Universität Bielefeld WS 2003/2004

Greifsteuerung für antropomorphe Roboterhände

Import von C-Funktionen und C++-Klassen in Neo/NST

Kerstin Bunte 27. November 2003

Inhaltsverzeichnis

1 Erste Schritte

Es ist sinnvoll sich als erstes ein Arbeitsverzeichnis anzulegen, in das die Quelltexte gelegt werden. Die Makefiles, einige Sourcen und Beispiele bekommt man durch Ausführung folgendes Befehls:

cvs -d/vol/ni/src/cvsroot co mynst

Der Befehl liefert uns ein Verzeichnis namens *mynst*, in dem die sich die Makefiles, ein Ordner *mysrc* und ein Ordner *o.linux* befinden. Im *mysrc*-Verzeichnis liegt eine Datei Makefile.custom, die nun folgendermaßen angepasst werden kann.

 # Deklariere prog_unit loadlibs hier:

LOADLIBS = PUWimportCF

 # Objektdateien, die für PUWimportCF.so gebraucht werden.

PUWimportCF.so: PUWimportCF.o

 # Objektdateien, die neben nst/neo_vects.o für vects.so gebraucht werden.

vects.so: vects.o

Nach diesen Eintragungen wird der Aufruf *make all* im *mynst*-Verzeichnis alle Ordner mit der Endung *src* durchsuchen und die Dateien PUWimportCF.so, vects.so, und diverse *.d in das *o.linux*-Verzeichnis legen.

BEACHTE: beim löschen einer im Makefile vermerkten Datei im *src*-Verzeichnis treten Probleme auf, wenn nicht auch die zugehörigen *.d Dateien aus dem *o.linux*-Verzeichnis gelöscht werden!

Nun muß nur noch der LD_LIBRARY_PATH in der .bashrc um das Arbeitsverzeichnis erweitert werden, damit Neo/NST unsere erstellten shared libraries auch finden kann.

! Bei Änderung mit erneuter Compilation der shared library muß Neo immer neu gestartet werden, damit die Änderung auch übernommen wird.

Im folgendem werden diese Synonyme verwendet:

xxx	steht für den Namen einer Funktion oder Klasse.
nnn	steht für den Datenbanknamen der shared library (ohne Pfad).

2 Neo

Neo ist eine interaktive Umgebung für die graphische Konstruktion und Ausführung von Datenflußdiagrammen (Circuits) bestehend aus NST-Units. Diese Units erscheinen als „Icons“ die mithilfe der Buttons in der obersten Zeile bearbeitet werden können. Circuits können zusammen mit ihren Zuständen gespeichert und geladen werden. Subcircuits können zum Wiedergebrauch in den „circuits folders“ gesammelt werden.

3 Der prog_unit-Wrapper

Der einfachste Weg um neue Objekte zu NST hinzuzufügen ist über die prog_unit. Sie kann C-Funktionen und C^{++}-Klassen aus shared libraries importieren. Hierfür müssen wir ein wenig Wrappercode in den Quelltext oder in eine externe Datei übernehmen.

3.1 Importieren von C-Funktionen

Um eine Funktion für die prog_unit verfügbar zu machen erfordert es zwei Schritte:

1. Damit die prog_unit eine Schnittstelle für den Funktionsaufruf hat wird nach den *#include*-Anweisungen folgende Zeile eingefügt.
   *char *FUNCTIONS_nnn[] ={„scalar_prod(float*,float*)“, NULL};*
   Der Name der shared library ist hier nur optional, aber es wird empfohlen dies so zu machen.

2. Anschließend erfolgt die Implementation, bzw. das Wrapping der Funktion selbst, z.B. *PUWimportCF.c*.
   *float scalar_prod(int *piDim, void **ppvArg) { ... }*

Diese Datei muß nun nur noch mit dem Befehl
    gcc -shared -o PUWimportCF.so PUWimportCF.c (im *mysrc*-Verzeichnis)
oder *make all* (im *mynst*-Verzeichnis)
zu einer shared library kompiliert werden. Nun starten wir Neo und ziehen eine prog_unit in das circuit-Fenster. Mit dem Befehl #import „nnn“ steht uns nun die Funktion scalar_prod zur Verfügung.

3.2 Importieren von C^{++}-Klassen

Es ist möglich Klassenobjekte für die prog_unit zu definieren. Die Elementfunktionen und Variablen jeder Klasse sind in einem *char**-Array ähnlich wie in der *FUNCTIONS-Deklaration oben unterzubringen.
Die Arraynamen (einer für jede definierte Klasse) werden dann in einem weiteren char **CLASSES_nnn -Array gesammelt.
Zum Beispiel wird eine einzige Klasse foo in einer library definiert:

*char *foo_c = {*

```
        „c_:“,          // Präfix wegen Namenskollisionen
        „foo(int)“,     // Klassen-Konstruktor
        „~foo“,         // Klassen-Destruktor
        „float fX“,     // eine float-Variable
        „int iX“,       // eine int-Variable
        „readonly int iY“,
        „float *pfFloat“,       // float-Array
        „void change(int)“,     // exportierte Methode
        NULL    };

char **CLASSES_nnn[] = {foo_c, NULL};
```

Nachdem ein *#import „nnn“* in der prog_unit erstellt wurde, können Instanzen der Klasse deklariert werden mit:

class foo a(5), b(10);

Anwendungen könnten nun folgendermaßen aussehen:
a.iX = 15;
b.fX = a.fX + 1;
for(i = 0; i < dimof(a.pfFloat); i++) a.pfFloat[i] = i;
b.change(3);

! Readonly definierte Variablen können nur über Funktionen der gleichen Klasse geändert werden, jedoch nicht direkt in der Prog-Unit.

! Eine library kann gleichzeitig Definitionen von Klassen (*CLASSES_nnn[]*) und Funktionen (*FUNCTIONS_nnn[]*) besitzen.

(Obgleich das *CLASSES_nnn[]*-Array Definitionen von mehreren Klassen innerhalb einer einzigen library nnn spezifizieren kann, wird es empfolen dieses immer nur für eine einzige Klasse pro library zu tun. Und für die library den gleichen Namen zu wählen wie für die Klasse).
Dies wird der prog_unit erlauben die erforderte library automatisch zu laden, wenn eine Klassendeklaration ohne ein vorangegangenes explizites *#import*-Statement erfolgt. (Zu beachten ist, dass das automatische Laden von Klassen immer die erste passende library des LD_LIBRARY_PATH heraussucht).

4 Der NST/Neo-Wrapper

Mithilfe der Quellcode-Generatortools *mku+* und *mku++* können C-Funktionen und C^{++}-Klassen in NST eingebunden werden.

- C: *mku+* generiert den notwendigen Interfacecode um eine Kollektion von kooperierenden C-Routinen (einer C-Klasse) in NST-Programmen oder interactiv in Neo als NST-Unit zugänglich zu machen.
- C^{++}: *mku++* stellt einen ähnlichen Service für eine echte C^{++}-Klasse dar.

Beide Tools sind Erweiterungen des älteren mku-Tools und erlauben eine saubere Trennung zwischen den NST/Neo-Niveau und dem Niveau der interfaced Objekts (bei denen es reicht in kompilierter Form vorhanden zu sein).

4.1 NST

NST bietet mit einer kompletten Bibliothek (library) von mehrfachverwendbaren software-Komponenten die „Units“ genannt werden den Rahmen zur Konstruktion von künstlichen neuronalen Systemen. Jede NST-Unit implementiert eine bestimmte Funktion in Form eines Objektes, das ausgeführt werden kann (und eine geringe Zahl von zusätzlichen Operationen besitzt, wie Initialisierung, Adaption, laden und speichern des gegenwärtigen Zustands).
Die meisten Units besitzen Inputs und Outputs, mit denen Sie mit anderen Units verbunden werden können. So können „circuits“ erstellt werden, die die Funktionen von mehreren Units miteinander verbinden. Ein Circuit kann „eingekapselt“ und somit als neue Unit verfügbar gemacht werden.
Neo ist eine grafische Programmieroberfläche für NST und kann die Circuit-Diagramme auch ohne Kompilation ausführen, was die Erprobung wesentlich vereinfacht.
Für das Erstellen von Units, stellt NST eine große Sammlung von Konstruktorroutinen zur Verfügung, wobei jede Konstruktorroutine Instanzen eines bestimmten Unit-Typen erstellen kann. Die meisten Konstruktorroutinen erwarten Parameter, die die Eigenschaften der erstellten Unit weiter definieren. Rückgabewert ist immer vom Typ *.unitptr* der als weitere Referenz der erstellten Unit dient. Dieser zeigt auf eine Datenstruktur Unittype die alle wichtigen Informationen über die Unit enthält (wie viele Inputs und Outputs sie hat, Zeiger auf ihre Werte und andere private Variablen). Dies schließt 6

Funktionszeiger zu C-Subroutinen ein, die die 6 Methoden execution, adaption, loading, saving, control und removal implementieren, die jeder Typ Unit enthält. Diese 6 C-Subroutinen zusammen repräsentieren die Funktionalität der NST-Unit. Wer seine eigene NST-Unit schreiben will muß diese 6 Subroutinen zusammen mit einer Konstruktorfunktion, die ein Unittype-struct erstellt, implementieren.

4.1.1 Die NST-Datentypen

Die Datenitems eines Pins können eines der Sechs grundlegenden Typen *NST_SCALAR, NST_FLOAT, NST_INT, NST_BYTE, NST_STRING* oder *NST_SPECIAL* enthalten. Dies entspricht den Typen in C.

NST_SCALAR	ein einziger float-Wert
NST_FLOAT	ein Array von float-Werten („float-Vektor“)
NST_INT	ein Array von int-Werten („int-Vektor“)
NST_BYTE	ein Array von unsigned char-Werten („byte-Vektor“)
NST_STRING	eine variable length von null-terminated character string („string Typ“)
NST_SPECIAL	für zukünftige Erweiterungen, um willkürliche Datentypen zu handeln

Die Anzahl der Datenelemente in dem Datenitem eines Pins, Pin-dimension genannt, wird von der Funktion *nst_dim_of_p(p)* zurückgegeben. Ebenso, gibt *nst_size_of_p(p)* die Bytegröße eines Datenarrays aus Pin p zurück (für einen Pin vom Typ *NST_STRING*, ist das die Stringlänge plus eins).

4.1.2 Inputs und Outputs von NST-Units zugänglich machen

Um auf die Werte an den Pins zuzugreifen, stellt NST Macros zur Verfügung. Jeder Macroname besteht aus einem Prefix um den Type des Pins der zugänglich gemacht werden soll anzuzeigen. *NstInp(u,f,p)* oder *NstOut(u,f,p)* wird benutzt um den p-ten Pin des f-ten Input- oder Output-Feldes der Unit u zugänglich zu machen.
Tabelle der Outputtype-Macros (die Inputtype-Macros sehen genauso aus, nur mit *Inp* statt *Out*).

NST Pintyp	Macroname	steht für
float vector	pfNstOut(u,f,p)	ptr zum ersten Element des float-Vektor
int vector	piNstOut(u,f,p)	ptr zum ersten Element des (short) int-Vektor
byte vector	pcNstOut(u,f,p)	ptr zum ersten Element (unsigned) char-Vektor
string	ptNstOut(u,f,p)	ptr zum Start des var len string
any	pvNstOut(u,f,p)	void ptr zum ersten Datenelement, jeglichen Typs
scalar float	fNstOut(u,f,p)	Wert des scalar-float-Pin
pin	pNstOut(u,f,p)	ptr zum gesamten NST-Pin struct

Beachte, dass der vom Macro gegebene ptr dereferenziert sein muß um ein einziges Element in einem Vektor Pin1 zugänglich zu machen. Zum Beispiel ist
piNstInp(u,1,2)[3] das Element an der Position 3 (das 4. Element) des *int*-Vektor-Pins an Position 2 (der 3. Pin) im Feld 1 (das 2. Feld) der Unit *u*.

4.1.3 Das *unittype*-Struct

Das grundlegenste Objekt in NST wird Unit genannt. Anders als eine Procedure, ist eine Unit ein Objekt, zu dem einige Daten und einige Methoden gehören. Während C^{++} das Klassenkonzept für die direkte Implementation von solchen Objekten bietet, benutzt man für die *C*-Implementation ein *struct*-Konzept zusammen mit Funktionszeigern um dieses Objekt zu implementieren.
Eine NST-Unit wird als eine Instanz der Struktur vom *C*-Typ *unittype* erstellt. Diese Struktur ist in *nst_types.h* definiert und hat den folgenden grundlegenden Inhalt:

```
typedef struct unittype{
    /* Zeiger auf die Inputs und Outputs: */
    field **in; /* Zeiger zum Feldzeigerarray für das Inputinterface */
    field **out; /* Zeiger zum Feldzeigerarray für das Outputinterface */
    ...
    /* Funktionszeiger zu den klassenspezifischen Methoden: */
    int (*exec)(struct unittype*); /* main Methode */
    ... weitere Methoden für adaption, loading, saving, removal ...
    /* Zeiger zu klassenspezifischen privaten Variablen: */
    void *var;
    ...
    /* weitere zusätzliche Zeiger */
} unittype, *unitptr;
```

Die *unitptr*-Struktur stellt den Zugang zu allen Eigenschaften einer Unit dar. Eine Instanz einer Unit einer bestimmten Klasse zu erstellen, erfordert die Schaffung einer Instanz der obigen Struktur und die Initialisation dessen Interface, Daten- und Funktionszeiger, die die spezifischen Eigenschaften der gewählten Klasse der Unit reflektieren. Diese Schritte müssen von der Konstruktorfunktion ausgeführt werden. Somit besteht die Hauptaufgabe bei Implementation einer Unit in der Programmierung dieser Konstruktorfunktion. (weiterzulesen in nstdoc.ps.gz, Seite 3 Abschnitt 2)

4.1.4 Klassen als NST Units verfügbar machen

Einfache Klassen: Für einfache Klassen mit einer main-Operation und einigen zusätzlichen Kontrolloperationen sowie Methoden zum laden und speichern des gegenwärtigen Zustand von einem File oder in ein File ist der einfachste Weg eine Klasseninstanz als eine einzige NST-Unit mit den *exec, ctrl, load* und *save*-Funktionen darzustellen. Zusätzlich bietet NST eine spezielle *adapt*-Methode für Klassen die adaptive Objekte darstellen, welche eine zweite main-„Adaptions"-Operation besitzen. Die Mehrheit der existierenden NST Units folgen diesem Schema und das Tool *mku* kann das Schreiben des notwendigen Codes unterstützen.
Große Klassen: Um eine Klasse mit einer grösseren Ansammlung von Methoden in NST einzubinden wird empfohlen jede Klasseninstanz als benannte Container-Unit darzustellen, die für jede Klassenmethode eine separate Subunit enthält. Mit seiner *exec*-Funktion repräsentiert jede Klassen-Subunit eine Klassenmethode und deswegen kann ihr Input- und Output-Interface spezifisch zugeschnitten werden für den Gebrauch seiner Methode. Die Methode einer Klasseninstanz kann somit beim Ausführen der entsprechenden Klassen-Subunit aufgerufen werden. Dies geht aber nur, wenn beide, der Klassen-Container und die Klassen-Subunit benannt sind (in Neo, wird dies durch eine unterstrichene Beschriftung angezeigt). Ein Aufruf kann beispielsweise folgendermaßen aussehen: *NamedesContainers:NamederSubunit* etc.
Alle Klassen-Subunits eines Klassen-Containers haben gemeinsame *ctrl, load* und *save*-Funktionen die durch ihren Klassen-Container vereinigt werden und deswegen immer zu der vollständigen Klassen-Instanz gehören.
Um dieses sehr flexible Schema für eine vorhandene Klasse zu implementieren, brauchen wir eine geeignete Konstruktorfunktion die einen Klassen-Container mit den verlangten Klassen-Subunits instanziert und die mittels *exec*-Funktionen der Klass-Subunits die vorhandenen Klassenmethoden zuweist.

Die Tools *mku+* (für in *C* implementierte Klassen) und *mku++* (für in C^{++}

implementierte Klassen) generieren einen Großteil des Codes den man benötigt, um Instanzen von Klassen-Containern interaktiv in Neo zu erstellen und zu benutzen.

4.2 *C*-Klassen einbinden mit *mku+*

Als Beispiel betrachten wir die *C*-Klasse *vects* die Methoden zum speichern und manipulieren von *float*-Vektoren mit einer festen Dimension bereitstellt. Eine Instanz von *vects* kann null oder mehrere Vektoren haben und Methoden wie *add(), set(), get()* und *clear()* erlauben das Hinzufügen von weiteren Vektoren, das Austauschen oder Bekommen der Elemente eines gespeicherten Vektors und das Entfernen aller Vektoren. Eine weitere Methode *forall* wird zur Verfügung gestellt um etwas für alle Vektoren durchzuführen. Das Interface für eine Implementation könnnte durch folgendes Headerfile definiert werden.

```
typedef struct vec_ { /* dies ist ein zusätzliches struct */
        float *pfData;
        int iIndex;
        struct vec_ *pNext, *pPrev;
} vec_t;

typedef struct vects_ {
        /* Dies impementiert eine Instanz */
        int iDim;
        int iNum;
        vec_t *pFirst, *pLast;
        void (*add)(float*, struct vects_ *);
        int (*num) (struct vects_ *);
        float *(*get) (int, float*, struct vects_ *);
        float *(*set) (int, float*, struct vects_ *);
        void (*clear) (struct vects_ *);
        void *(*forall) (float*, int*, void*, struct vects_ *);
} vects;
extern vects *create_vects(int);
extern void remove_vects(vects*);
```

4.2.1 Auflistung 1: Definition einer *C*-Klasse

Um diese Klasse in NST benutzen zu können, generieren wir den notwendigen Wrapper-Code, indem wir *mku+* mit Argumenten, wie den gewünschten

Namen (z.B. *vc1*) für die Konstruktorfunktion des NST Klassencontainers, gefolgt von den Namen für die Klassen-Subunits, die die Methoden wrappen sollen, aufrufen. Für die 5 Elementfunktionen aus unserem Beispiel würden wir z.B. folgenden Aufruf benutzen:

mku+ vc1 vects add num get set clear

Nun finden wir die drei folgenden Dateien:

1. ***nst_vc1.c*** Eine Datei, die eine Code-Schablone für eine NST-Konstruktorfunktion *vc1(...)*, einen Klassencontainer mit Subunits (benannt *add, num, get, set* und *clear*) mit den dazugehörigen *exec*-Funktionen *exec_add .. exec_clear*, die die entsprechenden Klassenmethoden wiederspiegeln.

2. ***neo_vc1.c*** Eine Datei, die eine Code-Schablone zum wrappen der NST-Konstruktorfunktion *vc1()*, die in *nst_vc1.c* zum interaktiven Gebrauch in Neo definiert ist, enthält. Diese Schablone brauchen wir hauptsächlich, um das Dialog-Fenster zu spezifizieren, das die Parameter für den Aufruf *vc1()* und die Ausgabe desselben mit den Parametern aus diesem Fenster liefert.

3. ***vc1.mak*** Ein *makefile* zum kompilieren des generierten Paares *nst_vc1.c* und *neo_vc1.c* von NST/Neo Wrapper-Dateien in eine shared library *vc1.so*. Das *makefile* setzt die Existenz einer weiteren Datei *vects.o* vorraus, die die gewrappte Klasse *vects* implementiert (statt der Objektdatei können auch zwei Source-Dateien *vects.[c,h]* vorhanden sein, aus denen die Objektdatei erstellt werden kann).

Nun folgen einige generelle Anmerkungen für die notwendigen Schritte zum vervollständigen der generierten Wrapper-Dateien. (Illustration durch das Beispiel *nst_vc1.c*).

- **Import der orginal *C*-Klasse:** Für eine Klasse werden alle notwendigen Definition (der Klassenstructs und der Methoden) in einem Headerfile gesammelt, das den gleichen Namen der Klasse (hier z.B. *vects.h*) tragen muß, mithilfe dieses Files wird *mku+* die entsprechende *#include*-Directive im generierten *nst_nnn.c*-File eintragen.

- **Definition der Konstruktorfunktion für den NST-Klassen-Container:** Die meisten Definitionen der Konstruktorfunktkion sind bereits in den generierten Codeschablonen vorhanden. Der Name ist im ersten Befehlszeilenparameter gegeben (hier *vc1*). Normalerweise hängt die zu wrappende Klasseninstanz von einer Anzahl von Parametern ab und die Konstruktorfunktion muß sich diese über ihre Argumentparameter holen. In unserem Fall, brauchen wir nur die Dimension des Vektors, z.B. haben wir einen einzelnen int-Parameter (zusätzlich zum immer anwesenden *uHost*-Argument). Mit diesen Informationen wird eine entsprechende Instanz der zu wrappenden Klasse erstellt. Weiterhin gebrauchen wir ein *struct nst_vc1*, das die Adresse der Klasseninstanz in einer lokalen Variable *pClass* für die erstellte NST-Unit hält, um die Instanz für *uNew* und alle Subunits zugänglich zu machen. Unser Beispielstruct wird dem *nst_var_of()* Zeiger der gewrappten Containerunit *uNew* zugewiesen.

```
typedef struct nst_vc1{
    /* Zeiger zur Instanz der benuzten C Klasseninstanz */
    vects *pClass; } nst_vc1_t;

unitptr vc1(int iDim, unitptr uHost) {
    unitptr uNew; /* handhabt die Unit zum zurückgeben */
    nst_vc1_t *pVar; /* handhabt die lokalen Variablen der Unit*/
    ...
    /* erstellt die Containerunit: */
    if ((uNew=primitive_unit(0,0,uHost))==NULL) {
        ... Diagnose ...
    }
    /* erstellt die vects-Instanze der zu wrappenden Klasse und weist
    den Adressenhandler für uNew's pVar -> pClass Zeiger zu: */
    nst_var_of(uNew) =
    pVar = CALLOC(1,nst_vc1_t); /* erstellt pVar-Instanz */
    pVar->pClass = create_vects(iDim);
    return_unit(uNew); /* zum Schützen der Subunits unten */
    nst_indicate_export_of_methods(uNew);
    ...
}
```

4.2.2 Auflistung 2: NST-Konstruktor zum wrappen einer *C*-Klasse

In vielen Fällen ist *pClass* das einzige Element, das in dem Struct *nst_vc1_t* gebraucht wird. Dann kann dieses Struct im Prinzip ausgelassen werden und überall kann *pClass* direkt anstelle von *pVar* benutzt werden. Jedoch muß man dann in der *remove*-Routine Obacht geben, wenn *pClass* freigegeben ist. So lange wie *pClass* mit dem *nst_var_of()* Zeiger von der zu löschenden

Unit übereinstimmt, wird NST die *free()*-Routine automatisch auf diesen Zeiger anwenden. Normalerweise wird dies mit der vorhergehenden Klassendestruktorroutine, die auf den gleichen Zeiger wirkt, kollidieren. Dieses Problem kann durch setzen von *nst_var_of()=NULL* befor NST den *free()*-Aufruf vollzieht, umgangen werden:

```
int remove_vc1(unitptr u) {
    vects *pClass = nst_var_of(u); /* pClass wird direkt benutzt */
    remove_vects(pClass); /* Anwendung des Klassendestruktor */
    /* nun ist pClass bereits freigegeben,
    aber NST wird es erneut versuchen! */
    nst_var_of(u) = NULL; /* nun kann NST's free() nicht
    mehr schaden */
    return nst_remove0(u);
}
```

Die restlichen zwei Statements schützen gegen direkte Ausführung der Klassencontainersubunits und setzten ein Flag das NST anzeigt, dass der Container eine spezielle Funktion ausführt, die mit den Subunits eine Sammlung von Klassenmethoden repräsentiert.

- **Konstruktoren von Klassensubunits des Klassencontainers:** Der generierte Code enthält bereits einen Konstruktor für den Aufruf jeder spezifischen Klassensubunit. Jedoch muß das Interface, das jede Klassensubunit braucht von dem Programmierer erstellt und die verwendbaren Parameter der *primitive_unit()*-Konstruktor-Aufrufe eingefügt werden. Anhand des Beispiels könnte das Klassenunit-Interface folgendermaßen aussehen.

```
{
    unitptr uSub_add = primitive_unit(NST_INP, NST_FLOAT, iDim,
            NST_OUT, NST_SCALAR, 1, NST_HOST, uNew);
    unitptr uSub_get = primitive_unit(NST_INP, NST_SALAR, 1, NST_OUT,
            NST_FLOAT, iDim, NST_OUT, NST_SCALAR, 1, NST_HOST,
            uNew);
    unitptr uSub_set = primitive_unit(NST_INP, NST_SCALAR, 1,
            NST_INP, NST_FLOAT, iDim, NST_OUT, NST_SCALAR,
            1, NST_HOST, uNew);
    unitptr uSub_clear = primitive_unit(0,0,uNew);
    /* benennt Subunits und erbt uNew's Datenzeiger */
    name_unit(„add“, uSub_add);
    nst_var_of(uSub_add) = nst_var_of(uNew);
    uSub_add->exec = exec_add;
```

```
    ...
}
```

4.2.3 Auflistung 3: NST Konstruktor weiterführend

Der meiste Code ist bereits an seinem Platz, nur die Argumentliste des *primitive_unit*-Aufrufs muß noch vervollständigt werden. Die *nst_var_of()* Statements erlauben jeder Subunit zum Teil Zugang zu der gewrappten Klasseninstanz, über ihren Wirt *uNew*.

***Exec*-Funktionen vervollständigen:** Als letzten Schritt muß jede Klassensubunit

1. mit ihren gegenwärtigen Inputdaten die verwendbaren Aufrufsparameter erstellen für ihre dazugehörige Klassenmethode,
2. diese Klassenmethode aufrufen, und
3. die Ergebniswerte zurück in die Outputs seines Interface schreiben.

Die Schritte (1-3) müssen für jeden Aufruf der *exec*-Funktion der Klassensubunit durchgeführt werden. Die Schablone für diese *exec*-Funktion ist bereits von *mku+* erstellt worden, aber die Implementation der Schritte (1-3) müssen noch eingefügt werden.

Zum Beispiel sollte die Subunit *uSub_get* die Methode *get* implementieren, die einen gespeicherten Vektor über einen Integerindex holt und die Startadresse der Vektordaten zurückgibt, wenn der Vektor gefunden werden kann. Andernfalls sollte *NULL* zurückgegeben, wenn der Index nicht erlaubt ist.

Um dies zu implementieren, hat die Subunit *uSub_get* einen einzigen Scalar-Input-Pin um den Index zu spezifizieren, und einen *iDim*-dimensioniertes Output-Feld, um die Vektordaten und einen einzelnen weiteren Scalar-Pin zurückzugeben, der Erfolg oder Mißerfolg der Operation anzeigt. Eine *exec*-Funktion, die die 3 Schritte implementiert könnte folgendermaßen aussehen.

```
int exec_get(unitptr u){ float *pfData;
    nst_vc1_t *pVar= nst_var_of(u); /* Lokale Variablen bekommen */
    vects *pClass = pVar->pClass; /* nun wird pClass=Class behandelt */
    /* Klassenmethoden mit Parametern aus dem Input einbeziehen */
    if((pfData = (*pClass->get)((int) fNstInp(u,0,0),pClass))) {
        /* kopiert die Ergebnisdaten in die Outputs: */
        memcpy(pfNstOut(u,0,0),pfData,pClass->iDim*sizeof(float));
        fNstOut(u,1,0)      = 1; /* Zeigt den Erfolg an */
    } else fNstOut(u,1,0) = 0; /* zeigt den Fehlschlag an */
    return OK;
```

```
}
```
Ähnlicher Code muß auch für die anderen *exec*-Funktionen eingefügt werden, um die Implementation zu vervollständigen.

4.2.4 Auflistung 4: Eine Methode wrappen

Weitere Methoden: Die *ctrl, load* und *save* Methoden werden analog ergänzt. Der einzige Unterschied besteht darin, dass dies, mit klarer Semantik für *load* und *save* für die gesamte Klasseninstanz zutrifft. Die *remove* Methode muß mit dem Aufruf der Klassendestruktor-Funktion vervollständigt werden, danach wird die NST-Default-Konstruktor-Funktion aufgerufen:

```
int remove_vc1(unitptr u){
    nst_vc1_t *pVar = nst_var_of(u); /* Lokale Variablen bekommen */
    remove_vects(pVar->pClass); /* « Diese Zeile wurde eingefügt */
   /* ! Entferne nie pVar direkt-das wird von nst_remove0 gemacht! */
    return nst_remove0(u);
}
```

4.2.5 Auflistung 5: Implementation der *remove*-Funktion

Eine Klasse als einen Klassencontainer zu wrappen hat den Vorteil, das später weitere Klassensubunits hinzugefügt werden können, ohne die Brauchbarkeit des veränderten Klassencontainers für vorher generierte .*.NST Dateien unter Neo zu beeinträchtigen.

Werden *adapt* Methoden gebraucht, werden sie völlig analog wie die *exec* Methoden behandelt. Zum Beispiel werden analoge Definitionen *adapt_get* etc. gesetzt, entsprechend der Schritte (1-3) der Code eingefügt und die entsprechenden *adapt*-Funktionszeiger der NST Klassensubunit gesetzt.

4.3 C^{++}-Klassen mit *mku++* wrappen

nst_vects.C ist die C^{++}-Sourcedatei die das obige Beispiel in C^{++}, unter Gebrauch der Headerdatei *vects.H* und des Quelltextes *vects.C* implementiert. Ein Blick auf den generierten Sourcecode kann folgende Punkte verständlicher machen:

1. Die Headerdatei enthält wiedereinmal die Definition der C^{++}-Klasse.

2. Die erzeugte Sourcedatei definiert eine begleitende Klasse die den gleichen Namen trägt wie die Orginalklasse mit angehängtem *_aux*. Die abgeleitete zusätzliche Klasse braucht zusätzliche Elementfunktionen,

die Interfacefunktionen für NST (z.B., eine *exec*-Funktion für jede Methodensubunit und die globalen *ctrl, load* und *save* Funktionen). Die Schablonen für diese Funktionen werden zwar zur Verfügung gestellt, müssen aber wieder vervollständigt werden. Zusätzlich müssen wir einen Klassenkonstruktor definieren, der zum Konstruktor der Basisklassenmethoden passt.

3. Ähnlich wie in vorherigen Beispiel, müssen wir einen Konstruktoraufruf einfügen, um eine Instanz der (abgeleiteten) C^{++}-Klasse in der Konstruktorfunktion für den NST Klassencontainer zu erstellen. Außerdem brauchen wir seine Adresse um den *var* Zeiger des Klassencontainers, sowie den *var* Zeiger aller Klassensubunits zu setzten.

4. Sobald die *exec* Methoden innerhalb der abgeleiteten zusätzlichen Klasse definiert sind, werden die notwendigen Interfacefunktionen „draußen" sehr einfach und vollständig von *mku++* generiert.

5. Es sieht vielleicht einfacher aus, den Umweg über die „außen"-liegenden Interfacefunktionen zu vermeiden und stattdessen direkt die *exec* Funktionszeiger der NST-Subunits an die Adressen der Elementfunktionen der abgeleiteten zusätzlichen Klasse zu setzen. Doch das geht nicht. Der Gebrauch der Funktionszeiger in $.C^{++}$ Klassenmethoden ist gegenüber den gewohnten aus $.C$ eingeschränkt: der Kompiler ist nicht in der Lage, die richtige Korrespondenz zwischen den Funktionen und ihren Klasseninstanzdaten unterzubringen.

6. Iteratormethoden werden auf gleiche Weise implementiert, aber die angenommene Konvention für den Iteratorzeiger (rückt *vor* der Iteration vor) unterscheidet sich vom Standardgebrauch von C^{++}-Iteratoren (welche *nach* jeder Iteration vorrücken).

Wie schon in obigen Beispiel brauchen wir eine Headerdatei und die Objektdatei, die die gewrappte C^{++}-Klasse implementiert.

4.3.1 Form der *nst_nnn.C*-Datei

Wir schauen uns im Folgendem mal die grundsätzliche Struktur der *nst_xxx.C* Datei an. Die roten, fettgedrukten Punkte zeigen an, dass hier der klassenspezifische Code eingefügt werden muß, die blaue Schrift gibt Hilfen für das Füllen.

... / #include-Direktiven, die mit Hilfe der xxx.H-Datei erzeugt wurden. */*
class vects_aux : vects{ / begleitende Klasse */*
    public:

```
    vects_aux( ... ) : vects( ... ){ };
    /* Es folgen die Interfacefunktionen für NST, also exec-Funktionen
       für jede Methode und die globalen ctrl, load und save-Funktionen */
    int exec_add(unitptr u) {... return OK;};
-* int exec_add(unitptr u){ add(pfNstInp(u,0,0)); fNstOut(u,0,0)=iNum;
                               return OK; }; *-
        ... get ...
    int ctrl(int mode, unitptr u){(void) mode; (void) u; ... return OK;};
    int load(FILE *fp, unitptr u){(void) mode; (void) u; ... return OK;};
    int save ...
}
/* Nun beginnt die NST-Konstruktorfunktion, wobei alle exec_METHODE(unitptr)
   und adapt_METHODE(unitptr), globale ctrl, load, save und remove-
   Funktionen bereits von mku++ generiert wurden, wir fahren also fort
   mit dem unitptr: */
extern „C“ {
    unitptr vc1( ... , unitptr uHost){
    -* unitptr vc1(int iDim, unitptr uHost) für dieses Beispiel *-
       unitptr uNew;
       if((uNew=primitive_unit(NST_HOST,uHost))==NULL) ... „Error“
            ... return NULL;}
       ...
       nst_var_of(uNew) = new vects_aux( ... ); /* Instanz erstellen */
       ...
       /* Es folgen die Subunits für jede Klassenmethode */;
       { unitptr uSub_add = primitive_unit( ... , NST_HOST, uNew);
       -* unitptr uSub_add = primitive_unit(NST_INP, NST_FLOAT, iDim,
                                            NST_OUT, NST_SCALAR, 1,
                                            NST_HOST, uNew)
       Parameter: ein Input vom Typ Floatvektor, der Dimension iDim,
                  ein Output des Typs Scalar der Dimension 1 und
                  schließlich der Container uNew. *-
                        ...
         nst_name_unit(„add“, uSub_add );
         nst_var_of(uSub_add) = nst_var_of(uNew);
         uSub_add -> exec = exec_add;
                        ...
       }
       /* Nun werden die ctrl, load und save-Methoden noch gesetzt (das
          hat mku++ schon erledigt) und anschließend die begleitende Klasse
          mit dem return-Befehl beendet */
```

```
        ...
        label_unit(uNew, „vc1“);
        return uNew;
    }
    /* Als letztes kommt das Interface zu den Elementfunktionen, dass von
        mku++ generiert wird. */
    ...
}
```

4.3.2 Form der *neo_nnn.c*-Datei

```
... /* externe Deklarationen für neue Units */
extern unitptr vects( ... unitptr);
extern NstIconDef_t ICONDEF_vects;
/* Hier wird deklariert, welche Icons der Folder (in welcher Reihenfolge)
    exportiert: */
BEGINFOLDER(vects)
    ICONENTRY(vects)
ENDFOLDER;
/* Nun kommt der Wrapper für die Unit vects */
ICONFORM1(vects) /* formt Dialogfenster1 Inputwindowsyntax */
    „unit vets - %n“
-* weitere Angaben, z.B. „dimension: %2i %n“ *-
ICONLABEL(„vects“) /* Label für die Unit */
ICONMANPAGE(„vects“) /* Name der Manpage */
ICONMAKE(vects){
    unitptr uNew;
    SRCLINE(„$ = #(AT); \n“); /* !Kriptisch! */
    /* Aufruf der Konstruktorfunktion, der Dimensionsparameter ist im IVAL(0).
        Alle Inputfelder des Inputwindow sind numeriert und können mit (int)IVAL(i),
        (float)FVAL(i) oder (char*)TVAL(i) abgerufen werden. */
    uNew = vcl(IVAL(0),HOST);      uNew = vc1(HOST);
/* mku++ hat hier einen langen Kommentar eingefügt, der
    nichtkommentiert einen Hilfstext in neo ausgeben kann. */
z.B. ICONFORM_FOR_METHOD(clear)
    „clear – clear all vectors %n“
  END;
    ...
} ...
```

4.4 Klassencontainer in Neo

In Neo sind Klassencontainer durch ein rotes Plus markiert. Sie werden genauso behandelt und ihre Verwendung ist ebenso vertraut, wie die der „einfachen“ Klassenunits. Anders als „einfache Klassenicons“, können die erstellten Icons geöffnet werden, um die Methoden, die sie enthalten kontrollieren zu können. Jede Methode erscheint als „feste Subunit“, die individuell nicht bewegt, verdoppelt oder gelöscht werden kann. Man kann sie aber mit ihrem Namen über die NST *method* oder *use_named* Units beeinflussen. Klassenicons dieser Art werden durch ein rotes Plus in der linken oberen Ecke gekennzeichnet.
Die Klassenuntericons eines Klassencontainers können nicht unabhängig voneinander verändert werden, dies geht nur zusammen mit ihrem Klassencontainer. Es ist möglich für jeden Klassenuntericon ein unterschiedliches Neo-Dialogfenster zu definieren. Dafür gibt es das *ICONFORM_FOR_METHOD()*-Macro innerhalb der *ICONMAKE()*-Funktion.

...
ICONFORM_FOR_METHOD(nameofmethod_1)
  „ ... textstring für das method_1 Dialogfenster ... “
ENDFORM;
...
ICONFORM_FOR_METHOD(nameofmethod_k)
  „ ... textstring für das method_k Dialogfenster ... “
ENDFORM;
...

Hier ist jeder *nameofmethod* der Name einer der Klassensubicons (ohne Anführungszeichen), und alle oder auch nur einige der Klassensubicons können Dialogfensterdefinitionen haben. Wenn es nur einen einzigen Neowrapper für den top-level Klassencontainer gibt, muß jede Dialogfensteranweisung für die eingenisteten Klassensubunits innerhalb der *ICONMAKE*-Routine der top-level Klassenunit erstellt werden. Gekennzeichnet wird die korrekte eingenistete Klassensubunit über ihren zusammenhängenden Pfadnamen (z.b. *set:beta* für eine Klassensubunit „*beta*“ eingenistet Klassensubunit „*set*“, innerhalb des top-level Klassencontainers).
Um die Methoden in einem Klassencontainer aufzurufen, benutze die *method*-Unit. Wenn die referenzierte Methodeninstanz existiert, erbt die *method*-Unit das Interface der Methodenunit und die Ausführung wird von der referenzierten Methodeninstanz aufgerufen. Eine ähnliche Funktion wird von der *use_named*-Unit zur Verfügung gestellt, aber der Aufruf der *method*-Unit wird empfohlen, da er effizienter ist.
Zwei Unterschiede der Semantik der *method*- und der *use_named*-Unit:

1. Die Outputs der *method*-Unit stellen einen Zugang zu den Ausgabewerten der referenzierten Unit *by-reference* zur Verfügung (während die *use_named*-Unit dies *by-copy* tut).

2. Die *method*-Unit verläßt die referenzierten Unit ohne ihre Eingabewerte zu verändern (abermals unterscheidet sich dies von der *use_named*-Unit, die die referenzierte Unit mit neuen (den kopierten) Eingabewerten verläßt).

Mit diesem Schema können auch komplexe C- oder C^{++}-Objekte in kleine und unabhängig aufrufbare Teile zerlegt werden, mit genügend einfach und klaren Interfaces für den interaktiven Gebrauch. Es stellt auch eine bequeme Basis für die Implementation von Objekten dar, dessen Inhalt dynamisch wachsen oder schrumpfen kann. Mit exportfähigen Iteratoren stellen sie die nötige Flexibilität zur Verfügung um jedes Element eines dynamischen Klassenobjekts durch einen willkürlichen Neocircuit zu verarbeiten.

4.5 Source- und Makefiles

nst_vects.c	*NST-wrapper für das C-Beispiel*
neo_vects.c	*NEO-wrapper für das C-Beispiel*
vects.c	*Implementation der C-Klasse*
vects.h	*Headers der C-Klasse*
nst_vects.C	*NST-wrapper für das C^{++}-Beispiel*
neo_vects.c	*NEO-wrapper für das C^{++}-Beispiel*
vects.C	*Implementation der C^{++}-Klasse*
vects.H	*Headers der C^{++}-Klasse*

Durch den Aufruf *make all* werden alle Beispielsourcen kompiliert und die shared libraries in den *o.linux* Ordner gelegt. Für vects wird nur die C^{++}-Implementation zur Verfügung gestellt. Wenn die Namenskonventionen befolgt wurden und die C^{++}-Dateien mit der Extension $*.C$ im entsprechenden *xxxsrc*-Verzeichnis liegen, werden die *nst5*-Makefiles automatisch das richtige tun.
Um die gewrappte C- oder C^{++}-Klasse zu benutzen, rufen wir Neo mit folgender Zeile auf:

neo -i +vects.so

Wir finden den Ordner an lezter Position der Neo-Toolbar. Dies kann man aber ändern, indem man die Datei *.neofolders* editiert.

FOLDERGROUP MeinEigenerOrdner

wird einen offen Ordner erstellen, der beim Start an entsprechender Stelle eingefügt wird. Wenn man um den Ordnernamen eckige Klammern setzt wird es ein geschlossener Ordner, der durch anklicken geöffnet werden kann.