juli 1992.

[25] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 2nd edition, 1992.