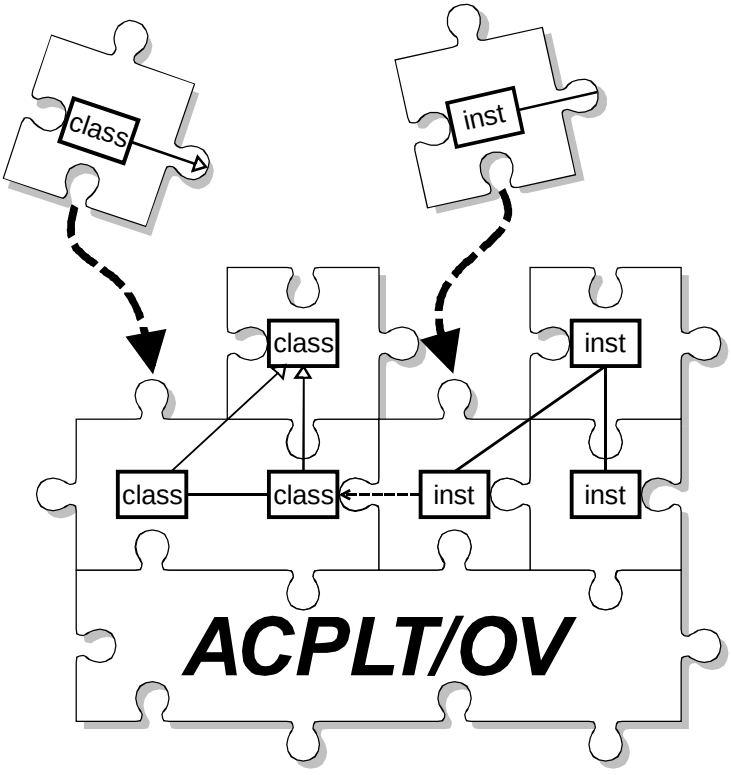
ACPLT/OV

Programmer's Guide

Version 1.0.3
08. Juli 2002



Lehrstuhl für Prozessleittechnik (PLT)
Rheinisch-Westfälische Technische Hochschule (RWTH) Aachen
D-52064 Aachen, Deutschland
Telefon +49 (0) 241 80 94339
Fax +49 (0) 241 80 92238

Inhalt

1 Dateien

1.1 Dateitypen

Der Programmierer, der in der Umgebung der Objektverwaltung ACPLT/OV arbeitet, kommt typischerweise mit folgenden Dateitypen in Berührung:

Tabelle 1: Dateitypen im Zusammenhang mit ACPLT/OV-Programmbibliotheken

Dateityp	Sprache	Beispiel	Inhalt
OV-Modell-Datei (*.ovm)	ACPLT/OV-M	*mylib*.ovm	Definitionen der Klassen und Assoziationen einer OV-Bibliothek (oder mehrerer).
OV-Datentyp-Datei (*.ovt)	ANSI C	*mylib*.ovt	Deklarationen der ANSI-C-Datentypen, die über das OV-M-Schlüsselwort `C_TYPE` in den Instanzdaten von OV-Objekten verwendet werden.
OV-Funktionsprototyp-Datei (*.ovf)	ANSI C	*mylib*.ovf	Deklarationen der ANSI-C-Funktionsprototypen von Operationen (Methoden), die über das OV-M-Schlüsselwort `C_FUNCTION` in den Methodentabellen von OV-Objekten verwendet werden.
C-Header-Datei (*.h)	ANSI C	*mylib*.h	Vom C-Code-Generator automatisch aus der entsprechenden *mylib*.ovm-Datei erzeugte ANSI-C-Deklarationen der Datenstrukturen für Instanzdaten und Methodentabellen von OV-Objekten.
		myclass.h	Vom Anwender definierte ANSI-C-Deklarationen zur Klasse *myclass*.
		myassoc.h	Vom Anwender definierte ANSI-C-Deklarationen zur Assoziation *myassoc*.
C-Quellcode-Datei (*.c)	ANSI C	*mylib*.c	Vom C-Code-Generator automatisch aus der entsprechenden *mylib*.ovm-Datei erzeugte ANSI-C-Definitionen der zum Laden einer Bibliothek benötigten Metainformationen.
		myclass.c	Implementation der Variablenzugriffe und Methoden der Klasse *myclass*.
		myassoc.c	Implementation der Methoden der Assoziation *myassoc*.
Objektcode-Datei (*.o oder *.obj)	Binärcode	*mylib*.o(bj), *myclass*.o(bj), *myassoc*.o(bj)	Vom Compiler aus der entsprechenden C-Quellcode-Datei (*.c) und den eingebundenen C-Header-Dateien (*.h) erzeugte Objektcode-Datei.
Bibliotheks-Datei (*.so oder *.dll)	Binärcode	*mylib*.so bzw. *mylib*.dll	Vom Linker aus allen Objektcode-Dateien (*.o- bzw. *.obj) erzeugte, dynamisch ladbare Bibliothek mit ausführbarem Code.
OV-Datenbasis (*.ovd)	Binärcode	*database*.ovd	Persistente Datenbasis mit allen OV-Objekten inkl. Metainformationen.
Makefile	ASCII	Makefile	Makefile zur automatischen Übersetzung.

In dieser Tabelle sind *mylib*, *myclass* und *myassoc* Platzhalter für vom Anwender vergebene Namen von Bibliotheken, Klassen oder Assoziationen.

Letztendlich vom Anwender selbst (d.h. manuell) geschrieben werden nur
- OV-Modell- (*mylib*.ovm),
- OV-Datentyp- (*mylib*.ovt),
- OV-Funktionsprototyp- (*mylib*.ovf),
- C-Header- (*.h),
- C-Quellcode-Dateien (*.c) und
- das Makefile, das zumindest an die Bedürfnisse angepaßt wird.

Nicht von Hand geschrieben, sondern automatisch generiert werden
- die vom C-Code-Generator erzeugten C-Header- und C-Quellcode-Dateien (*mylib*.c und *mylib*.h),
- die vom Compiler erzeugten Objektcode-Dateien (*.o bzw. *.obj) und
- die vom Linker erzeugten Bibliotheks-Dateien (*mylib*.so bzw. *mylib*.dll).

1.2 Empfohlene Verzeichnisstruktur

Folgende Verzeichnisstruktur wird empfohlen:

```
mylib
  +-- build
  |     +-- linux    (enthält Linux-Makefile und generierte Dateien)
  |     +-- nt       (enthält Windows-NT-Makefile und generierte Dateien)
  |     +-- ...
  +-- model          (enthält mylib.ovm, mylib.ovt und mylib.ovf)
  +-- include
  |     +-- mylib    (enthält Anwender-definierte C-Header-Dateien)
  +-- source         (enthält C-Implementierungen der Klassen und Assoziationen)
```

1.3 Aufbau der Quelldateien

Hier soll beispielhaft gezeigt werden, wie die Quelldateien intern aufgebaut sind.

1.3.1 OV-Modell-Datei

```
/*
*   File: mylib.ovm
*/

#include "ov.ovm"

LIBRARY mylib
  VERSION = "1.0";
  AUTHOR = "Dirk Meyer";
  COPYRIGHT = "Copyright (C) 2000 PLT, RWTH Aachen";
  COMMENT = "My first OV library";

  /* Classes */
```

```
  CLASS myclass1 : CLASS ov/object
    IS_INSTANTIABLE;
    IS_FINAL;
    COMMENT = "My first OV class";
    VARIABLES
      temperature : SINGLE HAS_ACCESSORS UNIT = "C";
      myvar1[] : STRING HAS_GET_ACCESSOR COMMENT = "text list";
      myvar2 : C_TYPE <MYLIB_MYDATATYPE>;
      ...
    END_VARIABLES;
    OPERATIONS
      constructor : C_FUNCTION <OV_FNC_CONSTRUCTOR>;
      mymethod : C_FUNCTION <MYLIB_MYPROTOTYPE>;
      ...
    END_OPERATIONS;
  END_CLASS;

  CLASS myclass2 : CLASS ...
  END_CLASS;
  ...

  /* Associations */

  ASSOCIATION myassoc : ONE_TO_MANY
    PARENT myparent : CLASS mylib/myclass1;
    CHILD  mychild  : CLASS mylib/myclass2;
  END_ASSOCIATION;

  ASSOCIATION ...
  END_ASSOCIATION;
  ...
END_LIBRARY;
```

1.3.2 OV-Datentyp-Datei

```
/*
*   File: mylib.ovt
*/
#ifndef MYLIB_OVT_INCLUDED
#define MYLIB_OVT_INCLUDED

#include <stdio.h>

typedef struct {
  int   a;
  float b;
  FILE  *fp;  /* from stdio.h */
} MYLIB_MYDATATYPE;

#endif /* MYLIB_OVT_INCLUDED */
```

1.3.3 OV-Funktionsprototyp-Datei

```
/*
*   File: mylib.ovf
*/
#ifndef MYLIB_OVF_INCLUDED
#define MYLIB_OVF_INCLUDED

typedef OV_DLLFNCEXPORT void MYLIB_MYPROTOTYPE(
```

```
  OV_INSTPTR_mylib_myclass1 pobj, int a, float *pb
);

#endif /* MYLIB_OVF_INCLUDED */
```

1.3.4 C-Header-Datei

```
/*
*   File: myclass1.h
*/
#ifndef MYLIB_MYCLASS1_H_INCLUDED
#define MYLIB_MYCLASS1_H_INCLUDED

/* minimum temperature in Celsius */
#define TEMPERATURE_MIN (-273.14)

#endif /* MYLIB_MYCLASS1_H_INCLUDED */
```

1.3.5 C-Quellcode-Datei einer Klasse

```
/*
*   File: myclass1.c
*/

#include "mylib.h"          /* possibly already part of myclass1.h */
#include "mylib/myclass1.h"
#include "libov/ov_macros.h"

/* Variable accessor functions */

OV_DLLFNCEXPORT OV_SINGLE mylib_myclass1_temperature_get(
  OV_INSTPTR_mylib_myclass1 pthis
) {
  /* get the value */
  return pthis->v_temperature;
}

OV_DLLFNCEXPORT OV_RESULT mylib_myclass1_temperature_set(
  OV_INSTPTR_mylib_myclass1 pthis, const OV_SINGLE value
) {
  /* check temperature range */
  if (value < TEMPERATURE_MIN) return OV_ERR_BADVALUE;
  /* set the value */
  pthis->v_temperature = value;
  return OV_ERR_OK;
}

OV_DLLFNCEXPORT OV_STRING *mylib_myclass1_myvar1_get(
  OV_INSTPTR_mylib_myclass1 pthis, OV_UINT *pveclen
) {
  /* get the value */
  *pveclen = pthis->v_myvar1.veclen;
  return &pthis->v_myvar1.value;
}

/* Operations (virtual methods) */

OV_DLLFNCEXPORT OV_RESULT mylib_myclass1_constructor(
  OV_INSTPTR_ov_object pobj
) {
  /* local variables */
```

```
  OV_INSTPTR_mylib_myclass1 pthis;
  OV_RESULT                 result;
  /* get full instance pointer to see all variables */
  pthis = Ov_StaticPtrCast(mylib_myclass1, pobj);
  /* do the actual construction */
  result = ov_object_constructor(pobj);
  if(Ov_Fail(result)) {
    return result;
  }
  /* further construction things */
  ...
  return OV_ERR_OK;
}

OV_DLLFNCEXPORT void mylib_myclass1_mymethod(
  OV_INSTPTR_mylib_myclass1 pthis, int a, float *pb
) {
  ...
}

/* other methods/functions associated with the class (non-virtual) */

OV_DLLFNCEXPORT void mylib_myclass1_foo(
 OV_INSTPTR_mylib_myclass1 pthis
) {
  ...
}
```

1.3.6 C-Quellcode-Datei einer Assoziation

```
/*
*   File: myassoc.c
*/

#include "mylib.h"
#include "libov/ov_association.h"
#include "libov/ov_macros.h"

/* default implementations */

OV_IMPL_LINK(mylib_myassoc)

OV_IMPL_UNLINK(mylib_myassoc)

/* our own implementation */

OV_DECL_GETACCESS(mylib_myassoc) {
  /* only grant read access, no right to link/unlink from */
  /* outside the implementation of the two associated classes */
  return OV_AC_READ;
}
```

2 Werkzeuge

Zum Entwickeln und anschließenden Verwenden von ACPLT/OV-Bibliotheken wird eine Reihe von Werkzeugen benötigt. Neben einem C-Compiler (z.B. dem GNU-C-Compiler GCC, der nicht nur für Linux, sondern auch für Windows frei verfügbar erhältlich ist) und ggf. einem Makefile-Interpreter (z.B. GNU make, ebenfalls frei verfügbar) sind einige ACPLT-Werkzeuge notwendig.

2.1 Der OV-C-Code-Generator

Mit dem OV-C-Code-Generator werden aus den OV-Modell-Dateien *mylib*.ovm entsprechende C-Header- und C-Quellcode-Dateien *mylib*.h und *mylib*.c erzeugt. Diese Dateien enthalten alles was notwendig ist, um auf die Instanzdaten von Objekten und deren Methodentabellen zuzugreifen und um eine Modell-Bibliothek (*mylib*.so bzw. *mylib*.dll) zur Laufzeit in das OV-System zu laden.

Aufruf und Kommandozeilenparameter lauten wie folgt:

```
Usage: ov_codegen [arguments]

The following arguments are mandatory:
-f FILE, --file FILE            Set model filename (*.ovm), you may use
                                stdin

The following arguments are optional:
-o PATH, --output-path PATH     Set path for output files (*.h and *.c)
-I PATH, --include-path PATH    Set include path, may be used multiple
                                times
-l LIBRARY, --library LIBRARY   Only create output for this library
-d, --debug-mode                Turn on parser debug mode
-v, --version                   Display version information
-h, --help                      Display this help message
```

2.2 Das OV-Datenbasis-Utility

Mit dem OV-Datenbasis-Utility können neue (bis auf das ACPLT/OV-Metamodell leere) Datenbasen erzeugt werden und Informationen zu vorhandenen Datenbasen angezeigt werden.

Aufruf und Kommandozeilenparameter lauten wie folgt:

```
Usage: ov_dbutil [arguments]

The following optional arguments are available:
-f FILE, --file FILE            Set database filename (*.ovd)
-c SIZE, --create SIZE          Create a new database
-l LOGFILE, --logfile LOGFILE   Set logfile name, you may use stdout or
                                stderr
-v, --version                   Display version information
-h, --help                      Display this help message
```

Beim Erzeugen einer Datenbasis mit der Option „-c“ muß die Größe der Datenbasis in Bytes angegeben werden. Unter 64 kByte sind auf gar keinen Fall sinnvoll, da das Metamodell von OV, das in jedem Falle in die Datenbasis geladen wird, schon ca. 27 kByte groß ist. Unter

manchen Systemen kann die Datei nachträglich noch automatisch wachsen, z.B. unter Linux. Dies kann jedoch nicht in jedem Fall garantiert werden.

Der Quelltext des OV-Datenbasis-Utilities liegt in der Datei ov/source/server/ov_server.c und kann als Vorlage zur Erstellung eigener Utility-Programme verwendet werden, die beispielsweise beim Erstellen einer neuen Datenbasis bestimmte Bibliotheken automatisch laden und bereits bestimmte Objekte instanziieren.

2.3 Der OV-Server

Der OV-Server enthält ein Main-Programm, das die Laufzeit-Umgebung von ACPLT/OV auf einer gegeben Datenbasis startet und einen integriertem ACPLT/KS-Server besitzt. Die KS-Anbindung basiert auf der shared library libovks.so bzw. auf der DLL libovks.dll, die auch KS-Klientenfunktionalitäten bereitstellt. Für den Betrieb muß das ACPLT/KS-Manager sowie der ONC/RPC Portmapper laufen (siehe ACPLT/KS-Dokumentation).

Aufruf und Kommandozeilenparameter lauten wie folgt:

```
Usage: ov_server [arguments]

The following optional arguments are available:
-f FILE, --file FILE            Set database filename (*.ovd)
-b FILE, --backup FILE          Set backup database filename (*.ovd)
-s SERVER, --server-name SERVER Set server name
-i ID, --identify ID            Set Ticket Identification for server access
-p PORT, --port-number PORT     Set server port number
-l LOGFILE, --logfile LOGFILE   Set logfile name, you may use stdout or
                                stderr
-a , --activity-lock            Locks OV activities
-r , --reuse-address            Reuses the socket address/port
-n, --no-startup                Do not startup the database
-v, --version                   Display version information
-h, --help                      Display this help message
```

2.4 Der ACPLT Magellan

Der ACPLT Magellan ist ein generischer ACPLT/KS-Klient, der es gestattet, den Inhalt eines KS-Servers remote zu untersuchen und zu konfigurieren. Im Falle eines OV-Servers kann der Inhalt der entsprechenden OV-Datenbasis erkundet werden (Objekte mit deren Variablen und Links, Klassen und Assoziationen und sogar das OV-Metamodell):

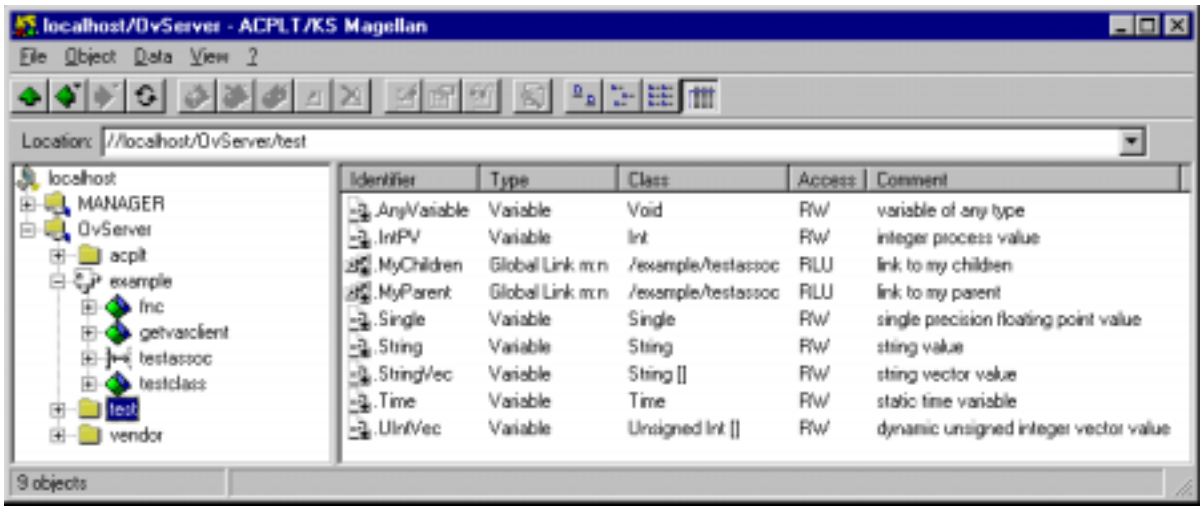


Bild 2.1: Einblick in eine OV-Datenbasis mit dem Magellan.

Darüber hinaus können mit dem ACPLT Magellan neue OV-Bibliotheken dynamisch geladen werden und Instanzen der geladenen Klassen erzeugt und verknüpft werden. Um die dem OV beiliegende Beispiel-Bibliothek „example“ zu laden, muß ein Objekt der Klasse „ov/library“ und dem Namen „example“ erzeugt werden, beispielsweise:

Bild 2.2: Laden der Bibliothek „example“ mit dem Magellan.

Voraussetzung ist, daß das System die shared library example.so bzw. DLL example.dll überhaupt finden kann. Dazu muß die Umgebungsvariable LD_LIBRARY_PATH (Linux) bzw. PATH (Windows) entsprechend gesetzt sein. Nach dem Laden der Bibliothek stehen die Klassen der Bibliothek zur Verfügung und können auf analoge Art und Weise instanziiert werden.

Für weitere Informationen sei auf die Dokumentation des ACPLT Magellan verwiesen.

3 Der Übersetzungsvorgang

Üblicherweise wird der Übersetzungsvorgang automatisch durch ein sogenanntes Makefile gesteuert. Empfohlen wird die Verwendung von GNU make, das eine reichhaltige Funktionalität besitzt und für fast alle Plattformen, insbesondere für Unix-Systeme und Windows, verfügbar ist.

3.1 C-Code-Generierung

Erster Schritt bei der Übersetzung ist das Erzeugen der C-Header- und C-Quellcode-Dateien aus der OV-Modell-Datei *mylib*.ovm mit dem C-Code-Generator. Erzeugt werden nicht nur die Dateien *mylib*.h und *mylib*.c für die Bibliothek *mylib* selbst, sondern auch die Dateien für die Bibliotheken, auf denen diese Bibliothek aufbaut, mindestens also auch die Dateien ov.h und ov.c. Dazu müssen dem Codegenerator neben der zu übersetzenden *mylib*.ovm-Datei die OVM-Include-Pfade, d.h. die Verzeichnisse bekannt gegeben werden, in denen die OV-Modell-Dateien (*.ovm) liegen, die direkt oder indirekt per #include-Direktive eingebunden werden (mindestens also des Verzeichnisses, das die Dateien ov.ovm, ov.ovt und ov.ovf enthält). Ein typischer Aufruf könnte so aussehen (auf Leerzeichen achten):

```
ov_codegen -f ../../model/mylib.ovm -I ../../../ov/model
```

3.2 Compilieren der C-Quellcode-Dateien

Anschließend müssen alle C-Quellcode-Dateien (*.c), d.h. die generierte Datei *mylib*.c sowie alle C-Quellcode-Dateien aus dem Verzeichnis *mylib*/source, mit einem ANSI-C-Compiler in entsprechende Objektcode-Dateien (*.o bzw. *.obj) übersetzt werden. Dies geschieht mit einem Compiler-Aufruf pro C-Quellcode-Datei. Dazu müssen dem Compiler neben der aktuellen C-Quellcode-Datei und den Compiler-spezifischen Schaltern die C-Include-Pfade sowie bestimmte C-Defines bekannt gegeben werden.

Zu den Include-Pfaden gehören

- das aktuelle Verzeichnis, in dem sich auch die vom C-Code-Generator erzeugten C-Header-Dateien befinden,
- das eigene Include-Verzeichnis (z.B. ../../include),
- die Verzeichnisse mit den OV-Datentyp- (*.ovt) und OV-Funktionsprototyp-Dateien (*.ovf),
- das Include-Verzeichnis von ACPLT/OV (z.B. ../../../ov/include) und von ACPLT/KS (z.B. ../../../plt/include und ../../../ks/include),
- unter Windows auch das Verzeichnis von ONC/RPC (z.B. ../../../../rpc).

Darüber hinaus können Include-Pfade für andere einzubindenden Bibliotheken notwendig sein.

Notwendige Defines sind

- für den Fall, daß Debugging-Makros von ACPLT/OV zu aktiviert werden sollen, das optionale Define OV_DEBUG,
- die OV- und KS-Defines OV_SYSTEM_*XXX*=1 und PLT_SYSTEM_*XXX*=1, wobei *XXX* für die verwendete Systemplattform, also z.B. LINUX oder NT steht, und
- das Define OV_COMPILE_LIBRARY_mylib (Groß-/Klein-Schreibung unbedingt konsequent beachten).

Für den GNU-C-Compiler GCC sieht ein solcher Compiler-Aufruf beispielsweise wie folgt aus:

```
gcc -g -Wall -O2 -shared -I. -I../../include -I../../../ov/model -I../../mo
del -I../../../ov/include/ -I../../../plt/include/ -I../../../ks/include –D
OV_DEBUG -DOV_SYSTEM_LINUX=1 -DPLT_SYSTEM_LINUX=1 -DOV_COMPILE_LIBRARY_myli
b -c ../../source/myclass.c -o myclass.o
```

3.3 Der Link-Vorgang

Abschließend werden die beim Kompilieren entstandenen Objektcode-Dateien (*.o bzw. *.obj) zu einer dynamisch ladbaren Bibliothek (shared library oder DLL, *.so bzw. *.dll) zusammengebunden („gelinkt“).

Für den GNU-C-Compiler GCC sieht ein solcher Linker-Aufruf beispielsweise wie folgt aus:

```
gcc -shared -o mylib.so mylib.o myclass1.o myclass2.o myassoc.o
```

Unter Windows müssen zusätzlich die Import-Bibliothek-Dateien aller vorausgesetzten Bibliotheken (*.lib) hinzugebunden werden. Wie dies geschieht entnehmen Sie bitte der Dokumentation Ihres Compilers.

3.4 Ein Beispiel-Makefile

Im folgenden finden Sie ein Beispiel-Makefile für Linux mit dem GNU-C-Compiler GCC. Das Makefile ist für GNU make geschrieben und wird durch den Aufruf

```
make
```

gestartet (entspricht „make all“, siehe unten).

Im Makefile sind folgende Targets definiert:

- depend – generiere die Abhängigkeiten zwischen C-Quellcode- und C-Header-Dateien.
- all – alles bauen.
- clean – Verzeichnis aufräumen (löscht automatisch generierte Dateien).
- install – kopiert die notwendigen Dateien in das plt/bin/linux-Verzeichnis.

Ein vollständige make-Aufruf-Sequenz wäre also

```
make depend
make all
make install
make clean
```

oder kurz

```
make depend all install clean
```

Für nähere Informationen schauen Sie bitte in der Dokumentation ihres Make-Interpreters nach.

Das eigentliche Makefile sieht nun in etwa wie folgt aus:

```
#
# File: Makefile
#

# Directories

ACPLT_DIR             = ../../../../
PLT_BIN_DIR           = $(ACPLT_DIR)bin/

MYLIB_DIR             = ../../
MYLIB_MODEL_DIR       = $(MYLIB_DIR)model/
MYLIB_INCLUDE_DIR     = $(MYLIB_DIR)include/
MYLIB_SOURCE_DIR      = $(MYLIB_DIR)source/

ACPLT_OV_DIR          = $(ACPLT_DIR)base/ov/
OV_INCLUDE_DIR        = $(ACPLT_OV_DIR)include/
OV_MODEL_DIR          = $(ACPLT_OV_DIR)model/

ACPLT_PLT_DIR         = $(ACPLT_DIR)base/plt/
ACPLT_PLT_INCLUDE_DIR = $(ACPLT_PLT_DIR)include/

ACPLT_KS_DIR          = $(ACPLT_DIR)base/ks/
ACPLT_KS_INCLUDE_DIR  = $(ACPLT_KS_DIR)include/

# Includes and Defines

MYLIB_INCLUDES = \
    -I$(MYLIB_MODEL_DIR) \
    -I$(MYLIB_INCLUDE_DIR)

MYLIB_DEFINES = \
    -DOV_DEBUG \
    -DOV_COMPILE_LIBRARY_mylib

OV_INCLUDES = \
    -I$(OV_INCLUDE_DIR) \
    -I$(OV_MODEL_DIR) \
    -I$(ACPLT_PLT_INCLUDE_DIR) \
    -I$(ACPLT_KS_INCLUDE_DIR)

OV_DEFINES = \
    -DOV_SYSTEM_LINUX=1 \
    -DPLT_SYSTEM_LINUX=1

INCLUDES = $(OV_INCLUDES) $(MYLIB_INCLUDES) -I.

DEFINES  = $(OV_DEFINES) $(MYLIB_DEFINES)

# Targets and their sources

MYLIB_SRC := mylib.c $(wildcard $(MYLIB_SOURCE_DIR)*.c)
MYLIB_OBJ = $(foreach source, $(MYLIB_SRC), \
            $(basename $(notdir $(source))).o)
MYLIB_DLL = mylib.so

ALL       = $(MYLIB_DLL)
SOURCES   = $(MYLIB_SRC)

# Compiler
```

```
CC         = gcc
CC_FLAGS   = -g -Wall -O2 -shared
COMPILE_C  = $(CC) $(CC_FLAGS) $(DEFINES) $(INCLUDES) -c
LD         = $(CC) -shared
OV_CODEGEN = ov_codegen

# Targets

all: $(ALL)

# Dependencies

.depend:
    touch $@

depend : $(SOURCES)
    $(COMPILE_C) -MM $^ > .depend

# Implicit Rules

.SUFFIXES:
.SUFFIXES: .c .h .ovm .o .so

SOURCE_DIRS = $(MYLIB_SOURCE_DIR)
VPATH       = $(SOURCE_DIRS) $(OV_MODEL_DIR) $(MYLIB_MODEL_DIR) .

.c.o:
    $(COMPILE_C) $< -o $@

.ovm.c:
    $(OV_CODEGEN) -I $(OV_MODEL_DIR) -f $<

.ovm.h:
    $(OV_CODEGEN) -I $(OV_MODEL_DIR) -f $<

# mylib library

$(MYLIB_DLL) : $(MYLIB_OBJ)
    $(LD) -o $@ $^

mylib.ovm : ov.ovm

# Install

install : all
    @echo Installing files to '$(PLT_BIN_DIR)'
    @-cp $(ALL) $(PLT_BIN_DIR)
    @echo Done.

# Clean up

clean :
    @echo Cleaning up...
    @-rm -f *.c *.h *.o *.so
    @echo Done.

# Include dependencies

include .depend
```

4 Export und Import bei Bibliotheken

Dynamische Bibliotheken – seien es shared libraries oder dynamic link libraries (DLLs) – sind Module, welche die darin enthaltenen Funktionen und/oder globalen Variablen anderen Modulen zur Verfügung stellen können (Export) oder umgekehrt Funktionen und/oder globalen Variablen anderer Module nutzen (Import).

Zum Import von Funktionen aus einer anderen DLL unter Windows müssen beim Link-Vorgang zusätzlich die Import-Bibliothek-Dateien der entsprechenden Bibliotheken (*.lib) hinzugebunden werden. Wie dies geschieht entnehmen Sie bitte der Dokumentation Ihres Compilers.

4.1 Export und Import von Funktionen

Der Export einer Funktion wird dem Compiler/Linker durch das Makro `OV_DLLFNC-EXPORT` bekannt gegeben, das einen entsprechenden Modifizierer enthält. Darauf folgt die eigentliche Funktionsdeklaration bzw. -definition (Implementierung).

C-Header-Datei:

```
/* Declaration */
OV_DLLFNCEXPORT void *foo(int a, float *pb);
```

C-Quellcode-Datei:

```
/* Definition (implementation) */
OV_DLLFNCEXPORT void *foo(int a, float *pb) {
  void *p;
  ...
  return p;
}
```

Beim Import einer Funktionen ist nichts weiter zu beachten, als daß der Funktionsprototyp wie üblich deklariert worden sein muß. Diese Deklaration darf ebenfalls den Modifizierer `OV_DLLFNCEXPORT` enthalten, so daß dieselbe C-Header-Datei für die Export- und die Import-Deklaration verwendet werden kann.

4.2 Export und Import von globalen Variablen

Der Ex- bzw. Import einer globalen Variable wird dem Compiler/Linker durch das Makro OV_DLLVAREXPORT bzw. OV_DLLVARIMPORT bekannt gegeben, das jeweils einen entsprechenden Modifizierer enthält. Darauf folgt die eigentliche Variablendeklaration bzw. -definition (Implementierung).

C-Quellcode-Datei:

```
/* export of a global variable implemented in this source code file */
OV_DLLVAREXPORT OV_STRING text = "exported string";

...

/* import of a global variable implemented in another library */
```

```
OV_DLLVARIMPORT OV_INT integer;
```

Komplizierter wird es, wenn eine globale Variable in einer gemeinsam für Import und Export genutzten C-Header-Datei deklariert werden soll. In diesem Falle muß durch ein Define bekannt gegeben werden, ob die C-Header-Datei von einer C-Quellcode-Datei eingebunden wird, die die Variable implementiert und exportiert, oder sie von einer C-Quellcode-Datei eingebunden wird, die die Variable importiert. Für OV-Bibliotheken geschieht dies durch das Define OV_COMPILE_LIBRARY_*mylib* (vgl. Abschnitt 3.2, „Compilieren der C-Quellcode-Dateien“).

Gemeinsame C-Header-Datei:

```
/*
*   File:    myfile.h
*   Library: mylib
*/
#ifdef OV_COMPILE_LIBRARY_mylib
extern          OV_SINGLE value;  /* implement and export "value" */
#else
OV_DLLVARIMPORT OV_SINGLE value; /* import "value" */
#endif
```

Implementierende und exportierende C-Quellcode-Datei:

```
/*
*   File:    myfile.c
*   Library: mylib
*/

/* make sure, that OV_COMPILE_LIBRARY_mylib is defined */
/* (you should provide this define in the Makefile when calling */
/* the compiler, so the following three lines are optional) */
#ifndef OV_COMPILE_LIBRARY_mylib
#define OV_COMPILE_LIBRARY_mylib
#endif

/* now include header files */
#include "myfile.h"

OV_DLLVAREXPORT OV_SINGLE value = 3.141;
```

Importierende C-Quellcode-Datei (Teil einer anderen Bibliothek):

```
/*
*   File:    anotherfile.c
*   Library: anyotherlib
*/

/* make sure, that OV_COMPILE_LIBRARY_mylib is NOT defined */
/* (you should make sure this is NOT defined by the compiler call */
/* in your Makefile, so the following three lines are optional) */
#ifdef OV_COMPILE_LIBRARY_mylib
#error File is not part of "mylib", do not define OV_COMPILE_LIBRARY_mylib!
#endif

/* now include header files */
#include "myfile.h"
#include <stdio.h>
```

```
/* you may now access the variable */
void print_value(void) {
  printf("value = %f\n", value);
}
```

5 Debugging und Logfiles

5.1 Debugging-Makros

Für den Fall, daß bei Compilieren das Define `OV_DEBUG` gesetzt ist, werden mit den folgenden Makros Debugging-Ausgaben in das Logfile geschrieben bzw. auf dem Bildschirm ausgegeben. Wird es später nicht mehr definiert, so werden die Makros „leer“ kompiliert, d.h. behandelt, als wären sie nicht da.

Beispiel:

```
#include "libov/ov_debug.h"

/* Print debugging information, warnings or errors */

Ov_Info("here we go!");
Ov_Warning("hmm, this doesn‘t look good!");
Ov_Error("shxx!");
Ov_WarnIf(x == 0);
Ov_WarnIfNot(pthis); /* Warn if the "this" pointer is NULL */

/* stop execution if a serious problem has occurred */

Ov_AbortIf(stressfrei != 2000);
Ov_AbortIfNot(stressfrei == 2000);
```

5.2 Logfiles

Grundsätzlich dienen Logfiles dazu, nachzuhalten, was im Verlauf der Zeit mit dem OV-System und seine Anwendungen passiert ist. Es ist möglich, in Dateien, System-Logs oder auf den Bildschirm zu schreiben. Typischerweise wird dies beim Start der Applikation über die Kommandozeilenparameter konfiguriert. Dies aber auch zur Laufzeit geschehen. Siehe dazu die LibOV API-Referenz.

Beispiel:

```
#include "libov/ov_logfile.h"

ov_logfile_info("system started");
ov_logfile_warning("could not load driver");
ov_logfile_error("Error: %s", ov_result_getresulttext(result));
ov_logfile_alert("a system alert");
ov_logfile_debug("Oups!");
```

6 Bibliotheken

6.1 Definition einer Bibliothek in der OV-Modell-Datei

Die Definition einer Bibliothek sieht wie folgt aus:

```
LIBRARY mylib
  VERSION   = "1.0";
  AUTHOR    = "Dirk Meyer";
  COPYRIGHT = "Copyright (C) 2000 PLT, RWTH Aachen";
  COMMENT   = "My ACPLT/OV Library";

  CLASS ...
  END_CLASS;

  ...

  ASSOCIATION ...
  END_ASSOCIATION;

  ...

END_LIBRARY;
```

Eine Bibliothek enthält neben einigen beschreibenden Angaben (Version, Autor, Copyright und Kommentar) die Klassen und Assoziationen der Bibliothek. Diese Informationen werden beim Laden der Bibliothek in die Datenbasis eingetragen.

Wie das Laden einer Bibliothek mit dem ACPLT Magellan geschieht, ist in Abschnitt 2.4, „Der ACPLT Magellan“ beschrieben.

7 Klassen und Objekte

7.1 Definition einer Klasse in der OV-Modell-Datei

Die Definition einer Klasse in einer Bibliothek sieht wie folgt aus:

```
LIBRARY mylib

  ...

  CLASS myclass : CLASS anylib/anybaseclass
    IS_INSTANTIABLE;
    IS_FINAL;
    COMMENT = "ACPLT/KS client sending GetVar requests";
    FLAGS = "abcxyz";
    VARIABLES ...
    END_VARIABLES;
    PARTS ...
    END_PARTS;
    OPERATIONS ...
    END_OPERATIONS;

END_CLASS;

  ...

END_LIBRARY;
```

Eine Klasse enthält die Definition der Instanzvariablen (`VARIABLES`), seiner eingebetteten Objekte (`PARTS`) und seiner Operationen (`OPERATIONS`). Über das Schlüsselwort `IS_INSTANTIABLE` wird vereinbart, ob Instanzen (Objekte) aus dieser Klasse gebildet werden können. Dies ist nur möglich, wenn die Klasse keine abstrakten Operationen mehr enthält. Durch das Schlüsselwort `IS_FINAL` kann festgelegt werden, daß die Klasse nicht als Oberklasse anderer Klassen herangezogen werden darf, also von ihr nicht abgeleitet werden soll. Darüber hinaus kann ein Kommentar und semantische Flags angegeben werden.

7.2 Repräsentation von Klassen und Objekten in ANSI C

Da Klassen und Objekte in ANSI C implementiert werden, ANSI C aber keine Klassen kennt, müssen diese in C emuliert werden. Dies geschieht mit Hilfe von Datenstrukturen für Instanzdaten und Methodentabellen. Insbesondere die Typkonvertierung zwischen Ober- und Unterklassen ist problematisch, da sie nicht vom C-Compiler durchgeführt werden kann.

Hier wurde tief in die Trickkiste gegriffen, um Typkonvertierung zu ermöglichen, aber dennoch zur Compile-Zeit eine gewisse Typsicherheit zu erreicht. Die eigentliche „Magie“ bei der Definition der Datenstrukturen für Instanzdaten und Methodentabellen von Klassen in ANSI C wird durch den C-Code-Generator durchgeführt und ist daher für den Anwender transparent. Im Quelltext allerdings ist seine Mithilfe gefragt, wenn z.B. virtuelle Methoden über die Methodentabelle aufgerufen werden sollen oder Typkonvertierungen durchgeführt werden sollen. ACPLT/OV stellt hierzu eine Vielzahl an Funktionen und Makros (siehe libov/ov_macros.h) zur Verfügung.

7.2.1 Unterschiede – OV vs. C++

Auf folgende Unterschiede zwischen ANSI-C-Implementierungen in OV und C++-Implementierungen sei besonders hingewiesen:

- In OV gibt es keine Mehrfachvererbung wie in C++, sondern nur Einfachvererbung.
- Jede Klasse muß in OV (genau) eine Oberklasse besitzen und damit zumindest von der Top-Level-Klasse „ov/object“ erben (ggf. auch von „ov/domain“, wenn das Objekt Kind-Objekte besitzen können soll).
- Der Konstruktor wird in OV wie eine virtuelle Methode behandelt. Beim Überladen muß der Konstruktor der Oberklasse von Hand aufgerufen werden. (Gleiches gilt für den Destruktor.)
- Der Konstruktor darf in OV fehlschlagen (Rückgabewert ungleich `OV_ERR_OK`).
- Der Konstruktor bekommt in OV keine Initialisierungsparameter. Vergleiche dazu den Abschnitt 7.5, „Erzeugen und Löschen von Objekten“.
- Alle Instanzvariablen und Methoden unter OV sind öffentlich sichtbar, da in ANSI C die Schlüsselworte „public“, „private“ und „protected“ fehlen. Es liegt an der Disziplin des Anwenders entsprechend umsichtig damit umzugehen und das Geheimnisprinzip nicht zu verletzen.
- Zugriffsfunktionen für Variable sind in OV stets nicht-virtuell und können daher nicht überladen werden. (Lösungsmöglichkeit: in der Implementierung der Zugriffsfunktion Indirektion auf eine virtuelle Methode, die im `OPERATIONS`-Block der OV-Modell-Datei definiert wird).
- Virtuelle (d.h. in abgeleiteten Klassen überladbare) Methoden, die auch abstrakt sein können, werden in OV im `OPERATIONS`-Block der OV-Modell-Datei definiert. Die Definition der Argumente erfolgt nicht innerhalb dieser Datei, sondern in der OV-Funktionsprototyp-Datei und wird nicht vom System verwaltet.
- Nicht-virtuelle Methoden werden nicht in der OV-Modell-Datei definiert und werden nicht vom OV-System verwaltet, da der C-Code-Generator hierfür keine Methodentabellen generieren muß.
- In der ANSI-C-Implementierung unter OV bekommen alle Methoden als erstes Argument einen Zeiger auf die Instanzdaten des Objekts. Dies geschieht in C++ implizit.
- Aufgrund der Typprüfungen des Compilers hat der Instanzdatenzeiger als erster Argument unter OV oft einen Typ, der einer Oberklasse entspricht, so daß nicht alle Instanzvariablen sichtbar sind. In diesem Fall muß eine lokale Variable als Instanzdatenzeiger mit dem Typ der aktuellen Klasse definiert werden, welcher der Instanzdatenzeiger per statischem Typ-Cast zugewiesen wird (Ov_StaticCast-Makro).
- Wird umgekehrt in OV eine Methode einer Basisklasse aufgerufen, so ist die Typkonvertierung des Instanzdatenzeigers in einen der Oberklasse entsprechenden Typ erforderlich. Dazu dient das Makro Ov_PtrUpCast.
- Dynamisches Typ-Casten zur Laufzeit ist in OV ähnlich wie in C++ möglich. Dazu dient das Makro Ov_DynamicPtrCast.
- Eingebettete Objekte sind in OV wie in C++ möglich. Hierzu dient in OV der PART-Block, vgl. Kapitel 9, „Parts – in Objekte eingebettete Objekte“.

Ganz wichtig:

- In OV ist ein Klasse nicht nur ein „Konzept“ zur Compile-Zeit, sondern auch ein Objekt zur Laufzeit. Über Klassenobjekte ist zur Laufzeit die volle Metainformation, die in den OV-Modell-Dateien steckt, verfügbar. Unter OV gibt es für jede geladenen Klasse eine globale Variable mit dem Name pclass_*mylib_myclass*, die auf das zugehörige Klassenobjekt verweist.

- Beziehungen zwischen Objekten werden in OV durch Assoziationen modelliert. In C++ existiert dieses Konzept nicht, es muß von „Hand“ über Containerklassen oder Zeiger nachgebildet werden. Übrigens gibt es auch für Assoziationen globale Variable, die auf das Assoziationsobjekt verweisen, sie heißt entsprechend passoc*_mylib_myassoc*).

7.2.2 Ein vergleichendes Beispiel

Wie dies im einzelnen funktioniert, wollen wir anhand eines Beispiels diskutieren. Für diejenigen, die Erfahrung in C++ haben, ist z.T. analoger C++-Code dargestellt. Alle Stellen, die besondere Aufmerksamkeit erfordern, sind fett gedruckt.

OV-Implementierung
Klassendefinitionen in der OV-Modell-Datei mylib.ovm:

```
LIBRARY mylib
  ...
  /* a class derived from the top level class */
  CLASS clsA : CLASS ov/object
    VARIABLES
      myvar : INT HAS_ACCESSORS;
      ...
    END_VARIABLES;
    OPERATIONS
      constructor : C_FUNCTION <OV_FNC_CONSTRUCTOR>;
      foo : C_FUNCTION <FOO_FNC> IS_ABSTRACT;
      ...
    END_OPERATIONS;
  END_CLASS;

  /* a class derived from class clsA */
  CLASS clsB : CLASS mylib/clsA
    IS_INSTANTIABLE; /* it is possible to create instance */
    IS_FINAL;        /* no derived classes possible */
    VARIABLES
      ...
    END_VARIABLES;
    OPERATIONS
      foo : C_FUNCTION <FOO_FNC>;
      ...
    END_OPERATIONS;
  END_CLASS;
  ...
END_LIBRARY;
```

Inhalt der zugehörigen OV-Funktionsprototyp-Datei mylib.ovf:

```
/*
*   File: mylib.ovf
*/
#ifndef MYLIB_OVF_INCLUDED
#define MYLIB_OVF_INCLUDED

typedef OV_DLLFNCEXPORT void FOO_FNC (
  OV_INSTPTR_mylib_clsA pA
);

#endif /* MYLIB_OVF_INCLUDED */
```

Die zugehörigen Implementierungen aus clsA.c:

```
/*
*   File: clsA.c
*/

#include "mylib.h"
#include "libov/ov_macros.h"

/* Variable accessor functions */

OV_DLLFNCEXPORT OV_INT mylib_clsA_myvar_get(
  OV_INSTPTR_mylib_clsA pA
) {
  /* get the value */
  return pA->v_myvar;
}

OV_DLLFNCEXPORT OV_RESULT mylib_clsA_myvar_set(
  OV_INSTPTR_mylib_clsA pA, OV_INT value
) {
  /* set the value */
  pA->v_myvar = value;
  return OV_ERR_OK;
}

/* Operations (virtual methods) */

OV_DLLFNCEXPORT OV_RESULT mylib_clsA_constructor(
  OV_INSTPTR_ov_object pobj
) {
  /* local variables */
  OV_RESULT            result;
  OV_INSTPTR_mylib_clsA pA;
  /* get full instance pointer to see all variables */
  pA = Ov_StaticPtrCast(mylib_clsA, pobj);
  /* do the actual construction */
  result = ov_object_constructor(pobj);
  if(Ov_Fail(result)) {
    return result;
  }
  /* further construction things */
  ...
  return OV_ERR_OK;
}

/* other methods/functions associated with the class (non-virtual) */

OV_DLLFNCEXPORT void mylib_clsA_bar(
  OV_INSTPTR_mylib_clsA pA
) {
  /* local variables */
  OV_INSTPTR_mylib_clsB pB;
  OV_VTBLPTR_mylib_clsA pvtable;
  /* get the pointer to the virtual function table */
  Ov_GetVTablePtr(mylib_clsA, pvtable, pA);
  /* call the virtual foo method */
  pvtable->m_foo(pA);
  ...
  /* test if this object actually is an instance of clsB (or derived) */
  pB = Ov_DynamicPtrCast(mylib_clsB, pA);
  if(pB) {
    /* yes, is an instance of clsB (or derived)! */
```

```
    mylib_clsB_do(pB);
  } else {
    /* no, is not an instance of clsB (or derived) */
    ...
  }
  ...
}
```

Die zugehörigen Implementierungen aus clsB.c:

```
/*
*   File: clsB.c
*/

#include "mylib.h"
#include "libov/ov_macros.h"

/* Variable accessor functions */

...

/* Operations (virtual methods) */

OV_DLLFNCEXPORT void mylib_clsB_foo(OV_INSTPTR_mylib_clsA pA) {
  /* local variables */
  OV_INSTPTR_mylib_clsB pB;
  /* get full instance pointer to see all variables */
  pB = Ov_StaticPtrCast(mylib_clsB, pA);
  ...
}

/* other methods/functions associated with the class (non-virtual) */

OV_DLLFNCEXPORT void mylib_clsB_do(
  OV_INSTPTR_mylib_clsB pB
) {
  /* get value of myvar and print its value */
  printf("myvar = %ld\n", mylib_clsA_myvar_get(
    Ov_PtrUpCast(mylib_clsA, pB)
  );
}
```

Analoge C++-Implementierung

Die Deklaration und Implementation einer analogen C++-Klasse sähe in etwa wie folgt aus:

```
class clsA : public object {
public:
  /* constructor */
  clsA();
  /* variables and accessor functions */
  int myvar;
  int myvar_get();
  result myvar_set(int value);
  ...
  /* virtual methods */
  virtual void foo() = 0;
  ...
  /* non-virtual methods */
  bar();
  ...
};
```

```
clsB : public clsA {
public:
  /* constructor */
  clsB();
  /* variables and accessor functions */
  ...
  /* virtual methods */
  virtual void foo();
  ...
  /* non-virtual methods */
  do();
  ...
};

clsA::clsA() {}

int clsA::myvar_get() {
  return myvar;
}

result clsA::myvar_set(int value) {
  myvar = value;
  return OK;
}

void clsA::bar() {
  /* local variables */
  clsB *pB;
  /* call the virtual foo method */
  foo();
  /* test if this object actually is an instance of clsB (or derived) */
  pB = dynamic_cast<mylib_clsB>(this);
  if(pB) {
    /* yes, is an instance of clsB (or derived)! */
    pB->do();
  } else {
    /* no, is not an instance of clsB (or derived) */
    ...
  }
  ...
}

clsB::clsB() {}

void clsB::foo() {
  ...
}

void clsB::do() {
  printf("myvar = %ld\n", myvar_get());
}
```

7.3 Datenstrukturen und Typkonvertierungen

Die Datenstrukturen für die Instanzdaten und die Methodentabellen, die der C-Code-Generator für das obige Beispiel generiert, sehen wie folgt aus:

```
/*
*   excerpt from file: mylib.h
```

```
*/

/* instance data structures */

typedef struct {
  /* instance data of class ov/object */
  OV_STRING                     v_identifier;
  OV_TIME                       v_creationtime;
  OV_INSTPTR                    v_pouterobject;
  OV_INT                        v_objectstate;
  OV_ATBLPTR                    v_linktable;
  /* instance data added in mylib/clsA */
  OV_INT                        v_myvar;
  ...
} OV_INST_mylib_clsA;
typedef OV_INST_mylib_clsA* OV_INSTPTR_mylib_clsA;

typedef struct {
  /* instance data of class ov/object */
  OV_STRING                     v_identifier;
  OV_TIME                       v_creationtime;
  OV_INSTPTR                    v_pouterobject;
  OV_INT                        v_objectstate;
  OV_ATBLPTR                    v_linktable;
  /* instance data added in mylib/clsA */
  OV_INT                        v_myvar;
  ...
  /* instance data added in mylib/clsA */
  ...
} OV_INST_mylib_clsB;
typedef OV_INST_mylib_clsB* OV_INSTPTR_mylib_clsB;

/* virtual function table structures */

typedef struct {
  /* virtual functions of class ov/object */
  OV_FNC_CONSTRUCTOR  *m_constructor;
  OV_FNC_CHECKINIT    *m_checkinit;
  OV_FNC_DESTRUCTOR   *m_destructor;
  OV_FNC_STARTUP      *m_startup;
  OV_FNC_SHUTDOWN     *m_shutdown;
  OV_FNC_GETACCESS    *m_getaccess;
  OV_FNC_GETFLAGS     *m_getflags;
  OV_FNC_GETCOMMENT   *m_getcomment;
  OV_FNC_GETTECHUNIT  *m_gettechunit;
  OV_FNC_GETVAR       *m_getvar;
  OV_FNC_SETVAR       *m_setvar;
  /* virtual functions added in clsA */
  FOO_FNC             *m_foo;
} OV_VTBL_mylib_clsA;
typedef OV_VTBL_mylib_clsA* OV_VTBLPTR_mylib_clsA;

typedef struct {
  /* virtual functions of class ov/object */
  OV_FNC_CONSTRUCTOR  *m_constructor;
  OV_FNC_CHECKINIT    *m_checkinit;
  OV_FNC_DESTRUCTOR   *m_destructor;
  OV_FNC_STARTUP      *m_startup;
  OV_FNC_SHUTDOWN     *m_shutdown;
  OV_FNC_GETACCESS    *m_getaccess;
  OV_FNC_GETFLAGS     *m_getflags;
  OV_FNC_GETCOMMENT   *m_getcomment;
  OV_FNC_GETTECHUNIT  *m_gettechunit;
```

```
  OV_FNC_GETVAR       *m_getvar;
  OV_FNC_SETVAR       *m_setvar;
  /* virtual functions added in clsA */
  FOO_FNC             *m_foo;
  ...
  /* virtual functions added in clsB */
  ...
} OV_VTBL_mylib_clsB;
typedef OV_VTBL_mylib_clsB* OV_VTBLPTR_mylib_clsB;
```

7.3.1 Up-Casts

Besitzt man eine Zeiger eines bestimmten Typs auf die Instanzdaten eines Objekts einer bestimmten Klasse clsA, so kann der C-Compiler diesen Zeiger nicht in einen Zeiger auf Instanzdaten eines einer anderen Klasse clsA verwandeln, auch wenn dieses aufgrund der Verwandtschaft der beiden Klassen möglich sein müßte (clsB ist spezieller als clsA, d.h. clsB ist eine von clsA abgeleitete Klasse), da diese Tatsache dem C-Compiler nicht bekannt ist – ANSI C kennt eben keine Klassen:

```
OV_INSTPTR_mylib_clsA pA;
OV_INSTPTR_mylib_clsB pB;

pB = ...;
pA = pB;  /* this line results in a compiler error */
```

Aus diesem Grunde muß in ANSI C prinzipiell mit Typ-Casts gearbeitet werden, etwa:

```
OV_INSTPTR_mylib_clsA pA;
OV_INSTPTR_mylib_clsB pB;

pB = ...;
pA = (OV_INSTPTR_mylib_clsA *)pB;  /* no error, but not type safe! */
```

Dies hat jedoch den Nachteil, daß der Compiler keine Typkontrolle durchführen kann, was die Fehleranfälligkeit beträchtlich erhöhen würde. Aus diesem Grund wurden in OV drei Makros implementiert, die – über einen Trick, der auf Zusatzinformationen in den generierten Strukturen beruht – die Typsicherheit wieder herstellen, sofern dies möglich ist.

Ist clsB tatsächlich ist spezieller als clsA, d.h. clsB ist eine von clsA abgeleitete Klasse, so kann man das Makro Ov_PtrUpCast (C-Header-Datei libov/ov_macros.h) einsetzen:

```
OV_INSTPTR_mylib_clsA pA;
OV_INSTPTR_mylib_clsB pB;

pB = ...;
pA = Ov_PtrUpCast(mylib_clsA, pB);  /* no error and type safe */
```

Sollte dies doch nicht stimmen, so meldet der Compiler einen Fehler – wenn auch nicht einen gut verständlichen, da er ja nichts von Vererbung kennt. Die Meldung könnte etwa lauten:

```
structure has no member named `is_of_class_mylib_clsA'
```

Eine derartige Meldung deutet immer auf einen mißlungenen Typ-Cast hin, auch in anderen Makros.

7.3.2 Down-Casts

Das zuvor besprochene Makro führt einen „Up-Cast“ zur Compile-Zeit durch, da von einer Unter- auf eine allgemeinere Oberklasse konvertiert wird. Der umgekehrte Fall, ein „Down-Cast“ von einer allgemeinen Oberklasse auf eine speziellere Unterklasse, kann nicht zur Compile-Zeit durchgeführt werden.

Im allgemeinen muß daher zur Laufzeit überprüft werden, ob eine Typkonvertierung überhaupt statthaft ist. Dies leistet das Makro Ov_DynamicPtrCast (C-Header-Datei libov/ov_macros.h), das einem dynamic_cast<...> in C++ entspricht:

```
OV_INSTPTR_mylib_clsA pA;
OV_INSTPTR_mylib_clsB pB;

pA = ...;
pB = Ov_DynamicPtrCast(mylib_clsB, pA); /* safe, but runtime overhead */

/* note, that pB may be NULL, even if pA is not! */
if(pB) {
  /* Cast was successful, object conforms to class clsB */
  ...
} else {
  /* Cast was not successful, pB must not be used (dereferenced) */
  ...
}
```

Dieses Makro ist typsicher – allerdings darf nicht vergessen werden, den zurückgegebenen Zeiger auf NULL zu testen.

Ein großer Nachteil dieses Makros ist, das es einen gewissen Laufzeit-Overhead mit sich bringt, da die Typkontrolle erst zur Laufzeit stattfindet. Manchmal weiß man aber doch schon zur Compile-Zeit, daß man tatsächlich einen Zeiger auf die Instanzdaten eines spezielleren Objekts besitzt, als im Typ des Zeigers zum Ausdruck kommt. In diesem Fall kann man den Compiler dazu zwingen, die Konvertierung durchzuführen – unter Verlust der Typkontrolle. Dies tritt typischerweise auf, wenn eine Methode die Methode einer Oberklasse überlädt und der Instanzdatenzeiger als erstes Argument nur auf die Instanzdaten dieser Oberklasse zeigt. Ein Beispiel hierfür ist:

```
OV_DLLFNCEXPORT void mylib_clsB_foo(OV_INSTPTR_mylib_clsA pA) {
  /* get full instance pointer to see all variables; BE CAREFUL! */
  OV_INSTPTR_mylib_clsB pB = Ov_StaticPtrCast(mylib_clsB, pA);
  ...
}
```

7.3.3 Zugriff auf Methodentabellen

Für den Zugriff auf die Methodentabelle eines Objekts muß dessen Instanzdatenzeiger bekannt sein. Der Typ des Methodentabellenzeigers muß zum Typ des Instanzdatenzeigers passen; dies wird ähnlich wie bei den Typkonvertierungen durch einen Trick zur Compile-Zeit überprüft. Mit Hilfe des Methodentabellenzeigers kann dann die virtuelle Methode aufgerufen werden.

Beispiel:

```
OV_DLLFNCEXPORT void mylib_clsA_bar(
  OV_INSTPTR_mylib_clsA pA
```

```
) {
  OV_VTBLPTR_mylib_clsA pvtable;
  Ov_GetVTablePtr(mylib_clsA, pvtable, pA);
  pvtable->m_foo(pA);
  ...
}
```

Übrigens sind grundsätzlich alle Methodenzeiger in einer Methodentabelle ungleich NULL, da sonst gar keine Instanz hätte gebildet werden können, denn Klassen mit abstrakten Methoden (Operationen mit dem Schlüsselwort `IS_ABSTRACT`) sind nicht instanziierbar.

7.4 Die Top-Level-Klasse „ov/object“

Da jede Klasse indirekt oder direkt von der Klasse „ov/object“ erbt, verfügt jedes Objekt über die in dieser Klasse definierten (virtuellen) Methoden und Instanzdaten. Die Methoden besitzen eine hohe Generizität und können in eigenen Klassen redefiniert (überladen) werden, um das Verhalten des Objekts entsprechend zu variieren. Es gibt zwei Kategorien von solchen Methoden:

- die Life-Cycle-Methoden,
- Zugriffsmethoden.

7.4.1 Instanzdaten

Die Instanzdaten der Klasse „ov/object“ lauten:

```
typedef struct {
  /* variables */
  OV_STRING                    v_identifier;   /* id (name) */
  OV_TIME                      v_creationtime; /* time of creation */
  OV_INSTPTR                   v_pouterobject; /* embedding obj if any */
  OV_INT                       v_objectstate;  /* object state */
  /* linktable */
  OV_ATBLPTR                   v_linktable;
} OV_INST_ov_object;
typedef OV_INST_ov_object* OV_INSTPTR_ov_object;
```

Übrigens eignet sich die Klasse „ov/domain“ ebenso als allgemeine Basisklasse; sie ergänzt die Klasse „ov/object“ um die Möglichkeit, Kind-Objekte (Assoziation „ov/containment“) zu enthalten, also im ACPLT Magellan als Domain aufzutreten, die neben Variablen und Links auch Objekte enthält.

7.4.2 Life-Cycle-Methoden

Die Life-Cycle-Methoden

- constructor,
- checkinit,
- startup,
- shutdown und
- destructor.

begleiten den Lebenszyklus eines Objekts.

Die „constructor“-Methode
Unmittelbar nach der Bereitstellung der Instanzdaten des Objekts durch das System und einige wenige Vorinitialisierungen (Link zur übergeordneten Domain und zur Klasse, Setzen des

Objekt-Namens (identifier) und Ausnullen aller Variablen und Links) wird zunächst der Konstruktor „constructor“ aufgerufen. In dieser Methode können weitere Initialisierungen durchgeführt werden, z.B. Variablen oder Links vorbesetzt, weitere Ressourcen beschafft oder weitere Objekte angelegt werden. Sollte es Ressourcen-Probleme geben, so kann ein Fehlercode ungleich `OV_ERR_OK` zurückgegeben werden, so daß das System das Objekt komplett wieder löscht, die Objekterzeugung also fehlschlägt. Vorher müssen alle neu belegten Ressourcen freigegeben werden. Ist alles in Ordnung, gibt man den Fehlercode `OV_ERR_OK` zurück.

Beispiel:

```
OV_DLLFNCEXPORT OV_RESULT mylib_myclass_constructor(
  OV_INSTPTR_ov_object pobj
) {
  OV_INSTPTR_mylib_myclass pthis = Ov_StaticPtrCast(mylib_myclass, pobj);
  pthis->v_temperatur = 20; /* 20 Celsius is the default temperature */
  return OV_ERR_OK;
}
```

Die „checkinit“-Methode

Unmittelbar nach einer erfolgreich beendeten „constructor“-Methode, wird eine von der erzeugenden Routine bereitgestellte Initialisierungsfunktion aufgerufen, die es gestattet, benutzerspezifische Initialisierungen durchzuführen, vgl. Abschnitt 7.5, „Erzeugen und Löschen von Objekten“. (Dies geschieht z.B. bei Verwendung von Initialisierungs-Parameter- und -Link-Listen mittels des CreateObject-Dienstes via ACPLT/KS.) Unmittelbar danach wird die Methode „checkinit“ aufgerufen, die zur Überprüfung dient, ob diese Initialisierung überhaupt möglich (erlaubt) ist. Ist dies der Fall, gibt man den Fehlercode `OV_ERR_OK` zurück, andernfalls den Fehlercode von `OV_ERR_BADINITPARAM`. Im Falle einer korrekten Initialisierung ist nun die Objekterzeugung vollständig beendet, andernfalls schlägt sie fehl und das Objekt wird gelöscht.

Beispiel:

```
OV_DLLFNCEXPORT OV_RESULT mylib_myclass_checkinit(
  OV_INSTPTR_ov_object pobj
) {
  OV_INSTPTR_mylib_myclass pthis = Ov_StaticPtrCast(mylib_myclass, pobj);
  if (pthis->v_temperatur < TEMPERATURE_MIN) {
    return OV_ERR_BADINITPARAM;
  }
  return OV_ERR_OK;
}
```

Die „startup“-Methode

Die „startup“-Methode eines Objekts wird zum ersten Mal im Lebenszyklus des Objekts unmittelbar nach der vollständigen Objekterzeugung (also nach der „checkinit“-Methode) aufgerufen. Diese Methode kann aber auch später noch beliebig häufig aufgerufen werden, wenn das OV-System heruntergefahren (oder gewaltsam beendet) und später wieder neu gestartet wurde – OV-Objekte überleben dies aufgrund ihrer Persistenz. In dieser Methode können alle Operationen durchgeführt werden, die notwendig sind, damit das Objekt seinen Betrieb aufnehmen kann (z.B. Registrieren beim OV-Scheduler, vgl. Kapitel 11, „Aktive Objekte“). Es ist nicht mehr möglich, einen Fehler zu melden um das Objekt löschen zu lassen (es kann sich prinzipiell aber selber löschen). Im Objekt gespeicherte, aber nicht von OV verwaltete Zeiger

(z.B. auf eine Datei-Deskriptor oder dynamisch auf dem Heap reservierten Speicher) sind beim Aufruf dieser Methode grundsätzlich nicht mehr gültig.

Beispiel:

```
void mylib_myclass_execute(OV_INSTPTR_ov_object pobj) {
  ...
}

OV_DLLFNCEXPORT void mylib_myclass_startup(OV_INSTPTR_ov_object pobj) {
  ov_object_startup(pobj);
  ov_scheduler_register(pobj, mylib_myclass_execute);
}
```

Die „shutdown“-Methode
Die „shutdown“-Methode eines Objekts wird zum letzten Mal im Lebenszyklus des Objekts unmittelbar vor dem tatsächlichen Löschen (also vor der „destructor“-Methode) aufgerufen. Diese Methode wird aber auch dann aufgerufen, wenn das OV-System heruntergefahren wird. In diesem Fall überlebt das Objekt aufgrund seiner Persistenz bis zum nächsten Neustart (bei dem es wieder zum Aufruf der „startup“-Methode kommt) und der Destruktor wird nicht aufgerufen. In dieser Methode können alle Operationen durchgeführt werden, die notwendig sind, damit das Objekt seinen Betrieb ordnungsgemäß beendet (z.B. Deregistrieren beim OV-Scheduler, vgl. , vgl. Kapitel 11, „Aktive Objekte“). Dazu gehört unter Umständen auch die Freigabe von Ressourcen wie Dateien oder Heap-Speicher, da diese nach dem Stoppen des OV-Systems nicht überleben können.

Beispiel:

```
OV_DLLFNCEXPORT void mylib_myclass_shutdown(OV_INSTPTR_ov_object pobj) {
  ov_scheduler_unregister(pobj);
  ov_object_shutdown(pobj);
}
```

Die „destructor“-Methode
Die „destructor“-Methode eines Objekts unmittelbar vor dem Ende des Lebenszyklus eines Objekts aufgerufen. Hier müssen alle noch verwendeten Ressourcen freigegeben werden, die nicht vom OV verwaltet werden. Gleichzeitig können weitere Aktionen durchgeführt werden, beispielsweise das Löschen von weiteren Objekten. Das OV-System sorgt von sich aus dafür, daß alle Links auf das Objekt, die nach dem Beenden des Destruktors noch existieren, entfernt werden, so daß referentielle Integrität sichergestellt ist. Auch Variablenwerte mit dynamischer Größe (Strings, dynamische Vektoren) werden vollständig aus der Datenbasis gelöscht.

7.4.3 Zugriffsmethoden

Die Zugriffsmethoden
- getaccess,
- getflags,
- getcomment,
- gettechnunit,
- getvar und
- setvar

kontrollieren den Zugriff auf Informationen im Zusammenhang mit dem Objekt und seinen Variablen und Links. Neben dem Instanzdatenzeiger besitzen alle diese Methoden ein Argu-

ment, das beschreibt, auf welchen Teil des Objekts verwiesen wird. Einiges wird sicherlich in den folgenden Beispielen deutlich, ansonsten sei auf das Kapitel 0, „

Introspektion und Reflektion“ verwiesen.

Die „getaccess“-Methode
Diese Methode legt die Zugriffsrechte des Objekts selbst und seiner Bestandteile fest. Dazu bekommt es ein Ticket (vgl. ACPLT/KS-Tickets) als Eingangsinformation, beispielsweise eine Kombination aus Benutzernamen und Paßwort oder eine NULL-Zeiger (kein Ticket).

Beispiel:

```
OV_DLLFNCEXPORT OV_ACCESS mylib_myclass_getaccess(
  OV_INSTPTR_ov_object pobj,
  const OV_ELEMENT     *pelem,
  const OV_TICKET      *pticket
) {
  /* local variables */
  OV_ACCESS access;
  /* get access rights defined by default */
  access = ov_object_getaccess(pobj, pelem, pticket);
  /* modify access rights */
  switch(pelem->elemtype) {
  case OV_ET_OBJECT:
    /* access rights for the object itself */
    access &= ~OV_AC_RENAMEABLE;       /* never allowed to rename */
    if(...) access |= OV_AC_DELETABLE; /* deleteable if condition true */
    break;
  case OV_ET_VARIABLE:
    /* access rights for variables of the object */
    switch(pelem->elemunion.pvar->v_offset) {
    case offsetof(OV_INST_mylib_myclass, v_myvar):
      /* not visible (no write and read access) if condition true */
      if(...) access &= ~ OV_AC_READWRITE;
      break;
    default:
      break;
    }
    break;
  case OV_ET_PARENTLINK:
    ...
    break;
  case OV_ET_CHILDLINK:
    ...
    break;
  default:
    break;
  }
  return access;
}
```

Die „getflags“-Methode
Diese Methode liefert die mit einem Objekt oder einem seiner Bestandteile assoziierten semantischen Flags zurück. Normalerweise sind dies die in der OV-Modell-Datei in der Klasse des Objekts, in der Variablendefinition oder mit der Assoziation festgelegten Flags, sie können aber über diese Methode dynamisch verändert werden.

Beispielhaft sei hier die Default-Implementation in der Klasse „ov/object“ dargestellt:

```
OV_DLLFNCEXPORT OV_UINT ov_object_getflags(
  OV_INSTPTR_ov_object pobj, const OV_ELEMENT *pelem
) {
```

```
  switch(pelem->elemtype) {
  case OV_ET_OBJECT:
    /* flags of the object itself */
    return Ov_GetParent(ov_instantiation, pobj)->v_flags;
  case OV_ET_VARIABLE:
    /* flags associated with a variable */
    return pelem->elemunion.pvar->v_flags;
  case OV_ET_PARENTLINK:
    /* flags associated with a parent link */
    return pelem->elemunion.passoc->v_parentflags;
  case OV_ET_CHILDLINK:
    /* flags associated with a child link */
    return pelem->elemunion.passoc->v_childflags;
  default:
    break;
  }
  return 0;
}
```

Die „getcomment“-Methode

Diese Methode liefert den mit einem Objekt oder einem seiner Bestandteile assoziierten Kommentar zurück. Normalerweise ist dieser in der OV-Modell-Datei in der Klasse des Objekts, in der Variablendefinition oder mit der Assoziation festgelegt, kann aber über diese Methode dynamisch verändert werden.

Der Aufrufer dieser Methode muß vor dem Aufruf die Funktion ov_memstack_lock() aufrufen und später, nachdem er nicht mehr auf den zurückgelieferten String zugreift entsprechend ov_memstack_unlock() aufrufen. Dementsprechend darf innerhalb dieser Routine Speicher mit ov_memstack_alloc() angefordert werden, beispielsweise um Speicher zur Aufnahme des Kommentars bereitzustellen. Dieser Speicher muß (und kann) nicht explizit freigegeben werden (dies geschieht implizit durch ov_memstack_unlock()).

Beispielhaft sei hier die Default-Implementation in der Klasse „ov/object“ dargestellt:

```
OV_DLLFNCEXPORT OV_STRING ov_object_getcomment(
  OV_INSTPTR_ov_object pobj, const OV_ELEMENT *pelem
) {
  switch(pelem->elemtype) {
  case OV_ET_OBJECT:
    /* flags of the object itself */
    return Ov_GetParent(ov_instantiation, pobj)->v_comment;
  case OV_ET_VARIABLE:
    /* flags associated with a variable */
    return pelem->elemunion.pvar->v_comment;
  case OV_ET_PARENTLINK:
    /* flags associated with a parent link */
    return pelem->elemunion.passoc->v_parentcomment;
  case OV_ET_CHILDLINK:
    /* flags associated with a child link */
    return pelem->elemunion.passoc->v_childcomment;
  default:
    break;
  }
  return NULL;
}
```

Die „getunit“-Methode

Diese Methode liefert die mit der Variable eines Objekts assoziierte technische Einheit zurück. Normalerweise ist diese in der Variablendefinition in der OV-Modell-Datei festgelegt, kann aber über diese Methode dynamisch verändert werden.

Der Aufrufer dieser Methode muß vor dem Aufruf die Funktion ov_memstack_lock() aufrufen und später, nachdem er nicht mehr auf den zurückgelieferten String zugreift entsprechend ov_memstack_unlock() aufrufen. Dementsprechend darf innerhalb dieser Routine Speicher mit ov_memstack_alloc() angefordert werden, beispielsweise um Speicher zur Aufnahme der technischen Einheit bereitzustellen. Dieser Speicher muß (und kann) nicht explizit freigegeben werden (dies geschieht implizit durch ov_memstack_unlock()).

Beispiel:

```
OV_DLLFNCEXPORT OV_STRING mylib_myclass_gettechunit(
  OV_INSTPTR_ov_object pobj, const OV_ELEMENT *pelem
) {
  /* local variables */
  OV_INSTPTR_mylib_myclass pthis = Ov_StaticPtrCast(mylib_myclass, pobj);
  /* get technical unit */
  if(pelem->elemtype == OV_ET_VARIABLE) {
    switch(pelem->elemunion.pvar->v_offset) {
    case offsetof(OV_INST_mylib_myclass, v_myvar):
      /* return a technical unit stored in a variable of the object */
      return pthis->v_TechUnit; /* variable type OV_STRING */
    default:
      break;
    }
  }
  /* fall back on the default implementation */
  return ov_object_gettechunit(pobj, pelem);
}
```

Die „getvar“-Methode

Diese Methode liefert den mit der Variable eines Objekts assoziierten technische Variablenwert inklusive Status und Zeitstempel zurück. Normalerweise erhält man – mit Ausnahme der PV-Variablentypen `BOOL_PV`, `INT_PV` und `SINGLE_PV` – nur den eigentlichen Wert, der Status ist OV_ST_NOTSUPPORTED und der Zeitstempel entspricht dem aktuellen Stand der Systemuhr. Dies kann aber über diese Methode dynamisch verändert werden.

Der Aufrufer dieser Methode muß vor dem Aufruf die Funktion ov_memstack_lock() aufrufen und später, nachdem er nicht mehr auf den zurückgelieferten String zugreift entsprechend ov_memstack_unlock() aufrufen. Dementsprechend darf innerhalb dieser Routine Speicher mit ov_memstack_alloc() angefordert werden, beispielsweise um Speicher zur Aufnahme des Wertes bereitzustellen. Dieser Speicher muß (und kann) nicht explizit freigegeben werden (dies geschieht implizit durch ov_memstack_unlock()).

Beispiel:

```
OV_DLLFNCEXPORT OV_RESULT mylib_myclass_getvar(
  OV_INSTPTR_ov_object pobj,
  const OV_ELEMENT     *pelem,
  OV_ANY               *pvarcurrprops
) {
  /* local variables */
  OV_INSTPTR_mylib_myclass pthis = Ov_StaticPtrCast(mylib_myclass, pobj);
  OV_RESULT result;
  /* use the default implementation and modify the timestamp */
```

```
  result = ov_object_getvar(pobj, pelem, pvarcurrprops);
  if(Ov_OK(result)) {
    switch(pelem->elemtype) {
    case OV_ET_VARIABLE:
      /* replace the time stamp with a time stored in the object */
      pvarcurrprops->time = pthis->v_BlockTime; /* variable type OV_TIME */
      return OV_ERR_OK;
    default:
      break;
    }
    return OV_ERR_BADPARAM;
  }
  return result;
}
```

Die „setvar“-Methode
Mit dieser Methode wird der Wert einer mit der Variable eines Objekts assoziierten Variable gesetzt, wobei nicht nur der eigentliche Wert, sondern auch Status und Zeitstempel gegeben sind. Normalerweise werden – mit Ausnahme der PV-Variablentypen `BOOL_PV`, `INT_PV` und `SINGLE_PV` – Status und Zeitstempel ignoriert. Dies kann aber über diese Methode auch diese Informationen nutzen, beispielsweise im Objekt speichern.

Beispiel:

```
OV_DLLFNCEXPORT OV_RESULT mylib_myclass_setvar(
  OV_INSTPTR_ov_object pobj,
  const OV_ELEMENT     *pelem,
  const OV_ANY         *pvarcurrprops
) {
  /* local variables */
  OV_INSTPTR_mylib_myclass pthis = Ov_StaticPtrCast(mylib_myclass, pobj);
  OV_RESULT result;
  /* use the default implementation and modify the timestamp */
  result = ov_object_setvar(pobj, pelem, pvarcurrprops);
  if(Ov_OK(result)) {
    switch(pelem->elemtype) {
    case OV_ET_VARIABLE:
      /* store status and time stamp in the object */
      pthis->v_Status = pvarcurrprops->state; /* variable type OV_UINT */
      pthis->v_Time = pvarcurrprops->time;    /* variable type OV_TIME */
      return OV_ERR_OK;
    default:
      break;
    }
    return OV_ERR_BADPARAM;
  }
  return result;
}
```

7.5 Erzeugen und Löschen von Objekten

Objekte werden mit Hilfe der Funktion ov_class_createobject() unter Angabe einer Initialisierungsroutine erzeugt.

Beispiel:

```
#include "libov/ov_class.h"
```

```
...

typedef struct {
  OV_INT    a;
  OV_SINGLE b;
} INITDATA;

OV_RESULT initfnc(OV_INSTPTR_ov_object pobj, OV_POINTER userdata) {
  /* local variables */
  OV_INSTPTR_mylib_myclass pthis = Ov_StaticPtrCast(mylib_myclass, pobj);
  INITDATA                 *pinit = (INITDATA *)userdata;
  /* initialized the object */
  pthis->v_myintvar = pinit->a;
  pthis->v_mysinglevar =  pinit->b;
  return OV_ERR_OK;
}

void foo(void) {
  /* local variables */
  OV_INSTPTR_mylib_myclass pinst;
  OV_RESULT                result;
  INITDATA                 init;
  /* create the object */
  init.a = 4711;
  init.b = 3.141;
  result = ov_class_createobject(pclass_mylib_myclass, &pdb->root,
    "myinstance", OV_PMH_DEFAULT, NULL, initfnc, &init,
    (OV_INSTPTR *)&pinst);
  if(Ov_OK(result)) {
    /* object was created */
    ...
  } else {
    /* object creation failed */
    ...
  }
}
```

In diesem Beispiel ist &pdb->root übrigens der Zeiger auf das Root-Domain-Objekt „/“. Wird keine Initialisierungsmethode benötigt, kann auch ein Makro verwendet werden. Beispiel:

```
#include "libov/ov_macros.h"

...

OV_INSTPTR_mylib_myclass pinst;
OV_RESULT                result;

result = Ov_CreateObject(mylib_myclass, &pdb->root, pinst, "myinstance");
if(Ov_OK(result)) {
  /* object was created */
  ...
} else {
  /* object creation failed */
  ...
}
```

Das Löschen von Objekten ist relativ einfach:

```
#include "libov/ov_macros.h"

...
```

```
OV_RESULT result;

...

result = Ov_DeleteObject(pobj);
if(Ov_Fail(result)) {
  /* deleting the object failed */
  ...
}
```

Wie das Erzeugen und Löschen von Objekten dem ACPLT Magellan geschieht, ist in Abschnitt 2.4, „Der ACPLT Magellan“ beschrieben.

8 Instanzvariablen

8.1 Definition einer Variablen in der OV-Modell-Datei

Die Definition von Variablen innerhalb einer Klasse sieht wie folgt aus:

```
LIBRARY mylib ...
  CLASS myclass ...
    VARIABLES;
      bool : BOOL;
      int : INT COMMENT = "signed integer variable (32 bit)";
      uint : UINT HAS_GET_ACCESSOR;
      single : SINGLE UNIT = "m/s" INITIALVALUE = 2.5;
      double : DOUBLE FLAGS = "abckl";
      time : TIME HAS_SET_ACCESSOR;
      timeSpan : TIME_SPAN IS_DERIVED;
      string : HAS_ACCESSORS;
      userDefined : C_TYPE <MY_OWN_TYPE>;
      fixedBoolVec[4] : BOOL ...;
      dynamicIntVec[] : INT ...;
      boolPV : BOOL_PV ...;
      intPV : INT_PV ...;
      singlePV : SINGLE_PV ...;
    END_VARIABLES;
    ...
  END_CLASS;
  ...
END_LIBRARY;
```

Folgende Datentypen stehen für Variablen zur Verfügung:

Tabelle 2: Variablentypen in ACPLT/OV

Datentyp	Wertebereich	Bedeutung
`BOOL`	{ FALSE, TRUE }	Boolsche Variable
`INT`	-2147483648 bis 2147483647	32-bittige Ganzzahl mit Vorzeichen
`UINT`	0 bis 4294967295	32-bittige Ganzzahl ohne Vorzeichen
`SINGLE`	-3.402xxE38 bis 3.402xxE38	Fließkommazahl mit einfacher Genauigkeit (4 Byte)
`DOUBLE`	-1.797xxE308 bis 1.797xxE308	Fließkommazahl mit doppelter Genauigkeit (8 Byte)
`TIME`	01.01.1970 bis ca. 01.01.2106	Datum und Zeit (gemessen in Sekunden / Mikrosekunden seit dem 01.01.1970, 0 Uhr)
`TIME_SPAN`	-2147483648.999999s bis 2147483647.999999s	Zeitdauer, gemessen in Sekunden / Mikrosekunden
`STRING`	0-terminierter ASCII-String	Zeichenkette
`C_TYPE <...>`	undefiniert	benutzer-definierter ANSI-C-Datentyp
`BOOL_PV`	wie BOOL, zusätzlich mit Status und Zeitstempel	Boolscher Prozeßwert
`INT_PV`	wie INT, zusätzlich mit Status und Zeitstempel	Ganzzahliger Prozeßwert

UINT_PV	wie UINT, zusätzlich mit Status und Zeitstempel	Ganzzahliger Prozeßwert mit Vorzeichen
SINGLE_PV	wie SINGLE, zusätzlich mit Status und Zeitstempel	Analoger Prozeßwert (einfache Genauigkeit)
DOUBLE_PV	wie DOUBLE, zusätzlich mit Status und Zeitstempel	Analoger Prozeßwert (doppelte Genauigkeit)
TIME_PV	wie TIME, zusätzlich mit Status und Zeitstempel	Datum und Uhrzeit aus dem Prozess
TIME_SPAN_PV	wie TIME_SPAN, zusätzlich mit Status und Zeitstempel	Prozeßzeitspanne
STRING_PV	wie STRING, zusätzlich mit Status und Zeitstempel	Text-Prozeßwert

Bis auf C_TYPE <> und xxx_PV können diese Datentypen sowohl in Skalaren, als auch in Vektoren mit fester Länge (fixedBoolVec[4]) oder veränderbarer Länge (dynamicIntVec[]) verwendet werden.

Für den Fall, daß ein C_TYPE <>-Datentyp verwendet wird, muß dieser innerhalb der entsprechenden OV-Datentyp-Datei deklariert werden.

Variablen können eine technische Einheit (UNIT), einen Kommentar (COMMENT) sowie semantische Flags (FLAGS) besitzen. Darüber hinaus können Zugriffsfunktionen zum Lesen (HAS_GET_ACCESSOR) und zum Schreiben (HAS_SET_ACCESSOR) des Variablenwertes vereinbart werden. Normalerweise besitzt eine Variable eine Repräsentation innerhalb der Instanzdaten; mit Hilfe des Schlüsselwortes IS_DERIVED kann die aber unterdrückt werden. In diesem Fall muß die Variable über Zugriffsfunktionen zum Lesen und/oder Schreiben besitzen, die beispielsweise den Variablenwert rechnerisch bestimmen oder anderweitig verarbeiten.

8.2 Repräsentation der Variablen in ANSI C

8.2.1 Skalare

Die Variablentypen werden bei Skalaren wie folgt in ANSI C repräsentiert:

Tabelle 3: Repräsentation skalarer Variablen

Datentyp	Repräsentation
BOOL	`typedef int OV_BOOL;`
INT	`typedef long OV_INT;`
UINT	`typedef unsigned long OV_UINT;`
SINGLE	`typedef float OV_SINGLE;`
DOUBLE	`typedef double OV_DOUBLE;`

TIME	`typedef struct {`
	`  OV_UINT  secs;`
	`  OV_UINT  usecs;`
	`} OV_TIME;`

TIME_SPAN	`typedef struct {`
	`  OV_INT  secs;`
	`  OV_INT  usecs;`
	`} OV_TIME_SPAN;`

STRING	`typedef char* OV_STRING;`
BOOL_PV	`typedef struct {`

	OV_BOOL value; OV_STATE state; OV_TIME time; } OV_BOOL_PV;
INT_PV	typedef struct { OV_INT value; OV_STATE state; OV_TIME time; } OV_INT_PV;
UINT_PV	typedef struct { OV_UINT value; OV_STATE state; OV_TIME time; } OV_SINGLE_PV;
SINGLE_PV	typedef struct { OV_SINGLE value; OV_STATE state; OV_TIME time; } OV_SINGLE_PV;
DOUBLE_PV	typedef struct { OV_DOUBLE value; OV_STATE state; OV_TIME time; } OV_SINGLE_PV;
TIME_PV	typedef struct { OV_TIME value; OV_STATE state; OV_TIME time; } OV_SINGLE_PV;
TIME_SPAN_PV	typedef struct { OV_TIME_SPAN value; OV_STATE state; OV_TIME time; } OV_SINGLE_PV;
STRING_PV	typedef struct { OV_STRING value; OV_STATE state; OV_TIME time; } OV_SINGLE_PV;

Für die Stati einer Variable sind z.Z. folgende Werte definiert:

```
#define OV_ST_NOTSUPPORTED 0x00000000 /* no state available */
#define OV_ST_UNKNOWN      0x00000001 /* state unknown at this time */
#define OV_ST_BAD          0x00000002 /* information is bad */
#define OV_ST_QUESTIONABLE 0x00000003 /* information is questionable */
#define OV_ST_GOOD         0x00000004 /* information is good */
```

Mit Ausnahme des `STRING`-Datentyps werden die Skalare vollständige im Objekt gespeichert; im Falle des Strings liegt im Objekt nur ein Zeiger auf den 0-terminierten String, ein char-Feld, z.B.:

```
typedef struct {
  ...
  OV_INT    v_int;
  OV_STRING v_string;
  ...
} OV_INST_mylib_myclass;
```

8.2.2 Vektoren fester Länge

Vektor-Variablen mit fester Länge werden entsprechend als eindimensionales Array im Objekt gespeichert, z.B.:

```
typedef struct {
  ...
  OV_INT    v_fixedBoolVec[4];
  OV_STRING v_fixedStringVec[13];
  ...
} OV_INST_mylib_myclass;
```

Auch hier sind Strings eine Ausnahme, im Objekt liegt nur das Zeigerfeld.

8.2.3 Dynamische Vektoren

Bei Vektoren variable Länge liegt im Objekt nur ein Datenfeld mit der augenblicklichen Vektorlänge und ein Zeiger auf ein außerhalb des Objekts liegendes eindimensionales Array, z.B.:

```
typedef struct {
  OV_UINT veclen;
  OV_INT  *value;
} OV_INT_VEC;

typedef struct {
  OV_UINT   veclen;
  OV_STRING *value;
} OV_STRING_VEC;

typedef struct {
  ...
  OV_INT_VEC     v_dynamicIntVec;
  OV_STRING_VEC  v_dynamicStringVec;
  ...
} OV_INST_mylib_myclass;
```

Für die anderen Datentypen gilt dies in analoger Weise.

8.3 Variablenzugriffe in der C-Implementation

Hier wird gezeigt, wie auf die Variablen eines Objekts in der ANSI C-Implementation seiner Klasse zugegriffen wird. Der Zugriff auf die Instanzvariablen eines anderen Objekts sollte prinzipiell immer über die Zugriffsfunktionen erfolgen, damit das Geheimnisprinzip nicht verletzt wird, vgl. Abschnitt 8.5, „Zugriff auf Variablen anderer Objekte“.

8.3.1 Zugriff auf Skalare und Vektoren fester Länge (keine Strings)

Der lesende und schreibende Zugriff auf Instanzvariablen eines Objekts aus einer Methode des Objekts heraus erfolgt normalerweise direkt über den Instanzdatenzeiger pthis:

```
#include "libov/ov_time.h"

...

OV_SINGLE x, y;
OV_UINT i;
```

```
...

x = pthis->v_single;
pthis->v_single = y;

...

pthis->v_singlePV.value = y;
pthis->v_singlePV.state = OV_ST_GOOD;
ov_time_gettime(&pthis->v_singlePV.time);

...

for(i=0; i<4; i++) {
  printf("fixedBoolVec[%ld] = %s\n", i,
    pthis->v_fixedBoolVec[i]?"TRUE":"FALSE");
  pthis->v_fixedBoolVec[i] = TRUE;
}
```

Zur Manipulation von Zeitpunkten (`TIME`) und Zeitspannen (`TIME_SPAN`) sind die Funktionen aus der C-Header-Datei libov/ov_time.h vorhanden (aktuelle Zeit, Summe aus Zeitpunkt und Zeitspanne, Differenz zwischen Zeitpunkten); siehe dazu die LibOV API-Referenz.

Der Zugriff auf komplexe Variablen (Strings, dynamische Vektoren) ist komplizierter. Der lesende Zugriff ist noch recht einfach, aber beim Schreiben muß u.U. dynamischer Speicher verwaltet werden.

8.3.2 Zugriff auf Strings

Der lesende Zugriff auf Strings ist unkritisch, beispielsweise:

```
printf("myString = %s\n", pthis->v_myString);
```

Das Ändern eines Strings geschieht über die Funktionen aus der C-Header-Datei libov/ov_string.h (siehe LibOV API-Referenz), z.B.:

```
#include "libov/ov_string.h"

...

if(Ov_Fail(ov_string_setvalue(&pthis->v_myString, "a new text"))) {
  /* could not set value, not enough memory */
  ...
}
```

8.3.3 Zugriff auf dynamische Vektoren

Der lesende Zugriff auf dynamische Vektoren ist ebenfalls unkritisch, beispielsweise:

```
OV_UINT i;

...

for(i=0; i<pthis->v_dynamicIntVec.veclen; i++) {
  printf("dynamicIntVec[%ld] = %s\n", i, pthis->v_dynamicIntVec.value[i]);
}
```

Das Ändern schon vorhandener Feldinhalte ist genauso unkritisch, etwa:

```
OV_UINT i;

...

for(i=0; i<pthis->v_dynamicIntVec.veclen; i++) {
  pthis->v_dynamicIntVec.value[i] = i*4;
}
```

Soll die Größe eines dynamischen Vektors geändert werden, so dient hierzu das Makro Ov_SetDynamicVectorLength aus der C-Header-Datei libov/macros.h:

```
#include "libov/ov_macros.h"

...

OV_UINT veclen;
OV_RESULT result;

...

veclen = ...;
result = Ov_SetDynamicVectorLength(&pthis->v_dynamicIntVec, veclen, INT);
if(Ov_OK(result)) {
  /* success, maybe change some entries */
  ...
} {
  /* failure, not enough memory */
  ...
}
```

Zu beachten ist, daß diese Operation fehlschlagen kann, weil nicht genügend Speicher vorhanden ist. Werden neue Felder erzeugt, weil der Vektor sich vergrößert hat, so werden diese mit 0 initialisiert.

Es ist auch möglich, einen dynamischen Vektor „ein einem Rutsch“ zu ändern. Dazu muß der gewünschte Wert schon im passenden Format vorliegen.

Beispiel:

```
#include "libov/ov_macros.h"

...

result = Ov_SetDynamicVectorValue(&pthis->v_dynamicIntVec,
  &pthis->v_anotherDynamicIntVec, pthis->v_anotherDynamicIntVec.veclen,
  INT);
if(Ov_OK(result)) {
  /* success, maybe change some entries */
  ...
} {
  /* failure, not enough memory */
  ...
}
```

8.4 Programmieren der Zugriffsfunktionen

Zugriffsfunktionen, die in der OV-Modell-Datei mit Hilfe der Schlüsselworte HAS_GET_ACCESSOR, HAS_SET_ACCESSOR und HAS_ACCESSORS deklariert werden, dienen der Abstraktion von Variablenzugriffen entsprechend dem Geheimnisprinzip. In diesen Zugriffsfunktionen können Berechnungen durchgeführt oder Bereichsgrenzen überprüft werden.

Get-Zugriffsfunktionen müssen immer einen Wert zurückliefern, set-Zugriffsfunktionen jedoch können den zu setzenden Wert überprüfen und ggf. OV_ERR_BADVALUE zurückliefern, ohne den gespeicherten Wert zu verändern.

Da Zugriffsfunktionen u.U. komplexe Werte als Ausgabe bereitstellen bzw. als Eingabe bekommen, ist grundsätzlich darauf zu achten, wer den notwendigen Speicher zur Verfügung stellt. Die getroffenen Konventionen werden im folgenden dargestellt.

8.4.1 Zugriffsfunktionen für nicht-komplexe Variablen

Unter den nicht-komplexen Variablen werden alle Skalare mit Ausnahme von STRING-, C_TYPE <>- und PV-Variablen (BOOL_PV, INT_PV, SINGLE_PV) verstanden. Variablenwerte dieser Typen werden direkt übergegeben, so daß der Compiler ein echte Kopie erzeugt.

Beispiele:

```
#include "libov/ov_time.h"

/* get accessor functions */

OV_INT mylib_myclass_myInt_get(OV_INSTPTR_mylib_myclass pthis) {
  /* a variable not declared as IS_DERIVED is stored in the object */
  return pthis->v_myInt;
}

OV_TIME mylib_myclass_myTime_get(OV_INSTPTR_mylib_myclass pthis) {
  /* a variable declared as IS_DERIVED must be calculated */
  OV_TIME time;
  ov_time_gettime(&time);
  return time;
}

/* set accessor functions */

OV_RESULT mylib_myclass_myBool_set(
  OV_INSTPTR_mylib_myclass pthis, OV_BOOL value
) {
  pthis->v_myBool = value;
  return OV_ERR_OK;
}

OV_RESULT mylib_myclass_mySingle_set(
  OV_INSTPTR_mylib_myclass pthis, OV_SINGLE value
) {
  if((value >= VALUE_MAX) && (value <= VALUE_MIN)) {
    pthis->v_mySingle = value;
    return OV_ERR_OK;
  }
  return OV_ERR_BADVALUE;
```

```
}
```

8.4.2 Zugriffsfunktionen für komplexe Variablen

Alle komplexen Variablen, d.h. Vektoren mit fester oder variabler Länge sowie (skalare) STRING-, C_TYPE <>- und PV-Variablen (BOOL_PV, INT_PV, SINGLE_PV), werden mit Hilfe von Zeigern auf den tatsächlichen Wert übergeben, so daß es kritisch ist, wem dieser Speicher gehört. Im Falle von schreibenden Zugriffen (set-Zugriffsfunktion) gehört der Speicher immer dem Aufrufer, wird von diesem verwaltet und von der Zugriffsfunktion nicht verändert. Im anderen Falle (get-Zugriffsfunktion) gilt folgende Konvention: Der Aufrufer initialisiert (blockiert) den Memory-Stack vor dem Aufruf der get-Zugriffsfunktion mit der Funktion ov_memstack_lock(), ruft dann die eigentliche set-Zugriffsfunktion auf, verarbeitet den zurückgegebenen Wert und gibt dann den Memory-Stack wieder mit ov_memstack_unlock() frei. Danach darf er auf den zurückgegebenen Wert nicht mehr zugreifen. Aufgrund dieser Konvention kann in der set-Zugriffsfunktion kann zur Bereitstellung des Wertes beliebig viel Speicher (nicht nur für den eigentlichen zur Verfügung gestellten Wert) mit Hilfe der Funktion ov_memstack_alloc() angefordert werden. Dieser muß und kann von Seiten der set-Zugriffsfunktion nicht freigegeben werden (dies geschieht implizit durch den Aufruf der Funktion mit ov_memstack_unlock() auf Seite des Anwenders). Der zurückgegebene Zeiger auf den Wert kann aber auch auf Bereiche in der Datenbasis oder statische allokierte Speicherbereiche verweisen, ohne daß weitere Maßnahmen getroffen werden müssen.

Im Falle von Vektoren fester oder variabler Länge wird stets noch die Länge des gelesenen oder zu setzenden Vektors übertragen (im Falle einer set-Zugriffsfunktion über eine Zeiger auf den Wert).

Beispiele:

```
#include "libov/ov_memstack.h"
#include "libov/ov_macros.h"
#include "libov/ov_string.h"

/* get accessor functions */

OV_STRING mylib_myclass_myString_get(OV_INSTPTR_mylib_myclass pthis) {
  OV_STRING retval;
  switch(pthis->v_currState) { /* some integer variable */
  case 0:
    return pthis->v_myString; /* string is valid */
  case 1:
    retval = (OV_STRING)ov_memstack_alloc(27);
    if(!retval) {
      return "out of memory";
    }
    strcpy(retval, "abcdefghijklmnopqrstuvwxyz");
    return retval;
  default:
    break;
  }
  return "unknown state";
}

OV_SINGLE_PV  *mylib_myclass_mySinglePV_get(OV_INSTPTR_mylib_myclass  pthis)
{
  return &pthis->v_mySinglePV;
}
```

```
OV_INT *mylib_myclass_myFixedIntVec_get(
  OV_INSTPTR_mylib_myclass pthis,
  OV_UINT                  *pveclen
) {
  *pveclen = 4;
  return &pthis->v_myFixedIntVec;
}

OV_TIME *mylib_myclass_myDynamicTimeVec_get(
  OV_INSTPTR_mylib_myclass pthis,
  OV_UINT                  *pveclen
) {
  OV_TIME *ptimes;
  OV_UINT i;
  if(pthis->v_Condition) {
    *pveclen = pthis->v_myDynamicTimeVec.veclen;
    return pthis->v_myDynamicTimeVec.value;
  } else {
    ptimes = (OV_TIME *)ov_memstack_alloc(5*sizeof(OV_TIME));
    if(ptimes) {
      *pveclen = 5;
      for(i=0; i<5; i++) {
        ptimes[i].secs = DEFAULT_TIME_SECS;
        ptimes[i].usecs = 0;
      }
      return ptimes;
    }
  }
  return NULL; /* no memory */
}

/* set accessor functions */

OV_RESULT mylib_myclass_myString_set(
  OV_INSTPTR_mylib_myclass pthis,
  const OV_STRING          value
) {
  return ov_string_setvalue(&pthis->v_myString, value);
}

OV_RESULT mylib_myclass_mySinglePV_set(
  OV_INSTPTR_mylib_myclass pthis,
  const OV_SINGLE_PV       *pvalue
) {
  pthis->v_mySinglePV = *pvalue;
  return OV_ERR_OK;
}

OV_RESULT mylib_myclass_myFixedIntVec_set(
  OV_INSTPTR_mylib_myclass pthis,
  const OV_INT             *pvalue,
  OV_UINT                  veclen
) {
  return Ov_SetStaticVectorValue(&pthis->v_myFixedIntVec,
    pvalue, veclen, INT);
}

OV_RESULT mylib_myclass_myDynamicTimeVec_set(
  OV_INSTPTR_mylib_myclass pthis,
  const OV_TIME            *pvalue,
  OV_UINT                  veclen
) {
  return Ov_SetDynamicVectorValue(&pthis->v_myDynamicTimeVec,
```

```
    pvalue, veclen, TIME);
}
```

8.5 Zugriff auf Variablen anderer Objekte

Wie bereits erwähnt, sollte der Zugriff auf die Instanzvariablen eines anderen Objekts prinzipiell immer über die Zugriffsfunktionen erfolgen, damit das Geheimnisprinzip nicht verletzt wird. Dies läßt sich mit Sprachmitteln von ANSI C allerdings nicht sicherstellen. Grundsätzlich kann man dieses Prinzip ohnehin umgehen (in C++ z.B. über „public“ definierte Variablen, d.h. data members), man sollte aber genau wissen, was man tut.

Beim Verwenden der Zugriffsfunktionen gelten die gleichen Konventionen, wie sie in Abschnitt 8.4, „Programmieren der Zugriffsfunktionen“, beschrieben wurden, d.h. beim Zugriff auf komplexe Variablen, d.h. Vektoren mit fester oder variabler Länge sowie (skalare) `STRING`-, `C_TYPE <>`- und PV-Variablen sind die Konventionen über die Speicherbereitstellung einzuhalten. Im Falle von schreibenden Zugriffen (set-Zugriffsfunktion) gehört der Speicher immer dem Aufrufer, wird von diesem verwaltet und von der Zugriffsfunktion nicht verändert. Im anderen Falle (get-Zugriffsfunktion) muß vor dem Aufruf der get-Zugriffsfunktion der Memory-Stack-Speicher mit der Funktion ov_memstack_lock() initialisiert (blockiert) werden. Nach dem Aufruf der eigentlichen set-Zugriffsfunktion auf kann der erhaltene Wert verarbeitet werden. Anschließend muß der Memory-Stack wieder mit ov_memstack_unlock() freigegeben werden.

9 Parts – in Objekte eingebettete Objekte

Parts sind Objekte, die in Objekte eingebettet werden. Sie stellen eine Möglichkeit dar, bestimmte Elemente eines Modells in anderen Klassen wiederzuverwenden, ohne daß dazu Vererbungsmechanismen genutzt werden. Dazu können bestimmte Aufgaben an die eingebetteten Objekte delegiert werden (Stichwort „Delegation statt Vererbung“). Parts werden als eingebettete Objekte automatisch mit dem umgebenden (einbettenden) Objekt instanziiert und gelöscht, sind also Komponenten im Sinne einer Kompositionsbeziehung.

9.1 Definition von Parts in der OV-Modell-Datei

Die Definition von Parts innerhalb einer Klasse sieht wie folgt aus:

```
LIBRARY mylib
  ...
  CLASS myclass ...
    ...
    PARTS
      mypart : CLASS anylib/anyclass [FLAGS=”a”];
    END_PARTS;
    ...
  END_CLASS;
END_LIBRARY;
```

Parts sind eingebettete Objekte einer anzugebenden Klasse, deren Rolle durch semantische Flags weiter klassifiziert werden kann.

9.2 Repräsentation von Parts in ANSI C

Parts werden in ANSI C repräsentiert, indem ihre Instanzdaten in die Instanzdaten des umgebenden Objekts eingebettet werden. Die Datenstruktur, die der C-Code-Generator erzeugt, sieht in etwa so aus:

```
typedef struct {
  /* instance data of ov/object */
  OV_INSTPTR v_pouterobject;
  ...
  /* instance data added in anylib/anyclass */
  ...
} OV_INST_anylib_anyclass;

typedef struct {
  ...
  OV_INST_anylib_anyclass p_mypart;
  ...
} OV_INST_mylib_myclass;
```

9.3 Zugriff auf eingebettete Objekte

Der Zugriff auf eingebettete Objekte geschieht über die Datenstruktur des umgebenden (einbettenden) Objekts:

```
OV_INSTPTR               pthis; /* given pointer to the embedding object */
OV_INSTPTR_mylib_myclass pobj;  /* unknown ptr to the embedded object */

...

pobj = &pthis->p_mypart;         /* get the ptr to the embedded object */
```

Der Typ (die Klasse) des eingebetteten Objekts ist bereits zur Compile-Zeit bekannt, da diese in der OV-Modell-Datei festgelegt ist.

9.4 Zugriff auf das umgebende Objekt

Der Zugriff von einem eingebetteten Objekt auf das umgebende (einbettende Objekt) erfolgt über einen Zeiger in den Instanzdaten des eingebetteten Objekts, der durch das OV-System verwaltet wird:

```
OV_INSTPTR_mylib_myclass pthis; /* given pointer to the embedded object */
OV_INSTPTR               pobj;  /* unknown ptr to the embedding object */

...

pobj = pthis->v_pouterobject;   /* get the ptr to the embedding object */
```

Wie man sieht, ist der Typ (die Klasse) des umgebenden Objekts i.A. unbekannt – bekannt ist nur, daß es mindestens von der Klasse „ov/object“ ist, wie jedes Objekt. Ggf. muß die Klasse zur Laufzeit ermittelt werden, z.B. mit dem Makro Ov_DynamicPtrCast.

10 Assoziationen und Links

Assoziationen definieren (mögliche) Beziehungen zwischen Instanzen zweier Klassen, der „Vater"- und der „Kind"-Klasse, die bestimmte „Vater"- und „Kind"-Rollen spielen und definieren einen Beziehungstyp. Eine tatsächlich vorhandene Beziehung zwischen Objekten ist quasi eine „Instanz" einer Assoziation und wird Link genannt. Liegt ein Link zwischen zwei Objekten vor, so besitzt jedes Objekt einen Zeiger (eine Referenz) auf des jeweils entgegengesetzt vorhandene Objekt. Das OV-System übernimmt die Verwaltung dieser Zeiger und sorgt dafür, daß alle Referenzen konsistent bleiben. Zwei Links des selben Typs zwischen denselben beiden Objekten sind nicht möglich, auch nicht bei *n*:*m*-Assoziationen, da Links keine eigene Identität besitzen.

10.1 Definition von Assoziationen in der OV-Modell-Datei

Die Definition von Assoziationen innerhalb einer Bibliothek sieht wie folgt aus:

```
LIBRARY mylib

  ...

  CLASS anyclass1 ...
  END_CLASS;

  CLASS anyclass2 ...
  END_CLASS;

  ...

  ASSOCIATION myassoc1N : ONE_TO_MANY
    PARENT myparent1 : CLASS mylib/anyclass1;
    CHILD mychild1 : CLASS mylib/anyclass2;
  END_ASSOCIATION;

  ASSOCIATION myassocNM : MANY_TO_MANY
    PARENT myparent2 : CLASS mylib/anyclass2 COMMENT = "my cute daddy";
    CHILD mychild2 : CLASS mylib/anyclass1 FLAGS = "abx";
  END_ASSOCIATION;
  ...

END_LIBRARY;
```

Assoziationen sind entweder 1:1-Beziehungen (`ONE_TO_ONE`), 1:*n*-Beziehungen (`ONE_TO_MANY`) oder *n*:*m*-Beziehungen (`MANY_TO_MANY`). Angegeben werden die Rollennamen der „Vater"- und der „Kind"-Klasse sowie die Klassenbezeichner dieser Klassen selbst. Die Rollennamen legen die Bezeichner der Linkenden in den Instanzdaten der Objektklassen fest. Den Rollen bzw. Linkenden kann jeweils ein Kommentar und semantische Flags zugeordnet werden.

10.2 Repräsentation von Assoziationen in ANSI C

Im Gegensatz zu den Klassenattributen werden die Instanzdatenstrukturen der betroffenen Klassen nicht verändert, da das OV die möglichen Assoziationen in dynamischen Tabellen verwaltet. Die Tabellenposition der jeweiligen Linkenden wird erst beim Laden der entsprechenden Assoziationen in das OV-System vergeben und an die bereits bestehenden Assoziati-

onsstrukturen zwischen den geladenen Klassen angepasst. Die damit verbundene Erweiterung oder auch Umstrukturierung der Linktabellen wird nicht nur auf Klassenebene sondern auch auf Instanzebene vorgenommen, d.h. das das Laden von neuen Assoziationen auch zwischen bereits instantiierten Klassen möglich ist.

10.3 Implementierung von Assoziationen

Assoziationen werden fast vollständig durch das OV-System bzw. den C-Code-Generator implementiert. Es gibt jedoch drei Funktionen, die durch den Anwender definiert werden können. Sie dienen dazu, das Verhalten einer Assoziation beeinflussen zu können. Für alle dieser Funktionen gibt es eine Default-Implementation, die über ein Makro verwendet werden kann, wenn nur das Standard-Verhalten gewünscht ist.

Beispiel:

```
#include "mylib.h"
#include "libov/ov_association.h"
#include "libov/ov_macros.h"

/* default implementations for assoc1 */

OV_IMPL_LINK(mylib_myassoc1N)

OV_IMPL_UNLINK(mylib_myassoc1N)

OV_IMPL_GETACCESS(mylib_myassoc1N)

/* our own implementation for assoc2 */

OV_DECL_LINK(mylib_myassocNM) {
  OV_RESULT result;
  result = ov_association_link(passoc_mylib_myassocNM,
    Ov_PtrUpCast(ov_object, pparent), Ov_PtrUpCast(ov_object, pchild),
    parenthint, Ov_PtrUpCast(ov_object, prelparent), childhint,
    Ov_PtrUpCast(ov_object, prelchild));
  if(Ov_OK(result)) {
    /* some actions required after creating a new link */
    ...
  }
  return result;
}

OV_DECL_UNLINK(mylib_myassocNM) {
  ov_association_unlink(passoc_mylib_myassocNM,
    Ov_PtrUpCast(ov_object, pparent), Ov_PtrUpCast(ov_object, pchild));
  /* some actions required after removing a link */
  ...
}

OV_DECL_GETACCESS(mylib_myassocNM) {
  /* only grant read access, no right to link/unlink from */
  /* outside the implementation of the two associated classes */
  return OV_AC_READ;
}
```

Wie das Beispiel deutlich macht, dienen diese Funktionen dazu, bestimme Aktionen ausführen zu können, wenn ein Link angelegt oder gelöscht wird bzw. dazu, die Zugriffsrechte festzulegen (`OV_AC_READ`, `OV_AC_LINKABLE`, `OV_AC_UNLINKABLE`);

10.4 Verwenden von Assoziationen bzw. Links

10.4.1 Anlegen und Löschen von Links

Das Anlegen und Löschen von Links geschieht am einfachsten über Makros aus libov/ov_macros.h. Beim Anlegen kann bestimmt werden, ob ein „Kind“-Objekt am Anfang, am Ende, oder vor bzw. hinter einem anderen Objekt in die Liste der dem „Vater“-Objekt assoziierten Objekte eingefügt wird. Bei *n*:*m*-Assoziationen gilt das gleiche für das „Vater“-Objekt und deren assoziierten „Kind“-Objekte. Das Erzeugen von Links kann fehlschlagen (aber nur aufgrund von Speichermangel), das Löschen von Links nicht (korrekte Parameter vorausgesetzt).

Beispiel:

```
#include "libov/ov_macros.h"

...

OV_INSTPTR_mylib_anyclass1 pc1_1, pc1_2,pc1_3;
OV_INSTPTR_mylib_anyclass2 pc2_1, pc2_2,pc2_3;

...

/* Link 1:n */

/* default placement */
if(Ov_Fail(Ov_Link(mylib_myassoc1N, pc1_1, pc2_1))) {
  /* no link created */
  ...
}

/* OV_PMH_BEGIN or OV_PMH_END (or OV_PMH_DEFAULT) */
if(Ov_OK(Ov_LinkPlaced(mylib_myassoc1N, pc1_1, pc2_2, OV_PMH_BEGIN))) {
  /* success */
  ...
}

/* OV_PMH_BEFORE or OV_PMH_AFTER (or any other, using a NULL ptr) */
/* here: child pc2_3 before pc2_2; no check for success/failure */
Ov_LinkRelativePlaced(mylib_assoc1, pc1_1, pc2_3, OV_PMH_BEFORE, pc2_2);

/* Unlink 1:n */

Ov_Unlink(mylib_myassoc1N, pc1_1, pc2_1);
Ov_Unlink(mylib_myassoc1N, pc1_1, pc2_2);
Ov_Unlink(mylib_myassoc1N, pc1_1, pc2_3);

/* Link n:m */

/* default placement */
Ov_LinkNM(mylib_myassocNM, pc2_1, pc1_1);

/* OV_PMH_BEGIN or OV_PMH_END (or OV_PMH_DEFAULT) for each object */
Ov_LinkPlacedNM(mylib_myassocNM, pc2_1, pc2_2, OV_PMH_BEGIN, OV_PMH_END);

/* OV_PMH_BEFORE or OV_PMH_AFTER (or any other, using a NULL ptr) */
/* here: child pc1_3 after pc1_2, parent pc2_1 default placement */
Ov_LinkRelativePlacedNM(mylib_myassocNM, pc2_1, pc1_3, OV_PMH_DEFAULT,
  NULL, OV_PMH_AFTER, pc1_2);
```

```
/* Unlink 1:n */

Ov_UnlinkNM(mylib_myassocNM, pc1_1, pc2_1);
Ov_UnlinkNM(mylib_myassocNM, pc1_1, pc2_2);
Ov_UnlinkNM(mylib_myassocNM, pc1_1, pc2_3);
```

Die gezeigten Makros beinhalten bereits alle notwendigen Typkonvertierungen, die für die übergebenen Instanzdatenzeiger notwendig sind (so weit sie zu Compile-Zeit schon möglich sind).

10.4.2 Iterieren über Links

Die mit einem Objekt über einen Link bestimmten Typs (einer bestimmten Assoziationen) assoziierten Objekte erhält man durch einen Iterationsprozess. Eine Ausnahme bildet die Abfrage des „Vater“-Objekts in einer 1:*n*-Beziehung, denn dort gibt es nur (maximal) ein Objekt.

Beispiel:

```
#include "libov/ov_macros.h"

...

OV_INSTPTR_mylib_anyclass1 pc1;
OV_INSTPTR_mylib_anyclass2 pc2;
/* define pit as an iterator for n:m associations */
Ov_DefineIteratorNM(mylib_myassocNM, pit);

/* Get parent of a 1:n association */

pc1 = Ov_GetParent(mylib_myassoc1N, pc2);

/* Get children of a 1:n association */

/* forward iteration */
pc2 = Ov_GetFirstChild(mylib_myassoc1N, pc1);
while(pc2) {
  /* pc2 points to the current child */
  ...
  pc2 = Ov_GetNextChild(mylib_myassoc1N, pc2);
}

/* identical to previous lines, but shorter */
Ov_ForEachChildNM(mylib_myassco1N, pc1, pc2) {
  /* pc2 points to the current child */
  ...
}

/* backwards iteration */
pc2 = Ov_GetLastChild(mylib_myassoc1N, pc1);
while(pc2) {
  /* pc2 points to the current child */
  ...
  pc2 = Ov_GetPreviousChild(mylib_myassoc1N, pc2);
}

/* Get parents of a n:m association */

/* forward iteration */
pc2 = Ov_GetFirstParentNM(mylib_myassocNM, pit, pc1);
while(pc2) {
```

```
  /* pc2 points to the current parent */
  ...
  pc2 = Ov_GetNextParentNM(mylib_myassocNM, pit);
}

/* identical to previous lines, but shorter */
Ov_ForEachParentNM(mylib_myasscoNM, pit, pc1, pc2) {
  /* pc2 points to the current parent */
  ...
}

/* backwards iteration */
pc2 = Ov_GetLastParentNM(mylib_myassocNM, pit, pc1);
while(pc2) {
  /* pc2 points to the current parent */
  ...
  pc2 = Ov_GetPreviousParentNM(mylib_myassocNM, pit);
}

/* Get children of a n:m association */

/* forward iteration */
pc1 = Ov_GetFirstChildNM(mylib_myassocNM, pit, pc2);
while(pc1) {
  /* pc1 points to the current child */
  ...
  pc1 = Ov_GetNextChildNM(mylib_myassocNM, pit);
}

/* identical to previous lines, but shorter */
Ov_ForEachChildNM(mylib_myasscoNM, pit, pc2, pc1) {
  /* pc1 points to the current child */
  ...
}

/* backwards iteration */
pc1 = Ov_GetLastChildNM(mylib_myassocNM, pit, pc2);
while(pc1) {
  /* pc1 points to the current child */
  ...
  pc1 = Ov_GetPreviousChildNM(mylib_myassocNM, pit);
}
```

Die gezeigten Makros beinhalten bereits alle notwendigen Typkonvertierungen, die für die übergebenen Instanzdatenzeiger notwendig sind (so weit sie zu Compile-Zeit schon möglich sind).

11 Aktive Objekte

Ein aktives Objekt ist ein Objekt, das nicht nur auf Ereignisse von außen reagiert (Lesen und Schreiben von Variablen, Methodenaufrufe), sondern selbständig bestimmte Aktionen ausführt. OV bietet aus verschiedenen Gründen (Micro-Controller-Kompatibilität, Synchronisationsprobleme, Stabilität des Systemzustands etc.) keine Threads, d.h. mehrere unabhängige Ausführungs-Kontrollstränge, an. Dafür gibt es ein Konzept, den OV-Scheduler, das „scheinbar“ aktive Objekte ermöglicht. Dieses Konzept ist „kooperativ“, erfordert also die Mithilfe des Anwendungsprogrammierers.

11.1 Der Scheduler

Um ein aktives Objekt zu erhalten, wird es zunächst beim OV-Scheduler angemeldet. Dieser sorgt nun dafür, daß eine bestimmte Methode des Objekts zu beliebig wählbaren Zeitpunkten aktiviert wird. Diese Methode darf jedoch nicht unbegrenzt lange laufen, sondern sollte nur „kurze“ Zustandsübergänge durchführen. Nach einem solchen Zustandsübergang muß wird die neue Bearbeitungszeit gewählt (absoluter Zeitpunkt, relativer Zeitpunkt oder – per default – „sobald wie möglich“) und die Kontrolle wieder an den OV-Scheduler zurückgegeben. Eine zyklische Bearbeitung ist natürlich möglich, aber ein Sonderfall.

Um Objektmethoden zu den gewünschten Zeitpunkten aufrufen zu können, werden diese vom OV-Scheduler in eine zeitlich geordnete Warteschlange eingereiht. Zu den entsprechenden Zeiten werden dann die Objekte aktiviert. Zwischen solchen Aufrufen werden ggf. Dienste des KS-Servers und des KS-Klienten bearbeitet. Steht nichts an, so legt sich der Prozeß schlafen. Nachteil dieses kooperativen Verfahrens ist, daß jede Objektmethode den OV-Scheduler im Prinzip beliebig lange blockieren kann, so daß bestimmte Objektmethoden u.U. nicht rechtzeitig bearbeitet werden.

Typischerweise wird sich ein aktives Objekt in seiner „startup“-Methode beim OV-Scheduler anmelden und in seiner „shutdown“-Methode wieder abmelden. Hierzu ein Beispiel:

OV-Modell-Datei:

```
/*
*   File: mylib.ovm
*/

#include "ov.ovm"

LIBRARY mylib
  ...
  CLASS myclass : ...
    VARIABLES
      ...
      state : INT;
      ...
    END_VARIABLES;
    OPERATIONS
      startup : C_FUNCTION <OV_FNC_STARTUP>;
      shutdown : C_FUNCTION <OV_FNC_SHUTDOWN>;
      ...
    END_OPERATIONS;
    ...
  END_CLASS;
```

```
  ...
END_LIBRARY;
```

C-Quellcode-Datei:

```
/*
*   File: myclass.c
*/

#include "mylib.h"
#include "libov/ov_scheduler.h"
#include "libov/ov_macros.h"

/*
*   Declaration of the method executed (triggered) by the scheduler
*/
void mylib_myclass_execute(OV_INSTPTR_ov_object pobj);

/*
*   Startup the object: register with the scheduler
*/
OV_DLLFNCEXPORT void mylib_myclass_startup(OV_INSTPTR_ov_object pobj) {
  ov_object_startup(pobj);
  Ov_WarnIfNot(Ov_OK(ov_scheduler_register(pobj, mylib_myclass_execute)));
}

/*
*   Shutdown the object: unregister
*/
OV_DLLFNCEXPORT void mylib_myclass_shutdown(OV_INSTPTR_ov_object pobj) {
  ov_scheduler_unregister(pobj);
  ov_object_shutdown(pobj);
}

/*
*   The method executed (triggered) by the scheduler
*/
void mylib_myclass_execute(OV_INSTPTR_ov_object pobj) {
  /* local variables */
  OV_TIME_SPAN             ts_1s = { 1, 0 };
  OV_TIME                  t_forever = { OV_VL_MAXUINT, 0 };
  /* do something */
  printf("execute method triggered!\n");
  ...
  /* reschedule (set a next event) */
  switch(...) {
  case ...:
    /* next action (transition) in one second */
    ov_scheduler_setreleventtime(pobj, &ts_1s);
    break;
  case ...:
    /* no next transition -- unless we change (re-set) the event time */
    /* in some other method */
    ov_scheduler_absreleventtime(pobj, &t_forever);
    break;
  default:
    /* doing nothing means: reschedule as soon as possible */
    break;
  }
}
```

11.2 Zustandsmaschinen

In der Implementierung der durch den OV-Scheduler getriggerten „aktiven“ Methode beginnt die Bearbeitung immer wieder von vorn. Zweck jeder Bearbeitung ist die Durchführung eines Zustandsübergangs. In quasi-kontinuierlich arbeitenden Objekten ist dies recht einfach, wenn zyklisch immer wieder ein bestimmter Algorithmus durchlaufen werden muß. Häufig ist aber ein bestimmter diskreter Zustand definiert und es wäre schön, die Bearbeitung jeweils an einer dem aktuellen Zustand entsprechenden Stelle weiterführen zu können. Dieser diskrete Zustand muß natürlich im Objekt gespeichert sein, damit er beim nächsten Aufruf wieder verfügbar ist.

11.2.1 Eine einfache Zustandsmaschine

Im einfachsten Falle erreicht man dies durch die Definition einer internen Zustandsvariable (beispielsweise ein Integer) und eine switch-case-Kontrollstruktur in der „aktiven“ Methode:

OV-Modell-Datei:

```
/*
*   File: mylib.ovm
*/

#include "ov.ovm"

LIBRARY mylib
  ...
  CLASS myclass : ...
    VARIABLES
      ...
      state : INT;
      ...
    END_VARIABLES;
    ...
  END_CLASS;
  ...
END_LIBRARY;
```

C-Quellcode-Datei:

```
/*
*   File: myclass.c
*/

#include "mylib.h"
#include "libov/ov_scheduler.h"
#include "libov/ov_macros.h"

...

/*
*   The method executed (triggered) by the scheduler
*/
void mylib_myclass_execute(OV_INSTPTR_ov_object pobj) {
  /* local variables */
  OV_INSTPTR_mylib_myclass pthis = Ov_StaticPtrCast(mylib_myclass, pobj);
  OV_TIME                  t_forever = { OV_VL_MAXUINT, 0 };
  OV_TIME_SPAN             ts_1s = { 1, 0 };
  /* the state machine */
```

```
  switch(pthis->v_state) {
  case 0: /* first state ------------------------------------------- */
    ...
    /* if condition holds, do a transition */
    if(...) pthis->v_state = 1; /* next time, continue with second state */
    /* reschedule (set next event in one second) */
    ov_scheduler_setreleventtime(pobj, &ts_1s);
    break;                      /* still first state next time */
  case 1: /* second state ------------------------------------------ */
    ...
    /* fall through... */       /* enter third state now */
  case 2: /* third state ------------------------------------------- */
    ...
    /* if condition holds, do a transition */
    if(...) pthis->v_state = 0; /* next time, continue with first state */
    /* reschedule (set next event in one second) */
    ov_scheduler_setreleventtime(pobj, &ts_1s);
    break;
  default: /* error state ------------------------------------------ */
    /* when restarted, restart with first state */
    pthis->v_state = 0;
    ov_scheduler_setabseventtime(pobj, &t_forever);
    break;
  }
}
```

11.2.2 Eine etwas elegantere Zustandsmaschine

Der Nachteil der Zustandsmaschine aus dem vorherigen Abschnitt ist, daß diese Zustandsmaschine relativ schwer leserlich ist und daß u.U. viele Vergleiche in der switch-case-Anweisung erforderlich sind. Etwas eleganter ist die Verwendung der setjmp/longjmp-Anweisungen aus der Systemheaderdatei setjmp.h, die es gestattet, den Ausführungszustand des Rechners zu speichern und direkt wieder anzuspringen. Dies zeigt das folgende Beispiel:

OV-Modell-Datei:

```
/*
*   File: mylib.ovm
*/

#include "ov.ovm"

LIBRARY mylib
  ...
  CLASS myclass : ...
    VARIABLES
      ...
      state : C_TYPE <INTERNAL_STATE>;
      ...
    END_VARIABLES;
    ...
  END_CLASS;
  ...
END_LIBRARY;
```

OV-Datentyp-Datei:

```
/*
*   File: mylib.ovt
*/
```

```
#ifndef MYLIB_OVT_INCLUDED
#define MYLIB_OVT_INCLUDED

#include <setjmp.h>

/*
*    Jump buffer and associated macros for use with state machines
*/
#if OV_SYSTEM_UNIX
typedef sigjmp_buf            PROFILIB_JMPBUF;
#define Mylib_SetJmp(jmpbuf)  sigsetjmp(jmpbuf, 1)
#define Mylib_LongJmp(jmpbuf) siglongjmp(jmpbuf, 1)
#else
typedef jmp_buf               PROFILIB_JMPBUF;
#define Mylib_SetJmp(jmpbuf)  setjmp(jmpbuf)
#define Mylib_LongJmp(jmpbuf) longjmp(jmpbuf, 1)
#endif

/*
*   Internal state
*/
typedef struct {
  OV_BOOL      sm_initialized;
  MYLIB_JMPBUF sm_jmpbuf;
} INTERNAL_STATE;

#endif /* MYLIB_OVT_INCLUDED */
```

C-Quellcode-Datei:

```
/*
*   File: myclass.c
*/

#include "mylib.h"
#include "libov/ov_scheduler.h"
#include "libov/ov_macros.h"

...

/*
*   Helper macros for the state machine
*/
#define InitStateMachine \
  if(pthis->v_state.sm_initialized) { \
    Mylib_LongJmp(ppm->v_state.sm_jmpbuf); \
  } \
  pthis->v_state.sm_initialized = TRUE

#define EnterState(label) \
  sm_state_##label: Mylib_SetJmp(pthis->v_state.sm_jmpbuf))

#define GotoState(label) \
  goto sm_state_##label

#define ExitStateMachine \
  return

#define ExitStateMachineFor(ts) \
  ov_scheduler_setreleventtime(pobj, &ts); return

#define ExitStateMachineForever \
```

```
  ov_scheduler_setabseventtime(pobj, &t_forever); return

/*
*   The method executed (triggered) by the scheduler
*/
void mylib_myclass_execute(OV_INSTPTR_ov_object pobj) {
  /* local variables */
  OV_INSTPTR_mylib_myclass pthis = Ov_StaticPtrCast(mylib_myclass, pobj);
  OV_TIME                  t_forever = { OV_VL_MAXUINT, 0 };
  OV_TIME_SPAN             ts_1s = { 1, 0 };
  /* the state machine */
  InitStateMachine;
  EnterState(FirstState); /* first state ------------------------------ */
  /* this state is entered the first time */
  ...
  /* if condition holds, do a transition */
  if(...) GotoState(State2);
  ExitStateMachineFor(ts_1s); /* execute first state again in one second */
  EnterState(State2); /* second state ---------------------------------- */
  ...
  /* automatically fall through into the third state */
  EnterState(The3rdState); /* third state ----------------------------- */
  ...
  /* if condition holds, do a transition */
  if(...) GotoState(FirstState);
  if(error) GotoState(ErrorState);
  ExitStateMachineFor(ts_1s); /* execute third state again in one second */
  EnterState(Error); /* error state ----------------------------------- */
  ExitStateMachineForever;
}
```

12 Introspektion und Reflektion

Introspektion bedeutet, daß die internen Eigenschaften einer Klasse oder Assoziation von außen erkundet werden (z.B. beim Erkunden der Systemstruktur mit dem ACPLT Magellan); Reflektion besagt, daß dies aus der Implementierung der Klasse selbst heraus geschieht. Introspektion und Reflektion sind damit nichts anderes als der Zugriff auf Metainformation aus unterschiedlichen Sichten.

Die in einem OV-System vorhandenen Metainformation ist sämtliche in den OV-Modell-Dateien enthaltene Information. Sie umfaßt Informationen über

- die geladenen Bibliotheken,
- die zentralen Eigenschaften aller geladenen Klasse,
- die Variablen einer Klasse,
- die Parts einer Klasse,
- die Operationen einer Klasse sowie
- die geladenen Assoziationen.

Diese Informationen können zur Laufzeit abgefragt und verwendet werden. Sie liegen in Form von Instanzdaten vor, denn Bibliotheken, Klassen und deren Variablen-, Part- und Operationsdefinitionen sowie Assoziationen sind selbst Objekte (genauer: Meta-Objekte). Die Klassen der Metaobjekte sind im Metamodell von OV festgelegt (ov.ovm).

12.1 Metainformationen der Metaobjekte

12.1.1 Bibliotheksobjekte

Ein Bibliotheksobjekt besitzt folgende Instanzdaten:

```
typedef struct {
  ...
  /* variables */
  OV_STRING v_version;   /* library version number/info */
  OV_STRING v_author;    /* author of the library */
  OV_STRING v_copyright; /* copyright information */
  OV_STRING v_comment;   /* comment about the library */
  ...
} OV_INST_ov_library;
typedef OV_INST_ov_library* OV_INSTPTR_ov_library;
```

Diese Informationen (version, author, copyright und comment) stammen aus dem Kopf einer Bibliotheksdefinition in der OV-Modell-Datei.

12.1.2 Klassenobjekte

Ein Klassenobjekt besitzt folgende Instanzdaten:

```
typedef struct {
  ...
  /* variables */
  OV_STRING                          v_comment;    /* class comment */
  OV_UINT                            v_flags;      /* semantic flags */
  OV_INT                             v_classprops; /* class properties */
  ...
```

```
  ...
} OV_INST_ov_class;
typedef OV_INST_ov_class* OV_INSTPTR_ov_class;
```

Kommentar (comment), semantische Flags (flags) und Klasseneigenschaften (classprops) stammen aus dem Kopf einer Klassendefinition in der OV-Modell-Datei. Die Klasseneigenschaften werden OR-verknüpft und sind wie folgt definiert:

```
#define OV_CP_INSTANTIABLE 0x00000001 /* class is instantiable */
#define OV_CP_FINAL        0x00000002 /* class cannot be subclassed */
```

Darüber hinaus sind Links zu den Instanzen der Klasse („Kind"-Objekte in der Assoziation „ov/instantiation") – sofern die Klasse instanziierbar ist –, zu den abgeleiteten Klassen und zur Basisklasse („Kind"-Objekte bzw. „Vater"-Objekt in der Assoziation „ov/inheritance"), zu den Assoziations-Definitionsobjekten, in denen die Klasse „Vater" oder „Kind" ist („Kind"-Objekte der Assoziation „ov/parentrelationship" bzw. „ov/childrelationship") und zu den Part-Objekten, die diese Klasse als Part-Klasse definieren („Kinder" der Assoziation „ov/embedment"), vorhanden.

12.1.3 Variablen-Definitionsobjekte einer Klasse

Ein Variablen-Definitionsobjekt besitzt folgende Instanzdaten:

```
typedef struct {
  ...
  /* variables */
  OV_INT        v_vartype;   /* data type of the variable */
  OV_STRING     v_ctypename; /* C type name in case of a C_TYPE variable */
  OV_INT        v_varprops;  /* variable properties */
  OV_UINT       v_veclen;    /* vector length */
  OV_STRING     v_comment;   /* comment */
  OV_UINT       v_flags;     /* semantic flags */
  OV_STRING     v_techunit;  /* technical unit */
  OV_UINT       v_size;      /* size of instance data */
  OV_UINT       v_offset;    /* offset in the instance data */
  OV_FNCPTR_GET v_getfnc;    /* pointer to the get accessor function */
  OV_FNCPTR_SET v_setfnc;    /* pointer to the set accessor function */
  ...
} OV_INST_ov_variable;
typedef OV_INST_ov_variable* OV_INSTPTR_ov_variable;
```

Variablentyp (vartype), C-Datentypname (ctypename), Variableneigenschaften (varprops), Vektorlänge (veclen), Kommentar (comment), semantische Flags (flags) und die technische Einheit (techunit) stammen aus der Variablendefinition in der OV-Modell-Datei. Als Variablentypen sind definiert:

```
#define OV_VT_VOID          ...
#define OV_VT_BOOL          ...
#define OV_VT_INT           ...
#define OV_VT_UINT          ...
#define OV_VT_SINGLE        ...
#define OV_VT_DOUBLE        ...
#define OV_VT_STRING        ...
#define OV_VT_TIME          ...
#define OV_VT_TIME_SPAN     ...
#define OV_VT_BOOL_VEC      ...
#define OV_VT_INT_VEC       ...
#define OV_VT_UINT_VEC      ...
```

```
#define OV_VT_SINGLE_VEC    ...
#define OV_VT_DOUBLE_VEC    ...
#define OV_VT_STRING_VEC    ...
#define OV_VT_TIME_VEC      ...
#define OV_VT_TIME_SPAN_VEC ...
#define OV_VT_BYTE          ... /* C-type variable */
#define OV_VT_BYTE_VEC      ... /* C-type variable */
#define OV_VT_BOOL_PV       ...
#define OV_VT_INT_PV        ...
#define OV_VT_UINT_PV       ...
#define OV_VT_SINGLE_PV     ...
#define OV_VT_DOUBLE_PV     ...
#define OV_VT_TIME_PV       ...
#define OV_VT_TIME_SPAN_PV  ...
#define OV_VT_STRING_PV     ...
```

Die Variableneigenschaften werden OR-verknüpft und sind wie folgt definiert:

```
#define OV_VP_GETACCESSOR 0x00000001 /* variable has a get accessor */
#define OV_VP_SETACCESSOR 0x00000002 /* variable has a set accessor */
#define OV_VP_ACCESSORS   (OV_VP_GETACCESSOR | OV_VP_SETACCESSOR)
#define OV_VP_DERIVED     0x00000004 /* variable is derived (virtual) */
```

Die Vektorlänge ist gleich Eins für einen Skalar (kein echter Vektor), Null für einen dynamischen Vektor und größer als Eins für einen Vektor mit fester Länge. Die Größe der Instanzdaten der Variable (size) und der Offset der Variable in den Instanzdaten (offset) wird mit Hilfe der C-Makros sizeof(type) und offsetof(type, member) durch den vom C-Code-Generator erzeugten Code berechnet. Die Variablen getfnc und setfnc sind Zeiger auf die definierten set- bzw. get-Zugriffsfunktionen oder NULL, falls diese nicht definiert wurden.

12.1.4 Parts-Definitionsobjekte einer Klasse

Ein Part-Definitionsobjekt besitzt folgende Instanzdaten:

```
typedef struct {
  ...
  /* variables */
  OV_UINT                   v_offset;    /* offset in the instance data */
  OV_UINT                   v_flags;     /* semantic flags of the part */
  ...
} OV_INST_ov_part;
typedef OV_INST_ov_part* OV_INSTPTR_ov_part;
```

Der zugehörige Link partclass zeigt auf das Klassen-Objekt, der in der Partdefinition in der OV-Modell-Datei definierten Klasse („Vater“-Objekt der Assoziation „ov/embedment“). Der Offset des eingebetteten Part-Objekts in den Instanzdaten wird mit Hilfe des C-Makros offsetof(type, member) durch den vom C-Code-Generator erzeugten Code berechnet.

12.1.5 Operations-Definitionsobjekte einer Klasse

Ein Operations-Definitionsobjekt besitzt folgende Instanzdaten:

```
typedef struct {
  ...
  /* variables */
  OV_INT    v_opprops;      /* operation properties */
  OV_STRING v_cfnctypename; /* C function type name (prototype name) */
  ...
} OV_INST_ov_operation;
```

```
typedef OV_INST_ov_operation* OV_INSTPTR_ov_operation;
```

Die Operationseigenschaften (opprops) und der C-Funktionsprototypname (cfnctypename) stammen aus der Operationsdefinition in der OV-Modell-Datei. Die Operationseigenschaften werden OR-verknüpft und sind wie folgt definiert:

```
#define OV_OP_ABSTRACT  0x00000001 /* operation is abstract (not impl.) */
```

12.1.6 Assoziationsobjekte

Ein Assoziationsobjekt besitzt folgende Instanzdaten:

```
typedef struct {
  ...
  /* variables */
  OV_INT                            v_assoctype;      /* assoc type */
  OV_STRING                         v_parentrolename; /* prnt role name */
  OV_STRING                         v_childrolename;  /* chld role name */
  OV_UINT                           v_parentoffset;   /* parnet offset */
  OV_UINT                           v_childoffset;    /* child offset */
  OV_STRING                         v_parentcomment;  /* parent comment */
  OV_STRING                         v_childcomment;   /* child comment */
  OV_UINT                           v_parentflags;    /* parent flags */
  OV_UINT                           v_childflags;     /* child flags */
  OV_FNCPTR_LINK                    v_linkfnc;        /* link fnc ptr */
  OV_FNCPTR_UNLINK                  v_unlinkfnc;      /* unlink fnc ptr */
  OV_FNCPTR_GETACCESS               v_getaccessfnc;   /* getaccess fptr */
...
} OV_INST_ov_association;
typedef OV_INST_ov_association* OV_INSTPTR_ov_association;
```

Der Assoziationstyp (assoctype), die Assoziationseigenschaften (assocprops), „Vater“- und „Kind“-Rollenname (parentrolename, childrolename), „Vater“- und „Kind“-Kommentar (parentcomment, childcomment) und semantische „Vater“- und „Kind“-Flags (parentflags, childflags) stammen aus der Assoziationsdefinition in der OV-Modell-Datei. Dabei sind die Assoziationstypen wie folgt definiert:

```
#define OV_AT_ONE_TO_MANY  0x00000001 /* 1:n association */
#define OV_AT_MANY_TO_MANY 0x00000002 /* n:m association */
```

„Vater“- und „Kind“-Offset der Datenstrukturen zur Aufnahme für einen Link benötigten Zeiger innerhalb der jeweiligen Instanzdaten werden mit Hilfe des C-Makros offsetof(type, member) durch den vom C-Code-Generator erzeugten Code berechnet. Die Zeiger linkfnc, unlinkfnc und getaccessfnc zeigen auf die für jede Assoziation zu implementierenden Funktionen (vgl. Abschnitt 10.3, „Implementierung von Assoziationen“). Darüber hinaus sind Links zu den der „Vater“- und der „Kind-„Klasse entsprechenden Klassenobjekten vorhanden (parentclass, childclass), d.h. die „Vater“-Objekte der „ov/parentrelationship“- bzw. der „ov/childrelationship"-Assoziation.

12.2 Die Struktur OV_ELEMENT

Die Struktur OV_ELEMENT enthält Werte, die es gestatten, jedes Element eines OV-Systems auf Instanzebene zu referenzieren. Zu diesen Elementen gehören nicht nur Objekte, sondern auch deren Variablen, Parts, Operationen und Linkenden. Die Struktur ist wie folgt definiert:

```
typedef struct {
  OV_ELEM_TYPE               elemtype; /* the type of the element */
  OV_INSTPTR_ov_object       pobj;     /* obj this element belongs to */
  OV_BYTE                    *pvalue;  /* pointer to variable value */
  OV_INSTPTR_ov_variable     *pvar;    /* pointer to class variable */
  union {
    OV_INSTPTR_ov_variable   pvar;   /* in case element is a variable */
    OV_INSTPTR_ov_part       ppart;  /* in case element is a part */
    OV_INSTPTR_ov_operation  pop;    /* in case element is an operation */
    OV_INSTPTR_ov_association passoc; /* in case element is a link end */
  } elemunion;
} OV_ELEMENT;
```

Was für eine Art von Element referenziert wird, wird durch das Strukturelement elemtype definiert, für das folgende Werte definiert sind:

```
enum OV_ELEM_TYPE_ENUM  {
  OV_ET_NONE       = 0x00, /* undefined (invalid element) */
  OV_ET_OBJECT     = 0x01,
  OV_ET_VARIABLE   = 0x02,
  OV_ET_MEMBER     = 0x04,
  OV_ET_PARENTLINK = 0x08,
  OV_ET_CHILDLINK  = 0x10,
  OV_ET_OPERATION  = 0x20,
  OV_ET_ANY        = 0x3F  /* used for search masks only */
};
typedef enum_t OV_ELEM_TYPE;
```

Das Objekt, zu dem das Element selbst gehört, wird durch das Strukturelement pobj referenziert.

Ist elemtype == OV_ET_OBJECT, so handelt es sich um die Referenz auf ein Objekt. Ist dieses Objekt ein Part-Objekt (pobj->v_pouterobject != NULL), so befindet sich in elemunion.ppart ein Verweis auf das Part-Definitionsobjekt, das die Metainformation zu dem Part bereitstellt. Handelt es sich um ein globales, nicht eingebettetes Objekte (pobj->v_pouterobject == NULL), so sind die Inhalte aller anderen Strukturelemente nicht definiert.

Für den Fall, daß es sich bei dem referenzierten Element um eine Variable handelt (elemtype == OV_ET_VARIABLE) zeigt pvalue unmittelbar auf den Wert innerhalb der Instanzdaten des Objekts (z.B. pvalue = (OV_BYTE *)&pthis->v_myvar für die Variable myvar) und elemunion.pvar auf das Variablen-Definitionsobjekt, das die Metainformation zu dieser Variable bereitstellt. Handelt es sich um eine Strukturelement (elemtype == OV_ET_MEMBER), so wird über pvar der Zeiger auf das Definitionsobjekt der Struktur-Variable der Klasse referenziert und elemunion.pvar verweist auf das Variablendefinitionsobjekt des entsprechenden Strukturelementes.

Handelt es sich bei dem referenzierten Element um eine Operation (elemtype == OV_ET_OPERATION), so ist pvalue nicht definiert und elemunion.pop zeigt auf das Operations-Definitionsobjekt der Operation.

Für den Fall, daß es sich bei dem referenzierten Element um ein „Vater“- oder „Kind“-Linkende handelt (elemtype == OV_ET_PARENTLINK oder elemtype == OV_ET_CHILDLINK), ist pvalue undefiniert. Gleichzeitig zeigt elemunion.passoc auf das entsprechende Assoziationsobjekt, das den Typ des Links angibt.

12.3 Generischer Zugriff auf die Elemente eines Objekts

Die Werte, die zur Referenzierung Objektelements in eine OV_ELEMENT-Struktur eingetragen werden müssen, brauchen zum Glück nicht vom Anwender selbst bereitgestellt werden. Dazu bietet das OV-System geeignete Funktionen an.

12.3.1 Suchen von Objektelementen

Das Suchen eines Objektelements gegebenen Typs und Namens geschieht mit Hilfe der Funktion ov_element_searchpart() aus der C-Header-Datei libov/ov_element.h:

```
#include "libov/ov_element.h"
#include "libov/ov_macros.h"

...

/* Search for an element of an object ("obj.myElement" in a path) */
OV_ELEMENT givenobject;
OV_ELEMENT searchedelement;

givenobject.elemtype = OV_ET_OBJECT;
givenobject.pobj = Ov_PtrUpCast(ov_object, pthis);
if(Ov_OK(ov_element_searchpart(&givenobject, &searchedelement, OV_ET_ANY,
 "myElement"))
) {
  /* element was found! */
  switch(searchedelement.elemtype) {
  case OV_ET_OBJECT: /* it is a (part) object */
    ...
    break;
  case OV_ET_VARIABLE: /* it is a variable */
    printf("found a variable, its unit is: %s\n",
      searchedelement.elemunion.pvar->v_unit);
    ...
    break;
  case OV_ET_PARENTLINK: /* it is a parent link end */
    ...
    break;
  case OV_ET_CHILDLINK: /* it is a child link end */
    ...
    break;
  case OV_ET_OPERATION: /* it is an operation */
    ...
    break;
  default:
    ...
    break;
  }
} else {
  /* no element with this name */
  ...
}
```

Das dritte Argument der Funktion ov_element_searchpart() ist eine Maske für die zu suchenden Elemente; hiermit kann die Suche eingeschränkt werden. Ist die Maske z.B. OV_ET_VARIABLE, so wird nur nach Variablen gesucht, ist sie OV_ET_PARENTLINK | OV_ET_CHILDLINK, so wird nur nach Linkenden gesucht.

12.3.2 Iterieren über Objektelemente

Das Iterieren über alle Objektelements gegebenen Typs geschieht mit Hilfe der Funktion ov_element_getnextpart() aus der C-Header-Datei libov/ov_element.h:

```
#include "libov/ov_element.h"
#include "libov/ov_macros.h"

...

/* Iterate over all elements of an object ("obj.*" in a path) */
OV_ELEMENT givenobject;
OV_ELEMENT iteratedelement;

givenobject.elemtype = OV_ET_OBJECT;
givenobject.pobj = Ov_PtrUpCast(ov_object, pthis);

iteratedelement.elemtype = OV_ET_NONE;
do {
  ov_element_getnextpart(&givenobject, &iteratedelement, OV_ET_ANY);
  switch(iteratedelement.elemtype) {
  case OV_ET_XXX:
    ...
    break;
  ...
  default:
    break;
  }
} while(iteratedelement.elemtype != OV_ET_NONE);
```

Das dritte Argument der Funktion ov_element_getnextpart() ist wiederum eine Maske für die zu suchenden Elemente; hiermit kann die Suche eingeschränkt werden. Ist die Maske z.B. OV_ET_VARIABLE, so wird nur nach Variablen gesucht, ist sie OV_ET_PARENTLINK | OV_ET_CHILDLINK, so wird nur nach Linkenden gesucht.

12.3.3 Auflösen von Pfadnamen

Besitzt man den Pfadnamen eines Objekts, z.B. „/ObjectContainer/MyObject.VarX“ (vgl. ACPLT/KS), so kann dieser mit Hilfe der Funktion ov_path_resolve() aus der C-Header-Datei libov/ov_path.h geschehen. Diese Funktion verwendet eine Struktur namens OV_PATH, die wie folgt aufgebaut ist:

```
typedef struct {
  OV_UINT    size;      /* number of path elements */
  OV_ELEMENT *elements; /* pointer to array of elements */
} OV_PATH;
```

Die Funktion erzeugt ein eindimensionales Feld von OV_ELEMENT-Werten, die jeweils einem durch „/“ oder „.“ getrennten Objektnamen (Identifier) im gegebenen Pfadnamen entsprechen. Den dazu notwendigen Speicher beschafft sich die Funktion intern mit Hilfe von ov_memstack_alloc(), so daß es notwendig ist, vor dem Aufrunf von ov_path_resolve() die Funktionen ov_memstack_lock() und später, nachdem nicht mehr auf die zurückgelieferten Elemente zugegriffen wird, ov_memstack_unlock() aufzurufen. Ein Pfadname kann relativ zu einem gegebenen Pfadnamen aufgelöst werden.

Beispiel:

```
#include "libov/ov_path.h"
```

```
#include "libov/ov_macros.h"

...

/* resolve an absolute and a relative path */
OV_PATH   abspath, relpath;
OV_RESULT res;

ov_memstack_lock();
/* resolve absolute path */
res = ov_path_resolve(&abspath, NULL, "/ObjectContainer/MyObject.VarX", 2);
if(Ov_OK(res)) {
  /* abspath.elements[abspath.size-1] now contains the OV_ELEMENT */
  /* value that describes the variable Var.X of MyObject contained */
  /* in ObjectContainer */
  ...
  /* resolve path relative to the first one */
  res = ov_path_resolve(&relpath, &abspath, „.VarY", 2);
  if(Ov_OK(res)) {
    /* relpath.elements[relpath.size-1] now contains the OV_ELEMENT */
    /* value that describes the variable Var.Y of MyObject contained */
    /* in ObjectContainer */
    ...
  }
}
ov_memstack_unlock();
```

Es gibt übrigens auch eine Funktion, die den (kanonische) Pfadname eines Objekts (nicht eines Objektelements) zurückliefert. Auch hier gilt die ov_memstack_lock()/ov_memstack_unlock()-Konvention:

```
#include "libov/ov_path.h"
#include "libov/ov_macros.h"

...

OV_INSTPTR_mylib_myclass pthis = ...;

ov_memstack_lock();
printf("pthis points to object: %s\n",
  ov_path_getcanonicalpath(Ov_PtrUpCast(ov_object, pthis), 2));
ov_memstack_unlock();
```

13 Datenbasis-Speicherverwaltung

In der Datenbasis werden die Instanzdaten der OV-Objekte (nicht jedoch mit einer Bibliothek geladener ausführbarer Code) gespeichert. Die Datenbasis ist normalerweise Datei, die per „File-Mapping“ in einem Stück in den Hauptspeicher gespiegelt wird. Änderungen in diesem Speicherbereich werden für den Anwender transparent durch das Betriebssystem in die Datei zurückgeschrieben. Dadurch sind die OV-Objekte persistent wenn das OV-System heruntergefahren und beendet wird – auch nach einem „kontrollierten Absturz“.

Wird ein OV-System auf einer Datenbasis erneut gestartet, so kann es passieren, daß das Betriebssystem dem „gemappten“ Speicherbereich eine andere Basisadresse zuweist, als beim letzten Mal. In diesem Moment sind alle in der Datenbasis gespeicherten Objektreferenzen (Zeiger) ungültig – die referenzierten Speicheradressen unterschieden sich von den tatsächlichen Adressen um einen gewissen Offset. Das OV-System kennt jedoch alle von ihm verwalteten Adressen (z.B. bei Links, Strings und dynamischen Vektoren) und kann diese Zeiger korrigieren. Alle nicht vom OV-System verwalteten Zeiger jedoch sind und bleiben nach dem „Mappen“ ungültig.

Benötigt der Anwender eigene Speicherbereiche, so hat dies folgende Konsequenz:

- Wird nicht-persistenter Speicher benötigt, z.B. mit malloc() und free() verwalteter Speicher, so „lebt“ dieser nur während der Laufzeit des OV-Systems. Am besten werden Zeiger auf solche Bereiche (die über `C_TYPE` <>-Variablen definiert werden) in der „startup“-Methode eines Objekts initialisiert und in der „shutdown“-Methode – nach Freigabe der Ressourcen – mit NULL belegt.
- Wird persistenter Speicher benötigt, der auch das Herunterfahren und erneute Starten des OV-Systems „überlebt“, so muß dieser von der Datenbasis bereitgestellt werden. Die sich die Basisadresse der Datenbasis nach einem erneuten Mappen verschoben haben kann, muß mit relativen Adressen, d.h. mit Offsets innerhalb der Datenbasis, gearbeitet werden.

Der Zugriff auf die Speicherverwaltung der Datenbasis geschieht über die Funktionen ov_database_alloc(), ov_database_realloc() und ov_database_free(), die sich wie ihre Pendants malloc(), realloc() und free() verhalten, jedoch Datenbasis-Speicher zur Verfügung stellen bzw. frei geben.

Beispiel:

```
#include "libov/ov_database.h"

...

void *ptr1, ptr2;

ptr1 = ov_database_alloc(8);
if(ptr1) {
  ...
  ptr2 = ov_database_realloc(ptr1, 15);
  if(ptr2) {
    ...
    free(ptr2);
  } else {
    free(ptr1);
  }
}
```

Das Allokieren von Speicher in der Datenbasis kann fehlschlagen, wenn kein genügend großer Block in der Datenbasis vorhanden ist. (Bei manchen Systemen, z.B. Linux, wird in diesem Fall versucht, die Datenbasis zu vergrößern, aber auch dies kann fehlschlagen.) In diesem Fall wird ein NULL-Zeiger zurückgegeben.

Wie bereits gesagt, können Referenzen auf selbstverwaltete Datenbasis-Bereiche nur als Adress-Offsets das erneute Mappen der Datenbasis überleben, z.B.:

```
#include "libov/ov_database.h"

...

OV_DLLFNCEXPORT void mylib_myclass_startup(OV_INSTPTR_ov_object pobj) {
  OV_INSTPTR_mylib_myclass pthis;
  OV_INT                   *pint;
  pthis = Ov_StaticPtrCast(OV_INSTPTR_mylib_myclass, pobj);
  /* allocate memory */
  pint = (OV_INT *)ov_database_alloc(8+sizeof(OV_INT));
  if(pint) {
    pthis->v_offset = (char *)pint-(char *)pdb; /* offset is an OV_INT */
  } else {
    pthis->v_offset = 0;
  }
  ...
}

void mylib_myclass_foo(OV_INSTPTR_mylib_myclass pthis) {
  OV_INT *pint;
  /* use memory */
  if(pthis->v_offset) {
    pint = (OV_INT *)(((char *)pdb)+pthis->v_offset);
  } else {
    pint = NULL;
  }
  ...
}

OV_DLLFNCEXPORT void mylib_myclass_shutdown(OV_INSTPTR_ov_object pobj) {
  OV_INSTPTR_mylib_myclass pthis;
  pthis = Ov_StaticPtrCast(OV_INSTPTR_mylib_myclass, pobj);
  /* free memory */
  if(pthis->v_offset) {
    free((((char *)pdb)+pthis->v_offset));
    pthis->v_offset = 0;
  }
  ...
}
```