OBSCURE: Eine Spezifikations- und Verifikationsumgebung

A 89/06

Fachbereich 14 - Informatik
Universität des Saarlandes

D - 6600 Saarbrücken

Dezember 1989

OBSCURE: Eine Spezifikations- und Verifikationsumgebung

Jacques Loeckx
FB 14 - Informatik
Universität des Saarlandes
D - 6600 Saarbrücken 11
e-mail: loeckx%fb10vax.informatik.uni-saarland.dbp.de

Das Ziel des *OBSCURE*-Projekts ist die Entwicklung einer Spezifikations- und Verifikationsumgebung. *OBSCURE* beruht auf einfachen und mathematisch soliden Grundlagen; dabei wird ein gewisser Verlust an Abstraktion im Vergleich zu "klassischen" Systemen in Kauf genommen. Das Projekt wird von der Deutschen Forschungsgemeinschaft unterstützt.

1 Einleitung

Programme werden üblicherweise modularisiert auf der Basis der *Kontrollstrukturen*. Eine solche Modularisierung führt in natürlicher Weise zu dem Begriff "Prozedur". Bei einer Modularisierung auf der Basis der *Datenstrukturen* werden die Operationen und die Datenstrukturen zu einer Einheit ("module", "cluster") zusammengeschnürt. Diese Modularisierung führt zum Begriff "abstrakter Datentyp".

Es gibt im wesentlichen drei *Methoden zur Spezifikation* abstrakter Datentypen: die algebraische, die operationelle und die konstruktive Spezifikationsmethode. Eine *algebraische* (oder *axiomatische*) Spezifikation besteht aus einer Menge von Gleichungen (oder, allgemeiner, von Axiomen). Sie ist "abstrakt", weil sie die Trägermengen und Operationen "nur" mittels ihrer Eigenschaften charakterisiert. In dem letzten Jahrzehnt ist die algebraische Spezifikationsmethode ausführlich untersucht worden (z.B. [EM85], [EGL89]). Eine *operationelle* Spezifikation besteht essentiell aus einer Typdeklaration und einigen Funktionsprozeduren in einer imperativen Programmiersprache; die Typdeklaration definiert die Trägermenge, die Funktionsprozeduren definieren die Operationen. Solche Spezifikationen werden z.B. in *CLU* [Li81] benutzt. Sie sind nur insofern abstrakt, als ihre Implementierung vor dem Benutzer "versteckt" ist. In einer *konstruktiven* Spezifikation werden die Trägermengen und die Operationen in einer "abstrakten", aber konstruktiven Art definiert. Beispiele sind [Kl84, Lo87]. Von ihrer Natur her sind

konstruktive Spezifikationen verwandt mit, aber weniger abstrakt als axiomatische Spezifikationen.

Eine *Spezifikationssprache* ermöglicht es, Spezifikationen zusammenzufügen, d.h. aus "kleinen" Spezifikationen "größere" zu bauen. Spezifikationssprachen ermöglichen deshalb den modularen Aufbau von Spezifikationen. Beispiele solcher Spezifikationssprachen sind *Clear* [BG80], *PLUSS* [Bi88], *OBJ3* [GW88], *Larch* [GWH85], die Spezifikationssprache von *OBSCURE* [LL88].

Das *OBSCURE-System* besteht aus einer Spezifikationsmethode, einer Spezifikationssprache und einer Spezifikations- und Verifikationsumgebung. Diese drei Komponenten werden nun nacheinander kurz behandelt.

2 Die algorithmische Spezifikationsmethode

Die algorithmische Spezifikationsmethode ist eine konstruktive Spezifikationsmethode. Sie ist präzise beschrieben in [Lo87]. Wir begnügen uns hier mit ein paar allgemeinen Bemerkungen und einem Beispiel.

Eine algorithmische Spezifikation führt eine Sorte und die zugehörigen Operationen ein. Die *Trägermenge* ist definiert als die Termsprache, die von den — explizit angegebenen — Konstruktoren erzeugt ist. Die *Operationen*, die keine Konstruktoren sind, werden definiert durch rekursive Programme (siehe, z.B. [LS87], Abschnitt 3.3).

Eine Spezifikation von Listen von Objekten der Sorte *element* ist:

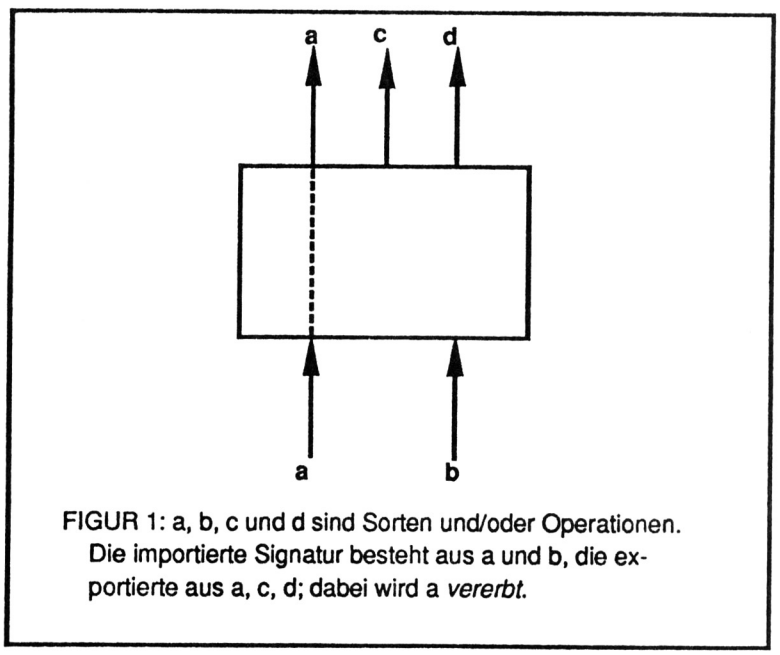
create sorts *list*
  opns constructor ε : $\rightarrow$ *list*
    constructor _._ : *list* $\times$ *element* $\rightarrow$ *list*
    $\_ \odot \_$: *list* $\times$ *list* $\rightarrow$ *list*
  programs
    $s \odot t \Leftarrow$ **case** t **of**
      $\varepsilon : s$
      $t'.e : (s \odot t').e$
    esac
endcreate

Die Trägermenge besteht aus Worten wie
ε
$\varepsilon.e$
$\varepsilon.e.e'$
...
wobei e und e' Objekte der Sorte *element* sind. Die Anwendung der Operation "." auf die Argumente $\varepsilon.e_1.e_2$ und e_3 führt einfach zu dem Wort $\varepsilon.e_1.e_2.e_3$. Die Anwendung der Operation "$\odot$" wird illustriert durch:

$(\varepsilon.e_1) \odot \varepsilon = \varepsilon.e_1$
$(\varepsilon.e_1) \odot (\varepsilon.e_2) = \varepsilon.e_1.e_2$

3 Die Spezifikationssprache von *OBSCURE*

Eine Spezifikationssprache dient dazu, einzelne Spezifikationen zusammenzusetzen. Wir präzisieren darum zuerst die Semantik einer Spezifikation.

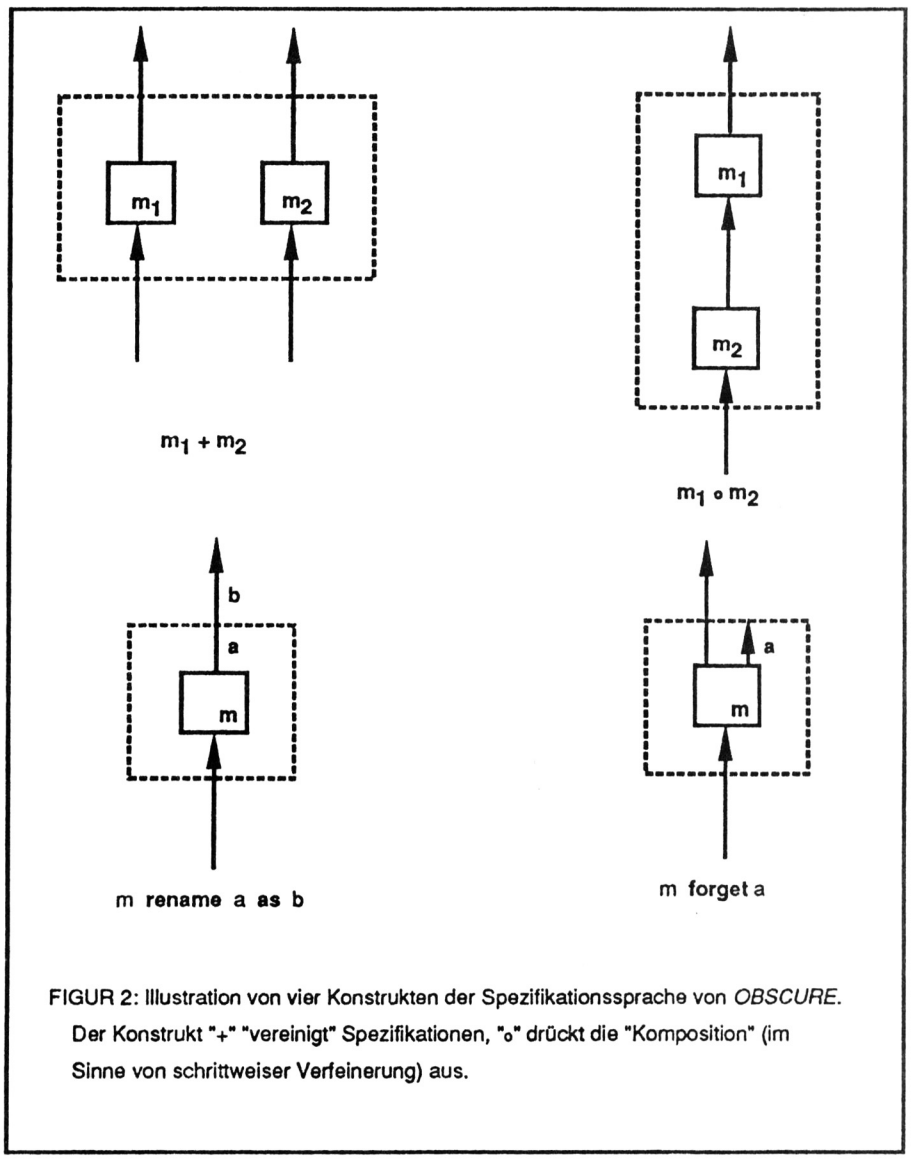


FIGUR 1: a, b, c und d sind Sorten und/oder Operationen. Die importierte Signatur besteht aus a und b, die exportierte aus a, c, d; dabei wird a *vererbt*.

In der klassischen Literatur wird eine Spezifikation interpretiert als eine Algebra. In *OBSCURE* wird eine Spezifikation interpretiert als eine Funktion, die "importierte" in "exportierte" Algebren abbildet. Etwas genauer, eine *Spezifikation* in *OBSCURE* wird syntaktisch charakterisiert durch eine *importierte* und eine *exportierte Signatur*[1]; sie wird semantisch charakterisiert durch eine Funktion, die Algebren der importierten Signatur in Algebren der exportierten Signatur abbildet, und die die folgende *Persistenzbedingung* erfüllt:

[1]Eine Signatur besteht aus Sorten und den "zugehörigen" Operationen

jede Sorte oder Operation, die in beiden Signaturen auftritt, hat in den entsprechenden Algebren dieselbe Bedeutung.



FIGUR 2: Illustration von vier Konstrukten der Spezifikationssprache von *OBSCURE*. Der Konstrukt "+" "vereinigt" Spezifikationen, "∘" drückt die "Komposition" (im Sinne von schrittweiser Verfeinerung) aus.

Im Spezifikationsbeispiel von Abschnitt 2 besteht die importierte Signatur aus der Sorte *element*, die exportierte Signatur aus den Sorten *element* und *list* sowie aus den Operationen "ϵ", "." und "$\odot$". Sei A eine importierte Algebra, d.h. eine Algebra mit einer (nicht näher definierten) Trägermenge der Sorte *element* und F die Funktion, die die Semantik des Spezifikationsbeispiels darstellt; die Persistenzbedingung drückt aus, daß die Objekte der Sorte *element* in der importierten Algebra A und in der exportierten Algebra $F(A)$ identisch sind.

Schematisch kann eine Spezifikation als ein Kästchen dargestellt werden — wie illustriert in Figur 1.

Die Spezifikationssprache von *OBSCURE* besteht aus etwa zehn Konstrukten. Einige Konstrukte werden schematisch illustriert in Figur 2. Eine formale Beschreibung der Syntax und Semantik der Spezifikationssprache findet man in [LL88]. An dieser Stelle begnügen wir uns damit, auf einige Unterschiede zu den aus der Literatur bekannten Spezifikationssprachen hinzuweisen. Die Spezifikationssprache von *OBSCURE* ist unabhängig von der benutzten Spezifikationsmethode. Weiter hängt die Semantik einer Spezifikation nicht von der "Umgebung" ab ("encapsulated semantics") und ist die Persistenzbedingung syntaktisch prüfbar. Die Spezifikationssprache besitzt Konstrukte, die explizit Teilalgebren oder Quotientalgebren erzeugen. Schließlich kann eine Spezifikation prädikatenlogische Formeln enthalten. Die Interpretation einer Spezifikation wird somit ähnlich zu der einer Hoare-Formel. Die Spezifikationssprache von *OBSCURE* hat daher eher einen "logischen" als einen "algebraischen" Charakter: eine inkonsistente Spezifikation führt nicht zum "Zusammenbruch" der Trägermengen, sondern zu *false* während der Verifikation.

Ein Beispiel einer Spezifikation in *OBSCURE* wird im nun folgenden Abschnitt skizziert.

4 Ein Spezifikationsbeispiel

Es ist nicht möglich, eine kurzes und sinnvolles Beispiel anzugeben, das die verschiedenen Konstrukte der Spezifikationssprache illustriert. Statt dessen kommentieren wir ein Beispiel, das die Philosophie des Spezifizierens in *OBSCURE* illustriert.

Das Ziel ist eine Spezifikation des Dateisystems von *UNIX*[2] oder, etwas genauer, der *UNIX*-Kommandos

cd p, **pwd**, **ls** p, **mkdir** p, **rmdir** p

Dabei ist p ein Ausdruck; sechs Beispiele für einen solchen Ausdruck sind:

usr, *name1/name2*, .. , / , */ulw*, *dir//.///*

[2] *UNIX* is a trademark of Bell Laboratories

Wir entwickeln nun die Spezifikation durch schrittweise Verfeinerung als eine Reihe von Spezifikationen, *Moduln* genannt. Der erste Modul führt — nach einem Test auf syntaktische Korrektheit — ein Kommando aus und hat sinnvollerweise den Namen *SYNTACTICALLY-CORRECT-COMMAND*:

module *SYNTACTICALLY-CORRECT-COMMAND* **is**
create
opns *Parse-and-execute*:
command × *environment*
→ *display* × *environment*
Issyncorrect: *command* → *boolean*
programs
Parse-and-execute $(c, e) \Leftarrow$
if *Issyncorrect(c)*
then *Execute(c, e)*
else ("SYNT-ERROR", *e*)
fi
Issyncorrect (c) ⇐ **case** *c* **of**
cd(p) : *Iscorrect (p)*
pwd : *true*
ls(p) : *Iscorrect (p)*
mkdir(p) : *Iscorrect(p)*
rmdir(p) : *Iscorrect(p)* **esac**
endcreate
forget opns *Issyncorrect*
endmodule

Diese Spezifikation kreiert zwei Operationen, *Parse-and-execute* und *Issyncorrect*, aber keine Sorten. Da eine der Operationen "vergessen" wird, enthält die exportierte Signatur nur die zusätzliche Operation *Parse-and-execute*. Diese Operation führt ein Kommando aus, nachdem sie — mit Hilfe der Operation *Issyncorrect* — geprüft hat, daß der Ausdruck *p* syntaktisch korrekt ist. Etwas genauer: Die Operation *Parse-and-execute* hat zwei Argumente der Sorte *command* bzw. *environment*. Das erste Argument ist das auszuführende Kommando; das zweite stellt die "Umgebung" dar, die im wesentlichen aus dem "working directory" innerhalb der "Dateistruktur" besteht. Der Wert der Operation *Parse-and-execute* ist ein Paar. Das erste Element des Paares hat die Sorte *display* und stellt die Ausgabe auf dem Bildschirm dar; falls z.B. der Ausdruck *p* syntaktisch nicht korrekt ist, erscheint die Fehlermeldung "SYNT-ERROR" auf dem Bildschirm. Das zweite Argument stellt die neue Umgebung dar. Zur Definition der Operationen *Parse-and-execute* und *Issyncorrect* wurden verschiedene Sorten und Operationen importiert: die Sorten *command, environment, display* und *boolean* sowie die Operationen *Execute*, "_", *Iscorrect, cd, pwd, ls, mkdir, rmdir* und *true*. Entsprechend der Philoso-

phie der schrittweisen Verfeinerung werden nun diese Sorten und Operationen mittels weiterer Spezifikationen eingeführt.

Eine solche Spezifikation ist:

module *COMMAND* **is**
  create
    sorts *command*
    opns **constructor** *cd* : *expression* → *command*
      constructor *pwd* : → *command*
      constructor *ls* : *expression* → *command*
      constructor *mkdir* : *expression* → *command*
      constructor *rmdir* : *expression* → *command*
  endcreate
endmodule

Diese Spezifikation kreiert (und exportiert) die Sorte *command* und die Operation *cd, pwd, ls, mkdir* und *rmdir*, die von der Spezifikation *SYNTACTICALLY-CORRECT-COMMAND* benötigt (weil importiert) werden.

Eine weitere Spezifikation exportiert die von der Spezifikation *SYNTACTICALLY-CORRECT-COMMAND* importierte Operation *Execute*:

module *EXECUTE-COMMAND* **is**
  create
    opns *Execute*:
      command × *environment*
        → *display* × *environment*
    programs *Execute (c, e)* ⇐
      if *Iscompatible(c, e)*
      then *Evaluate (c, e)*
      else (*Errormessage (c, e), e)* **fi**
  endcreate
endmodule

Diese Spezifikation importiert ihrerseits die Operationen *Iscompatible, Evaluate* und *Errormessage*. Dabei ist die Intention dieser Operationen wie folgt: *Iscompatible* "verwirft" Befehle wie

cd *nn*

falls *nn* nicht ein Sohn des "working directory" ist, *Evaluate* sorgt für die weitere Verarbeitung des Befehls und *Errormessage* erzeugt die zutreffende Fehlermeldung. Diese Intention muß natürlich noch festgelegt werden und zwar in den Spezifikationen, die diese Operationen kreieren.

Wir spezifizieren nun die Sorte *environment* :

```
module ENVIRONMENT is
        create
                sorts  environment
                opns constructor < _,_,_ >:
                                        dir-tree × path × path → environment
                     Iscorrect : environment → boolean
                     ⋮
                programs
                        Iscorrect (e) ⇐ case  e of
                                        < d,p1,p2 > :
                                                        Ispathin(p1, d)
                                                        ∧ Ispathin (p2, d)
                                                        esac
                                ⋮
        endcreate
endmodule
```

Diese Spezifikation führt u.a. die dreistellige "mixfix" Operation $< \_,\_,\_ >$ ein. In einem solchen Tripel $< d,p_1,p_2 >$ stellt die Komponente d den Dateibaum dar, p_1 der Pfad von der Wurzel zum "working directory" und p_2 der Pfad von der Wurzel zum "home directory".

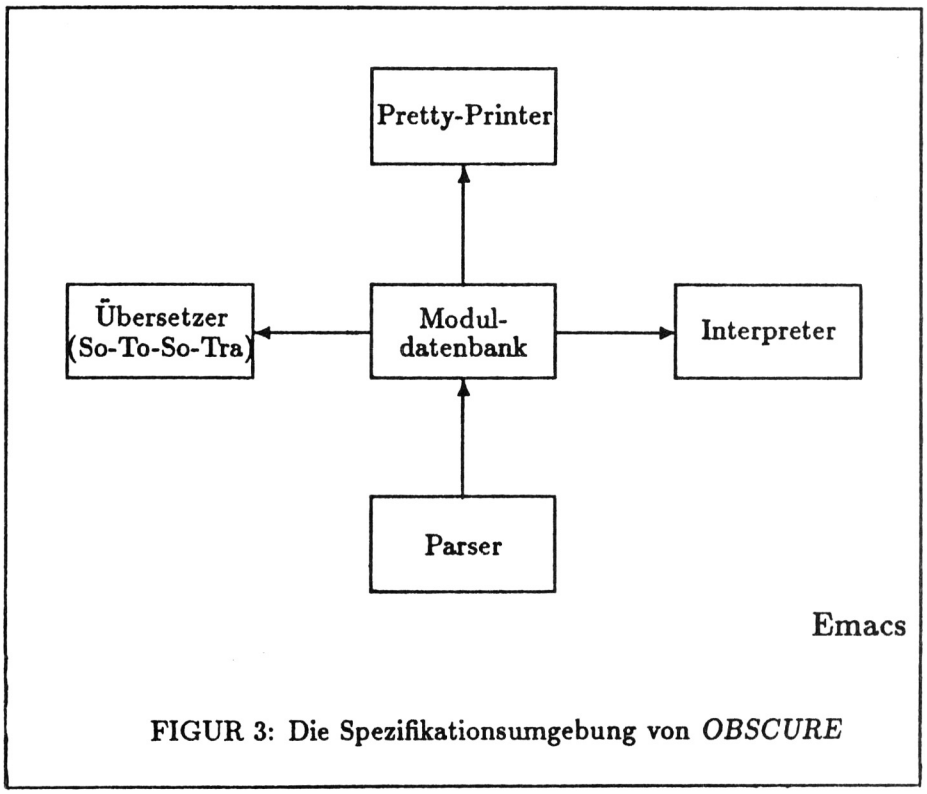
Die schrittweise Verfeinerung wird solange weitergeführt, bis nur noch "bekannte" Sorten (wie *integer* oder *boolean*) und "bekannte" Operationen (wie + oder ∧) importiert werden. In diesem Beispiel führt dies insgesamt zu etwa dreißig Spezifikationen ("Moduln"). Diese Spezifikationen werden dann mit Hilfe der Konstrukte "+" und "∘" der Spezifikationssprache zu einer einzigen Spezifikation zusammengefügt. Die importierte Signatur dieser resultierenden Spezifikation besteht nur aus "bekannten" Sorten und Operationen. Die exportierte Signatur enthält insbesondere die Operation *Parse-and-execute*, die die "Lösung" des "Problems des *UNIX*-Dateisystems" darstellt.

5 Die *OBSCURE*-Umgebung

Die *OBSCURE*-Umgebung besteht aus einer Spezifikationsumgebung und einer Verifikationsumgebung. Die Spezifikationsumgebung hilft dem Benutzer beim Entwurf von Spezifikationen. Die Verifikationsumgebung ermöglicht die Prüfung der semantischen Konsistenz der Spezifikation sowie den Beweis von Eigenschaften dieser Spezifikation. Semantische Konsistenzbedingungen treten z.B. bei der Bildung von Quotientenalgebren oder bei der Übergabe von bestimmten Parametern ("Parameter constraints") auf. Eine Eigenschaft, die man beweisen möchte, kann z.B. die Korrektheit der Implementierung einer Spezifikation durch eine andere sein. Da zur Zeit nur die Spezifikationsumge-

bung fertiggestellt ist, beschränken wir uns hier auf die Beschreibung dieses Teils der *OBSCURE*-Umgebung.

Die Spezifikationsumgebung ist ein interaktives System zum Editieren, Verwalten und Testen von Spezifikationen. Sie ist im *Emacs* eingebettet und besteht aus folgenden Komponenten (vgl. Figur 3).



FIGUR 3: Die Spezifikationsumgebung von *OBSCURE*

Der *Parser* führt die syntaktische Analyse einer Spezifikation aus und überprüft die Kontextbedingungen. Spezifikationen, die syntaktisch korrekt sind und alle Kontextbedingungen erfüllen, werden in die Moduldatenbank eingetragen.

Der "*pretty printer*" gibt den Text einer Spezifikation und/oder deren importierte und exportierte Signatur aus.

Die *Moduldatenbank* speichert Spezifikationen, deren importierte und ex-

portierte Signatur sowie deren gegenseitige Abhängigkeit. Anfragen werden in einer *SQL*-ähnlichen Sprache formuliert. Jeder Benutzer verfügt über eine persönliche Moduldatenbank. Zusätzlich gibt es noch eine gemeinsame Datenbank ("*pool*-Datenbank").

Der *Interpreter* berechnet den Wert eines vom Benutzer angegebenen Terms ("rapid prototyping").

Der *Übersetzer* übersetzt *OBSCURE*-Spezifikationen in die Programmiersprache *C++* oder *ML*. Er ermöglicht die Einbindung von *OBSCURE*-Spezifikationen in Programme, die in diesen Programmiersprachen geschrieben sind.

Bei der Konstruktion der Spezifikationsumgebung wurden verschiedene *UNIX*-Werkzeuge benutzt. So wurde der Parser mit *lex* und *yacc* generiert. Die Einträge in die Moduldatenbank werden mit *VC-ISAM* vorgenommen. Die Benutzerschnittstelle wurde in *GNU-Emacs-Lisp* realisiert.

Die Spezifikationsumgebung wurde auf einem *Sun 3/60*-Rechner entwickelt und auf die Rechner *Siemens MX2* und *VAX 8700* portiert. Der gesamte Speicherbedarf beträgt etwa 3 MByte.

Nähere Details findet man in [LP89].

6 Schlußbemerkungen

Aus den mit dem *OBSCURE*-System spezifizierten Beispielen können jetzt schon einige wichtige Folgerungen gezogen werden.

Das schwierigste Problem beim Spezifizieren besteht darin, ein "adäquates" Modell zu finden, d.h. ein Modell, das die Spezifikation einfach und durchsichtig macht (vgl. [Lo89]). Es ist z.B. wichtig, einen Unterschied zu machen zwischen einem leeren "directory" und einer (möglicherweise leeren) Datei, oder zwischen einem Pfad und dem Argument eines Kommandos wie cd oder ls.

Angesichts der großen Anzahl der eingeführten Namen ist die Benutzung einer Spezifikationsumgebung sehr hilfreich; insbesondere verliert man schnell den Überblick über die Operationen und deren Stelligkeit.

Das *UNIX*-Dateisystem wurde schon mehrfach spezifiziert und zwar mit Hilfe algebraischer (statt konstruktiver) Spezifikationsmethoden [BGM89, DA88]. Nun stellt sich heraus, daß in diesem Beispiel alle Operationen primitiv rekursiv sind. Die algebraische Spezifikation aus [BGM89] oder DA88] ist deshalb der oben skizzierten Spezifikation in *OBSCURE* sehr ähnlich. In diesem Fall bringt also der abstraktere Charakter von algebraischen Spezifikationen keinerlei Vorteile. Dafür ist in *OBSCURE* die Persistenz syntaktisch prüfbar und ist "rapid prototyping" einfach und effizient.

Zwei Aspekte von *OBSCURE* sind noch in Bearbeitung: die Verifikationsmethodologie und der Implementierungsbegriff. Zur Verifikation wurden einige wichtige Grundlagen gelegt [Tr89, Ho90]; die Realisierung der Verifi-

kationsumgebung steht aber noch aus. Der Implementierungsbegriff soll es ermöglichen, von der *Spezifikation* zum *Programm* zu kommen. Auch hier wurden Grundlagen gelegt [Le90] aber eine entsprechende Erweiterung des *OBSCURE*-Systems steht ebenfalls noch aus.

Neben dem Autor waren bzw. sind am OBSCURE-Projekt aktiv beteiligt: Jürgen Fuchs, Annette Hoffmann, Thomas Lehmann, Liane Meiss, Joachim Philippi, Ralf Treinen, Stephan Uhrig, Jörg Zeyer. Die gute Zusammenarbeit und der Arbeitseinsatz aller Beteiligten ermöglichten die bereits vorliegenden Ergebnisse. Joachim Philippi hat das Manuskript kritisch durchgelesen.

Referenzen

[BG80] Burstall, R., Goguen, J.A., The semantics of *CLEAR*, a specification language, *Proc. Advanced Course on Abstract Software Specifications, LNCS* **86** (1980), pp. 292 - 332

[BGM89] Bidoit, M., Gaudel, M.-C., Mauboussin, A., How to make algebraic specifications more understandable: An experiment with the *PLUSS* specification language, *Science of Computer Programming* **12**, *1* (1989), pp. 1 - 38

[Bi89] Bidoit, M., *PLUSS*, a language for the development of modular algebraic specifications, *These d'etat, Universite de Paris-Sud*, May 1989

[DA88] Dollin, Ch., Arnold, P., Coleman, D., Gilchrist, H., Rush, T., AXIS Tutorial: a simple filing system, *Internal Report HPL-ISC-TM-88-18*, Hewlett-Packard Ltd., Bristol (1988)

[EGL89] Ehrich, H.D., Gogolla, M., Lipeck, U.W., Algebraische Spezifikationen abstrakter Datentypen, *Teubner-Verlag*, 236 p. (1989)

[EM85] Ehrig, H., Mahr, B., Fundamentals of Algebraic Specification (Part 1), *Springer-Verlag* (1985)

[GW88] Goguen, J.A., Winkler, T., Introducing *OBJ3*, *SRI Report SRI-CSL-88-9* (1988)

[GHW85] Guttag, J., Horning, J., Wing, J., *Larch* in five easy pieces, *DEC SRC TR # 5*, 1985

[Ho90] Hoffmann, A., Eigenschaften von Modulausdrücken, Diplomarbeit, *Univ. Saarbrücken*, 1990

[Kl84] Klaeren, H.A., A constructive method for abstract algebraic software specification, *Theor. Comp. Sc.* **30**, 2 (1984), pp. 139 - 204

[Le90] Lehmann, Th., Ein Implementierungsbegriff für *OBSCURE*, Dissertation, *Univ. Saarbrücken*, 1990

[Li81] Liskov, B., et al., *CLU Reference Manual, LNCS* **114** (1981)

[LL88] Lehmann, Th., Loeckx, J., The specification language of *OBSCURE, in* Sannella, D., Tarlecki, A., (eds), *Recent Trends in Data Type Specification, LNCS* **332** (1988), pp. 131 - 153

[Lo87] Loeckx, J., Algorithmic Specifications: A Constructive Specification Method for Abstract Data Types, *TOPLAS* **9**, 4 (1987), pp. 646 - 685

[Lo89] Loeckx, J., On the use of specifications for practical problems, *Internal Report WP 89/23, Univ. Saarbrücken* (1989)

[LP88] Loeckx, J., Philippi, J., Das *OBSCURE*-Projekt, *in*: Golland, B., Paul, W.J., Schmitt, A. (Hrsg.), Innovative Informationsstrukturen, *Informatik-Fachberichte* **184**, *Springer-Verlag*, pp. 132 - 145 (1988)

[LP89] Loeckx, J., Philippi, J., Die Spezifikationsumgebung des *OBSCURE*-Systems, *Internal Report WP 89/26, Univ. Saarbrücken* (1989)

[LS87] Loeckx, J., Sieber, K., The Foundations of Program Verification (second edition), *Wiley-Teubner* (1987)

[Tr89] Treinen, R., A Logic for Algorithmic Specifications, *Internal Report WP 89/18, Univ. Saarbrücken* (1989)