Anleitung C objektorientiert in Simulink

Dr. Hartmut Schorrig, Ing. Richard Schorrig

24. Februar 2017

Zusammenfassung

Die Programmierung in C ist in Embedded Applikationen eine feste Größe. Die Programme werden komplexer. Damit ist die Überschaubarkeit der Zeilencodes Der vorliegende Artikel zeigt die Möglichkeiten der Anwendung der Objektorientierten Denkweise in Simulink-Modellen. Es wird damit eine andere ergänzende Herangehensweise an die Modellierung gezeigt. Diese Denkweise kann zu übersichtlicheren Modellen führen, da Daten und damit ausgeführte Operationen zweckmäßig getrennt in verschiedenen Modulen oder Subsystemen angeordnet werden können. Sie werden über sogenannte Handle einfach und überschaubar verbunden. Wichtig ist, dass dies für den Regelungstechniker transparent nachvollziehbar und verständlich ist, um die notwendige Akzeptanz zu erzeugen. Für den objektorientiert denkenden Informatiker, der eher eine *Unified Modelling Language* (UML) gewohnt ist, bietet Simulink damit eine neue nutzbare Plattform. Letztlich kann eine Objektorientierung in Simulink auch zur Rechenzeiteffizienz des generierten Codes beitragen, da referenzierte Module von mehreren Instanzen genutzt werden können und Kopieraufwand im Maschinencode eingespart wird.

In diesem Artikel wird dieser Ansatz am Beispiel verfolgt. Wesentliche Aspekte sind die sachliche und gute Darstellbarkeit in Simulink. Die dazu passenden Codegenerierung ist notwendig und selbstverständlich.

1 Objektorientierte Denkweise und Simulink

Die Objektorientierte Programmierung (OOP) ist eine feste Größe in der Softwaretechnologie. Objektorientierung im Kern bedeutet, dass Daten gemeinsam mit ihren Operationen, Methoden oder Funktionen betrachtet werden. Nicht-Objektorientierung liegt vor, wenn Funktionen ohne direkte Zuordnung zu Daten definiert werden. Weitere Eigenschaften der Objektorientierten Programmierung wie Abstraktion, Vererbung, überladene Methoden und weiteres sind ergänzend und im Gesamtsystem zweckmäßig, aber kein grundsätzliches Kennzeichen.

Objektorientierung wird zuerst mit den bekannten objektorientierten Sprachen wie C++ und Java in Verbindung gebracht. C-Entwickler sind oft und traditionell der Meinung, dass C dafür nicht geeignet sei. Daher wird oft nicht objektorientiert gedacht, sondern Einzelfunktionen geschrieben, die mit globalen Daten arbeiten. Allerdings kann man in C genauso objektorientiert arbeiten. Wichtige Sprachmittel sind Strukturen `struct` als Datenstrukturierung und Referenzen ('*Pointer*') auf die Daten und Funktionen. Werden diese Referenzen bestimmten C-Funktionen fest zugeordnet, dann hat man bereits Objektorientierung. Nicht zuletzt lassen sich dynamisch überladene Methoden mit den Funktionszeigern in C realisieren.

In Simulink sieht diese Welt traditionell anders aus. Der Blick ist hier auf Daten und deren Verknüpfung gerichtet. Dies kann auch als Gegenentwurf zur funktionsorientierten C-Programmierung aufgefasst werden, die primär auf Anweisungszeilen blickt. Die Daten in Simulink sind unabhängig und nicht an bestimme Operationen gebunden, sondern lediglich frei verknüpft. Damit liegt a priori keine Objektorientierung vor.

Charakteristisch für Simulink ist statt dessen die datenflussorientierte Darstellung: Verbindungspfeile sind immer in Datenflussrichtung gezeichnet. Würde man die UML als Basis ansehen, so entspräche dies eher dem Aktivitätsdiagramm. Dieses Diagramm führt aber weit weg vom dem, was in einem Simulink-Modell ausgedrückt werden soll.

In der UML gibt es das *Class Diagram* und das *Object Diagram*. Beide Diagrammarten werden oft nicht streng unterschieden, da sie gemeinsam die Eigenschaften von Klassen darstellen. Ein Rechteckblock ist eine Klasse oder einer Instanz einer Klasse, und stellt dabei die Klasseneigenschaften dar. Beziehungen zwischen den Klassen sind deren Nutzungsbeziehungen. Wenn eine Klasse Daten einer anderen Klasse verarbeiten will (über Operationen/Methoden oder direkt), dann gibt es eine Assoziations- oder Aggregations-Beziehung zu der anderen Klasse in Pfeilrichtung zur genutzten Klasse. Die Pfeilrichtung ist bei einem *Class-* oder *Object-Diagram* meist entgegen der Datenflussrichtung.

In Simulink werden **mit den Funktionsblöcken (*FB*) Objekte dargestellt, wenn die FBs Daten enthalten**, die für ihre Funktion notwendig sind. FBs, die keine Daten enthalten, sind keine Objekte sondern stellen **Operationen mit den ihnen zugelieferten Daten** dar. Bündeld man nun zusamengehörige Daten in bestimmten FBs, und definiert andere FBs, die mit diesen Daten arbeiten, dann unterteilt man in einem Simulink-Modell zwei Gruppen von FBs, ***Object-FB***s und ***Operation-FB***s. Der **Typ eines Object-FB** (in Simulink-Denkweise also ein FB, der als Library-Element referenziert wird), **ist eine Klasse**. Die zugehörigen **Operation-FBs gehören zu dieser Klasse** und stellen deren Operationen oder auch *Methoden* dar. Hier soll der Begriff *Operation* verwendet werden, wie auch in einigen UML-Toos.

Damit bekommt man in das Simulink-Modell eine objektorientierte Denkweise hinein. Nach Standpunkt des Verfassers kann die Objektorientierung von der datenflussorientierten Darstellung in Simulink und von der getrennten Blockdarstellung der Operationen profitieren. Andererseits sind auch objektorierntiere Prinzipien mit Simulink-Mitteln möglich und gut darstellbar. Letzlich ist dies auch ein Thema der Implementierung auf C-Ebene mit der Codegenerierung aus Simulink.

Im ersten Absatz wurde die Abstraktion, Vererbung und überladene Methoden als zweitrangig dargestellt. Der Artikel zeigt, dass auch diese Kennzeichen der Objektorientierung in Simulink abbildbar sind.

2 Ein Beispielmodul

Die Objektorientierung in Simulink wird am Beispiel der Anwendung sogenannter *Orthogonaloszillatoren* gezeigt. Ein Orthogonaloszillator erzeugt aus einem in etwa sinusförmigen Eingangssignal (schwingende, drehende Bewegung) zwei um 90 grad versetzte Ausgangssignale. Der Ausgang stellt das Signal im mathematisch komplexen Raum dar. Mit entsprechenden Operationen sind Betrag und Winkel oder auch 'stehende' Kennwerte berechenbar. Mit der Transformation zu stehenden Werten (Park-Tranformation benannt nach Robert H. Park) kann eine schnelle sich wiederholende Bewegung langsam weiterverarbeitet werden. In der Praxis braucht man häufig viele solcher Orthogonaloszillatoren in einem Modell. Im Beispiel wurde daran gedacht, dass 30 Signale parallel verarbeitet werden, als Mess-Signale an verschiedenen Stellen der gleichen Drehbewegung. Dabei werden pro Signal möglicherweise 10 dieser Module eingesetzt, um Oberschwingungen bzw. zyklische Unruhen oder auch Unrundheiten zu erkennen. Das ergibt also 300 Orthogonaloszillator-Module, die im Beispiel bei einer Abtastzeit von 20..50 us ca. 1000 Werte pro Periode mit einer Frequenz von 1000 U/min verarbeiten. Das Beispiel zeigt, dass Rechenzeit-Überlegungen oft genauso im Mittelpunkt stehen wie die eigentliche Funktionalität.

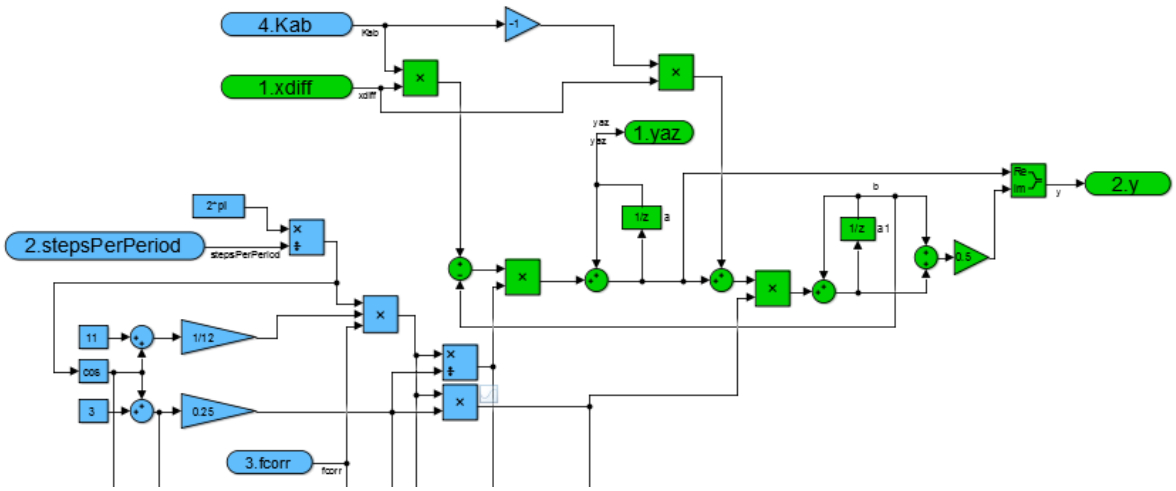

Abbildung 1: Orthogonaloszillator-Modul in Simulink

Das Bild zeigt eine Simulink- Realisierung eines Orthogonaloszillators. Farbig gestaltet sind die schnell zu berechnenden Anteile (grün) und die auch langsamer berechenbaren notwendige Adaption von Faktoren an die jeweilige Frequenz (blau). Der Orthogonaloszillator muss auf die Frequenz des Eingangssignals abgestimmt sein. Die Frequenz, etwa die Drehzahl einer Welle, wird anderweitig bestimmt, geglättet und meist in einer langsameren Abtastzeit verarbeitet. Da die Frequenzadaption trigonometrische Funktionen benötigt, ist deren Abarbeitung in einer langsamern Abtastzeit günstiger für die Rechenzeitbilanz.

Würde man dieses Modell nicht-Objektorientiert entwerfen, dann wären die grünen und blauen Modellelemente jeweils verschiedene Module, die in unterschiedlichen Abtastzeiten vielfach wiederholt im Gesamtmodell miteinander passend verbunden sein würden. Man wird sich dabei schon überlegen, dass die Parameterermittlung nur einmal für mehrere Module erfolgt und an die entsprechenden grünen Module mittels *From-Goto* oder anders geeignet verdrahtet werden. Im Abschnitt 4 wird der Ansatz der Objektorientierten Verdrahtung der Module gezeigt.

3 Objektorientierter Ansatz in C für das Modul

Für den Objektorientierten Ansatz in C (ohne Simulink) ergibt sich bei dieser relativ überschaubaren Aufgabenstellung folgende Herangehensweise:

Es wird für das Modul `OrthOsc2_FB` eine Datenstruktur definiert:

```
typedef struct OrthOsc2_FB_t {
union { int32 _obj_[6]; ObjectJc obj; };             //basicly data ObjectJc
union { int32 ipar[2]; Param_OrthOsc2_FB* par; };    //reference to all parameters
union { int32 _iangle_[2]; Angle_abwmf_FB* angle; }; //reference to angle and frequency
float_complex yab; //complex value of the oscillation
float_complex ypq; //complex value of the park-transformed static value of oscillation
float b;           //value of the b-Signals
float m, mr;       //magnitude and its reciprocal
} OrthOsc2_FB;
```

Da die Parameter für jeweils alle `OrthOsc2_FB` der selben Frequenz gleich sind, gibt es nur eine, gemeinsame Instanz `Param_OrthOsc2_FB` für die Parameter. Diese wird von allen `OrthOsc2_FB` referenziert. Dazu eine kurze Überlegung zur Rechenzeit: Ein referenzierter Zugriff auf Daten ist nicht langsamer als ein direkter Zugriff. Ein normaler moderner Prozessor hält die Referenz selbst in einem Prozessorregister und führt notwendige Adressrechnungen parallelläufig aus. Die andere Variante - alle Parameter dupliziert in der `OrthOsc2-FB`-Struktur direkt aufzunehmen ist damit nicht notwendig und erspart Kopieraufwand. Diese Fakten haben zu der Aussage "*Objektorientierung ist gegebenenfalls rechenzeitoptimaler*" in der Einleitung geführt. Gleiches trifft auch für die zweite Referenz `angle` zu. Diese enthält den drehenden Winkel passend zur Frequenz als komplexe Größe und als fortlaufender Winkelwert. Auch hier ist nur eine referenzierte Instanz für mehrere `OrthOsc2_FB` vorhanden. Der Ansatz der Referenzierung des Winkel-Moduls `Angle_abwfm_FB` ermöglicht die Nutzung eines vorhandenen universellen Moduls, das alle Werte passend enthält. Das ist *Wiederverwendung vorhandener Module*, entgegen dem häufigem Ansatz des copy&paste von Modellteilen in der Grafik.

Die *step*-Operation des `OrthOsc2_FB` in der schnellen Abtastzeit sieht dann wie folgt aus:

```
INLINE_Fwc
void step_OrthOsc2_FB(OrthOsc2_FB* thiz, float xAdiff, float xBdiff, float* yaz_y) {
  register float b;
  register Param_OrthOsc2_FB* par = thiz->par;
  thiz->yab.re += par->fIa * ( par->kA * xAdiff - thiz->b);
  *yaz_y = thiz->yab.re;
  b = thiz->b;
  thiz->b += par->fIb * ( par->kB * xBdiff + thiz->yab.re);
  thiz->yab.im = (thiz->b + b) /2;
}
```

Diese C-Funktion entspricht den grünen Teilen im Bild 2. Es ist in C optimiert. Das Schlüsselwort `register` aus dem klassischen C-Standard soll andeuten, dass der Compiler diese Werte in CPU-Registern hält. Die C-Funktion ist als `inline` definiert. Entweder man benutzt - wie oft vorhanden - einen C++-Compiler und definiert das `Inline_Fwc` als `inline`, oder für einen C-Compiler als

```
#define Inline_Fwc static
```

Für die Rechenzeiteffektivität ist es wichtig, dass ein Compiler Werte direkt in Registern behalten und unbenutzte Variable eliminieren kann. Das gelingt besser wenn kleine Code-Abschnitte direkt im Ablauf ("*inline*") eingebettet sind. Bei einer als `static` definierten C-Funktion ist das ebenfalls möglich. Es gilt das Optimierungspotenzial moderner Compiler zu nutzen.

Eine zweite Methode `calcpq_Orthi2_FB(...)` ist

hier nicht aufgeführt, es ist lediglich eine Vektormultiplikation mit dem gegebenem Winkel. Ebenfalls hier nicht aufgeführt aber in den genannten Internet-Downloadquellen zu finden, die Parameterberechnungen.

Für das Zusammenspiel des Moduls `OrthOsc2_FB` mit der Parameterberechnung kann man nun folgendes *Object Model Diagramm* aus der UML angeben:

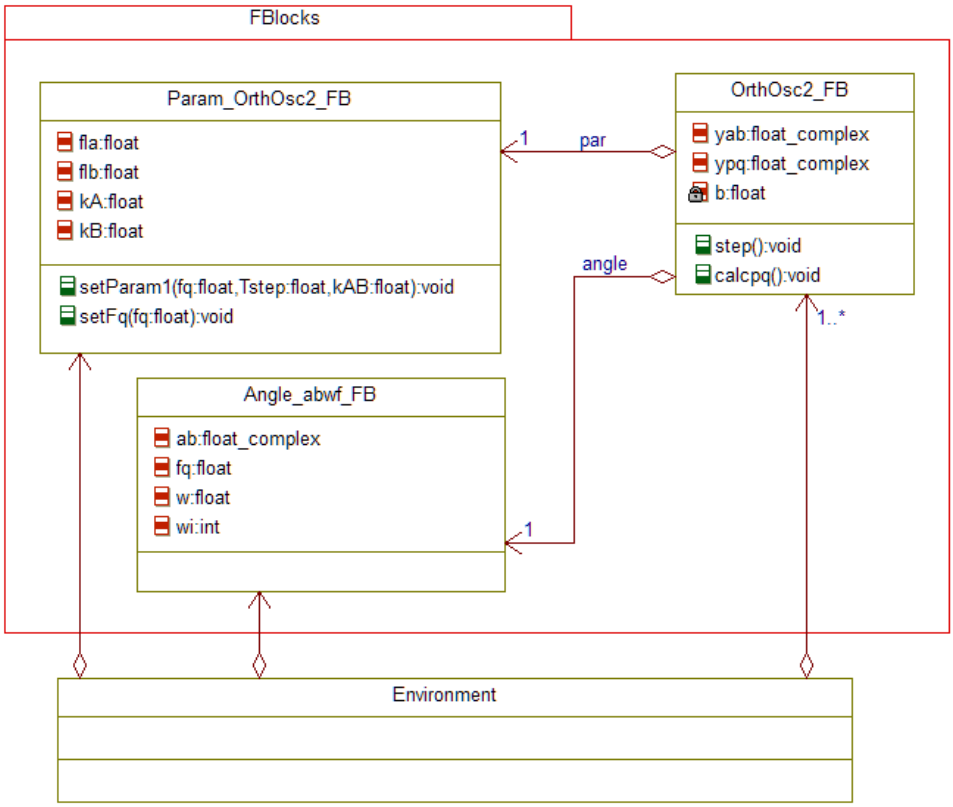

Abbildung 2: ObjectModelDiagram Orthogonaloszillator

Im Diagramm ist in der Aggregation aus dem Environment zu `OrthOsc2_FB` mit `1..*` gezeigt, dass es mehrere Instanzen `OrthOsc2_FB` aus der Umgebung heraus gibt. Je eine Instanz kennt eine Instanz von `Param_OrthOsc2_FB` und eine Instanz von `Angle_abwf_FB`. Es ist hier nicht sichtbar, dass es Gruppen von `OrtOsc2_FB` gibt die jeweils gemeinsam eine Instanz von `Param_OrthOsc2_FB` benutzen, aber die andere Gruppe benutzt eine andere Instanz von `Param_OrthOsc2_FB`.

Das hier benutzte *Object Model Diagram* aus dem UML-Tool *Rational Rhapsody (IBM)* stellt primär Klassenbeziehungen dar. Man kann auch mehrere Rechtecke der gleichen Klasse zeichnen, um anzudeuten, dass es verschiedene Instanzen der Klasse sind. In diesem Sinn müssten in diesem Diagramm mehrere `OrthOsc2_FB` dargestellt werden, die dann verschiedene `Param_OrthOsc2_FB` referenzieren. Dann wird im Diagramm auch ausgedrückt was gemeint ist. Das *Object Model Diagram* soll aber in erster Linie eine Hilfe sein, um Struktur in die Softwareteile bekommen, auch im Hinblick auf die Codegenerierung. Diesbezüglich ist der Blick auf die Klasse der wichtigere, da die Codegenerierung in UML-Tools nicht die gesamte Funktionalität automatisch generiert sondern bezüglich des *Object Model Diagrams* nur die Struktur einer Klasse mit ihren Attributen, Aggregationen und Operationen. Ein Object Model Diagram mit sogenanten *Parts* hilft zwar bei der Instanziierung der Objekte, die Parts stellen die konkreten Objekte einer Klasse dar. Eine gute Übersicht über die eigentliche Funktion ist damit aber nach Erfahrung des Verfassers auch nicht gegeben.

4 Verschaltung der Object-FBs und Operationen-FBs in Simulink

Simulink kann nun die im *ObjectModelDiagram* weniger gut darstellbare Zusammenarbeit der Objekte und Operationen besser präsentieren. Kerngedanke der Modelle in Simulink sind die konkreten Signalverschaltungen der Daten. In diesem Bild sind die im Abschnitt 1 vorgestellten **Object-FBs** und **Operation-FBs** gezeigt und miteinander verschaltet (*FB* = Funktionsblock).

Für den Übergang des im vorigen Kapitel gezeigten Ansatzes in C nach Simulink ist es zunächst notwendig, die C-Funktionen in Simulink-*S-Function-Module* umzusetzen. Die Abbildung von C-Code wird in Simulink gut unterstützt. Man kann die Bus-Definitionen und notwendigen *S-Function-Wrapper* per Hand schreiben. Vom Verfasser wurde aber ein Generator eingesetzt, der die Headerfiles in C analysiert und mit den so erhaltenen Daten über eine Textgenerierung die m-Files für Busdefinition und die C-Wrapper und tlc-Files für die S-Functions automatisch generiert. Für dieses Thema ist vom Verfasser ein weiterer Artikel in Arbeit, hier sei dies nur erst benannt.

In Simulink gibt es für dieses Beispiel das folgende Modell:

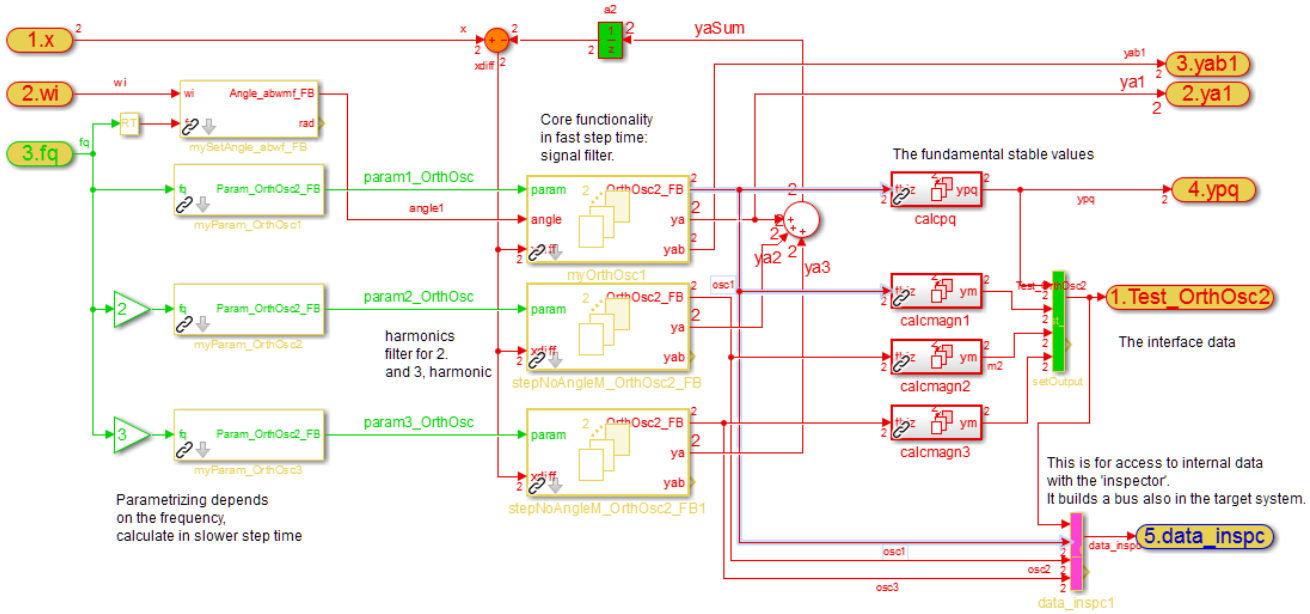

Abbildung 3: Simulink Orthogonaloszillator - Instanzmodell

Rechts im Bild sind drei Blöcke der Orthogonal-Oszillator-step-Operationen sichtbar. Jeder Block enthält genau die im vorigem Abschnitt gezeigte `step_OrthOsc2_FB(...)`-C-Funktion, die als *S-Function* in einem *for-each*-Subsystem enthalten ist. Das gesamte System ist vektoriell aufgebaut. Die Ausgänge der step-Operationen sind parallelgeschaltet und werden als Summe mit dem Eingang x verglichen. Die Differenz steuert die Orthogonaloszillator-step-Funktionalität aus. Die drei parallelgeschalteten Blöcke resonieren dabei jeweiligen mit einer Oberschwingungen. Auf diese Weise wird das Signal in Grundschwinung und harmonische Anteile zerlegt. In diesem Beispiel soll letztlich nur die so bereinigte Grundschwingung interessieren. Rechts oben im Bild wird dann aus der Grundschwingung (aus `thiz1_myOrthOsc`) die Park-transformierten stehenden Größen berechnet.

In der Mitte als `init..myOrthOsc` bezeichnet befinden sich die **Object-FBs** für die Orthogonalosziallatoren. Kennzeichnend für ein Object-FB ist: Ein Handle des Objektes ist Output. Es werden nicht drei Objekte angelegt wie auf den ersten Blick erkennbar sondern beliebig viele je nach Vektorgröße. Die `init...` -FBs müssen nur initial berechnet werden und sind daher einer `Tinit`-Rechenzeitscheibe zugeordnet.

Der rechte Bildteil soll in einer schnellen Abtastzeit laufen. Ist das Modell umfangreicher als in diesem Beispiel, dann kann sich dieser Modellteil etwa in einem eigenem referenziertem Subsystem befinden. Typisch ist: Diese FBs bekommen den Handle des Object-FB als Input.

Links im Bild sind die `Param_OrthOsc2_FB` angeordnet, die ebenfalls Object-FBs sind, sie haben einen Handle als Output. Gleiches gilt für `Angle_abwmf_FB`. Hier sind die Object-FBs nicht vektoriell da pro Harmonische nur ein Parametersatz für alle `OrthOsc2_FB` notwendig ist. Der Winkel sollte in der schnellen Abtastzeit verarbeitet werden - die Frequenz in einer langsameren. Diese Object-FBs enthalten im FB die einzige oder hauptsächliche Operation mit. Diese Object-FB benötigen keine zusätzliche Initialisierungs-Berechnung. Man spart durch Zusammenlegen von Objektbildung und Operation bei einfachen FBs Zeichenaufwand und erhält einen verbesserten Überblick.

4.1 Initialisierung, Aggregationen und Abtastzeiten

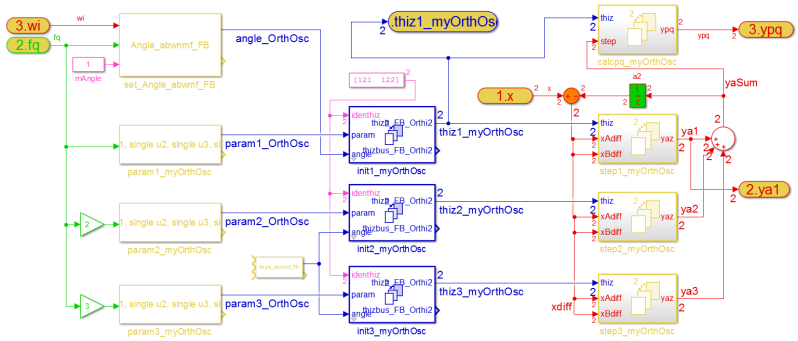

Abbildung 4: Simulink Orthogonaloszillator - Instanzmodell mit Abtastzeiten

Das obige Bild zeigt das selbe Modell wie Bild 3 nur mit colorierten Abtastzeiten. In blau ist deutlich erkennbar die `Tinit`-Abtastzeit. Die Handle-Ausgänge der linken FBs sind in der S-Function ebenfalls dieser Abtastzeit `Tinit` zugeordnet. Auch alle Handle-Eingänge sind `Tinit`. Die Handle-Werte selbst sind nach der Initialisierungsphase konstant.

In diesem Bild liefern die linken FBs Handle an die mittleren `init...`-FBs. Im Sinne der Objektorientierung oder UML sind das Aggregationen: Die `init...`-FBs kennen die FBs vom Typ `Param_OrthOsc2_FB` und `Angle_abwmf_FB`. Sie greifen auf deren Daten zu. Es kann sein, dass es gegenseitige Aggregationen zwischen Object-FBs gibt, oder auch indirekt über mehrere FBs hinweg im Kreis.

Eine Initialisierungsphase benötigt mehrere Durchläufe, damit die dort gerechneten `init_...` FBs Handle auf aggregierte Object-FBs speichern können. Wenn es Aggreationen im Kreis gibt, dann bekommt man eine Fehlermeldung: *Numerischen Schleife*. Diese kann und muss man in Simulink aufbrechen mit einem `1/z`-Verzögerungsblock in der `Tinit`-Abtastzeit. Die `Tinit`-Abtastzeit muss folglich mehrmals wegen der `1/z`-Verzögerungsblöcke durchlaufen werden, dann nicht wieder. Im generierten Code lässt sich das gestalten, indem die für diese Abtastzeit generierte `stepx...`-Funktion eben nur initial mehrfach aufgerufen wird, bevor das Gesamtsystem zyklisch gestartet wird.

Das Modul rechts oben berechnet in der schnellen Abtastzeit aus den Daten der Orthogonaloszillatoren die Park-Transformation mit den stehenden oder Gleich-Größen der Drehbewegung als Ergebnis. Die Verbindung der Ausgangssumme `yaSum` mit dem Eingang `step` dient eigentlich nur der Klärung der Rechenzeitreihenfolge. Nicht in der Codegenerierung aber im optimierenden Compiler wird diese Verbindung dann letzlich weggelassen, da `step` in der zugehörigen C-Routine nicht verarbeitet wird, die Routine `inline` ist und der Compiler die Situation damit erkennen kann. Die tatsächlich benötigten Daten werden dem Object-FB über das Handle entnommen. Die Rechenzeitreihenfolge ist aber deshalb wichtig, weil zuerst die Daten der drehenden Bewegung über das Input-x erneuert werden sollen, danach wird `ypq` mit diesen neuen Daten berechnet.

4.2 Datenflussrichtung

Für die Verbindung der Daten aus dem `step...` nach `calcpq...` ist der Datenfluss nicht mehr in Pfeilrichtung erkennbar. ***Dies ist ein Bruch mit der Simulink-Konvention, die besagt: Der Datenfluss geht nur in Pfeilrichtung***. Wie ist dies zu verstehen?

Die `init...`-Module liefern in Pfeilrichtung das Handle auf die Daten, die für die Verarbeitung verwendet werden. Das ist der Kerngedanke der Objektorientierung: Die Daten werden in Objekten (Instanzen) angelegt und in Operationen (*Methoden*) verwendet und geändert. Alle Module mit *Object-Handle-Input* sind die einzelnen *Operationen*. Also ist `step_myOrthOsc` der Aufruf einer Operation mit dem Namen des FBs `step_OrthOsc2_FB`, genauso wie `calcpq_myOrthOsc` den Aufruf der Operation `calcpq_OrthOsc2_FB` darstellt. Beide laufen in der schnellen Abtastzeit und arbeiten mit den Daten des Objektes, die sie per Handle kennen.

Das ist die eigentlich neue Konvention des Ansatzes der Objektorientierung in Simulink, die **zusätzlich zu der bekannten Konvention des Datenflusses in Pfeilrichtung hinzukommt**:

Es gibt Daten, die von Object-FBs geliefert werden. Die Bereitstellung der Daten wird in Pfeilrichtung dargestellt. Die nutzenden Operation-FBs oder Object-FBs arbeiten mit diesen bereitgestellten Daten, und können diese auch verändern. Es können mehrere FBs mit den selben Daten arbeiten. Nicht der Datenfluss selbst, sondern die Bereitstellung der Daten (über Handle) ist in Pfeil-Flussrichtung angegeben. Letzlich muss sich ein starr auf Datenfluss in Pfeilrichtung gewohnter Regelungstechniker etwas umstellen, wenn der Gedanke der Objektorientierung ihm bisher nicht bekannt war. Ein Informatiker und Regelungstechniker, der mit Objektorientierung umzugehen gewohnt ist, findet in diesem Schema eine erfolgversprechende Möglichkeit der Darstellung und Implementierung in Simulink.

4.3 Verbindung der Blöcke mit Handles, Speicher- und Typtest

Im generiertem Code stellen die Handles direkt die Speicheradressen der Daten dar. Zielsysteme sind meist nicht mehr als 32 bit breit, so dass das `uint32`-typisierte Handle ausreicht. Mit dem Handle als Input kennt also ein Block direkt die Daten.

Für die Simulink-Ebene gibt es ein kleines Problem: Meist läuft Simulink auf einem 64-Bit-System. Damit sind Speicheradressen nicht mehr mit 32 bit darstellbar. Simulink kennt bisher auch kein `uint64`. Nicht deshalb sondern wegen der Speicheroptimalität im Zielsystem soll das Handle nur 32 bit breit sein. Lösung dieses Problemes ist eine Tabelle, die als *shared memory* mit einer Länge angelegt wird, die größer ist als die Anzahl der Instanzen. Ein Wert von 1000 ... 10000 ist speichermäßig kein Problem (80 kByte). In diese Tabelle wird jede Instanzadresse eingetragen, nachdem sie angelegt wurde. Das Handle ist dann der Index auf diese Tabelle.

Signifikanz- und Typtest:

Bei der Verdrahtung der Handles im Modell kann es Verwechslungen geben. In einer vorigen Entwicklungsversion wurden statt der Handles Busse verwendet. Diese werden im Zielsystem als Pointer den Anwenderfunktionen (S-Functions) übergeben. Bei den Bussen prüft Simulink zur Compiletime, ob die Typisierung korrekt ist. Falsche Verbindungen fallen dann auf und müssen korrigiert werden, bevor die Abarbeitung beginnt.

Die Verwendung der Busse statt Handles wurde aus folgendem Grund aufgegeben: Der Vorteil ist gleichzeitig auch der Nachteil. Während der Entwicklung können auch bereits gezeichnete Modellteile aus verschiedenen Aspekten von der Abarbeitung herausgenommen werden. Man kann sie auskommentieren. Damit gibt es aber Fehlermeldungen, weil mindestens formell notwendige Bus-Inputs nicht mehr richtig versorgt werden. Man muss dort also dann auch noch nachbessern und umbauen, es entsteht Aufwand. In der C/C++/Java-Programmierung gibt es für solche Fälle eine einfache Lösung: Der `null`-Pointer ist kompatible zu allen Pointertypen. Bei Simulink geht das nicht. Man braucht einen Dummy-Bus-Creator oder ähnliches. Mit dem Handle ist es einfacher handhabbar. Wenn ein Handle abgeklemmt ist (durch Kommentierung), wird es automatisch mit einer 0 besetzt.

Das Problem des Typtests wird auf die Runtime im Simulink geschoben. Um den Typ einer Instanz zu erkennen, muss eine Typinformation in den Instanzdaten enthalten sein. Aus mehreren Gründen basieren die Instanzdaten der C-Implementierung auf einer `struct ObjectJc`, die allgemeingültige Daten enthält. Das Konzept ist Java entlehnt und nach C übertragen. In Java basiert jede Klasse auf `java.lang.Object`. Die hier interessierenden Daten sind die *Reflection-Informationen*, ebenfalls als Java-Konzept bekannt. Diese Reflection-Informationen werden auch dazu benutzt, um einen symbolischer Zugriff auf Daten in beliebigen Laufzeitumgebungen (insbesondere im Embedded-Zielsystem) zu realisieren.

Die entsprechenden Code-Zeilen im generierten S-Function-Wrapper für einen einfachen Typtest dieser Klasse lauten:

```
static void mdlOutputs(SimStruct *S, int_T tid)
{ .....
  uint32 thiz_Handle = *(uint32*)ssGetInputPortSignal(S, 0);        //input-handle
  if(thiz_Handle == 0) { return; }               //not initialized yet, only on startup.
  .....
  OrthOsc2_FB* thiz = (OrthOsc2_FB*) handle2Ptr->ptr[thiz_Handle]; //pointer from handle
  if( -1 == checkStrict_ObjectJc((ObjectJc*)thiz, sizeof(OrthOsc2_FB), 0, null, null)) {
    ssSetErrorStatus(S,"faulty ObjectJc-handle in step_OrthOsc2_FB: thiz");
    return;
  }
  char const* reflType = ((ObjectJc*)thiz)->reflectionClass->name;
  if( strcmp(reflType,"OrthOsc2_FB") !=0) {
    ssSetErrorStatus(S,"step_OrthOsc2_FB.thiz: faulty ref-type, expected: OrthOsc2_FB");
    return;
  }
}
```

Das Handle wird zuerst auf 0 getestet. In diesem Fall verhindert ein Handle=0 die Abarbeitung des Blocks. Das passiert wenn die `Tinit`-Abtastzeit noch nicht beendet ist. Danach wird für die S-Function das Handle als Index benutzt um den Pointer zu ermitteln. Das `void*`-Array `handle2Ptr` liegt in einem *shared-memory*-Bereich, damit es von allen S-Functions gleichermaßen gesehen wird.

Der Pointer sollte jetzt eine `struct` referenzieren, an dessen Anfang die `ObjectJc`-Daten steht. Die C-Funktion `checkStrict_ObjectJc` ist eine Operation der `struct ObjectJc` und prüft dessen Daten auf Signifikanz. Wenn diese Prüfung durchaufen ist, dann kann man mit casting auf den `reflectionClass->name` zugreifen und diesen textuell mit dem erwarteten Namen vergleichen. Mit der vorigen Prüfung wird ausgeschlossen, dass der Zeiger durch beliebige vorherige Fehler grundsätzlich falsch ist. Damit wird ein unkontrollierter 'Absturz' verhindert. Es gibt nur einen Abbruch mit einer klaren Simulink-Fehlermeldung.

Dieser Typ- und Sicherheitscheck wird aber nicht im Zielsystem und auch nicht im Accelerator-Mode im Simulink ausgeführt. Es ist davon auszugehen, dass das Simulink-Modell vor der Codegenerierung im Simulink mindestens einmal durchgelaufen ist, damit ist der Check der Verdrahtung erfolgt.

5 Black Box in C oder Objektorientierung auch mit Simulink-Modellteilen

Bisher wurde vorgestellt, dass die objektorientierten Module als *Object-* oder *Operation-FB* in C programmiert und in Simulink als *S-Function* bereitgestellt werden. Die Zusammenschaltung dieser Module erfolgt in Simulink.

Dahinter steht auch die Frage der Black-Box-Handhabung. Für den Regelungstechniker ohne C-Bezug wäre es zwar zumutbar, dass einfache C-Zeilen dieser Module verstanden, gelesen und auch korrigiert werden können. Der Aufruf des mex-Compilers ist auch nur ein Handgriff. Andererseits sollte eine funktionell klar beschriebene und getestete Black-Box akzeptabel sein. Eine Teamtrennung - C-Programmierer zuständig für die Kern-Module, Simulink-Designer für das Ganze - ist zweckmäßig, adäquat zur Aufteilung der Aspekte Systemverhalten, Regelung, Implementierung.

Es steht nun die Frage im Raum, ob auch die Bildung großer Module in Simulink objektorientiert vorteilhaft ist.

Ein anderer Anwendungsfall ist es, zu vorhandenen kleinen objektorientierten FBs in C noch zusätzliche Operationen hinzuzufügen. Diese sollen dann zunächst - oder überhaupt - im reinen Simulink entwickelt werden.

Um dieses zu realisieren, wird vom Verfasser wie folgt vorgegangen:

- Die Datenstrukturen für ein objektorientiertes Modul wird in einem C-Header aufgeschrieben. Man sollte dies nicht als C-Programmierung im komplexen aufwändigen Sinne verstehen. Ein Header-File ist überschaubar.
- Dort werden sogenannte Getter- und Setter-Methoden notiert. Möglich ist auch: Umwandlung der gesamten Daten in der `struct` in einen Simulink-Busausgang. Die Belegung der Daten in der `struct` kann aber nach zusammengehörigen Einzeldaten sortiert erfolgen. Man kann an dieser Stelle auch notwendige Mutex-Mechanismen für Multithread-Anwendungen unterbringen, die im Zielsystem eher in C beherrscht werden. Das erfordert meist den C- und System-Programmierer.
- Mit den so erstellten und als *S-Function* genutzten FBs können dann Daten in die objektorientierte Struktur geschrieben und von dort gelesen werden, bis hin zum Simulink-Buszugriff über den Bus-Selektor. Die Funktionalität ist dann Sache des Simulink-Modells.

6 Abstraktion, Vererbung, überladene Methoden

Der Regelungstechniker und Simulink-Entwickler wird mit diesen typisch objektorientierten Dingen zunächst nichts anzufangen wissen. Er wird aber selbstverständlich ein Modul an einen Handle hängen, von dem er weiß, dass das Modul zu dem Handle passt. Dabei kann es sein, dass der *Object-FB* in Simulink ein abgeleitetes Object bildet und dass der angehänge Verarbeitungsblock mit dessen Basisdaten arbeitet. Der Unterschied ist eigentlich nur der im Abschnitt 4.3 gezeigte Typtest. Der Typtest ist etwas komplexer als dort vorgestellt und sucht in den Reflection-Daten nach einer passenden sogenannten *Superclass*, also einer Basisklasse. In C ist die Bildung von abgeleiteten Klassen aus Basisklassen so geregelt, dass die Basisdaten am Anfang der `struct` der abgeleiteten Klassen stehen. Das Handle gilt also ohne Zusatzaufwand als Handle auf die Basisdaten. Die Erstellung solcher Konstrukte obliegt dann der C-Ebene. Die Verwendung ist transparent für den Simulink-Entwickler.

Überladene Operationen:

In der Objektorientierung gibt es sogenannte *overridden methods (operations)*. Häufiges Schulungsbeispiel Lohnabrechnung: Es wird für jeden Mitarbeiter aufgerufen: `personell.calculateSalary()`. Abhängig von der Art der Arbeitsvertragsgestaltung, also Lohnarbeiter, Angestellter usw. werden aber verschiedene Algorithmen aufgerufen. Welcher Algorithmus, wird von den Daten des Mitarbeiters bestimmt.

Auf Simulink übertragen bedeutet das: Es wird ein und derselbe Operation-FB an einen Object-FB per Handle verbunden und dann aufgerufen. Welche

Funktionalität dieser Block ausführt, wird aber vom Object-FB bestimmt. Dieser enthält und bestimmt die passende Operation, der Operation-FB sorgt nur für den Aufruf. Der Aufruf wird intern über einen *Function-Pointer in C* oder eine *virtual*-Methode in C++ realisiert.

Diese sogenannten überladenen Operatoren sind zwar nicht mehr mit Simulink-Mitteln zu modellieren, denn ein *Enabled Subsystem* für die verschiedenen Mitarbeitertypen, vergleichbar einem `if..else if` in der Programmierung entspricht eben gerade nicht dem Stil der Objektorientierung.

Mit Blick auf die Zukunft wäre es vorstellbar, dass es in Simulink ein *Overridden-Call-Modul* für überladene Operationen gibt. Die Operation selbst muss dann in der abgeleiteten Klasse, sprich im abgeleitetem Modellteil implementiert sein. Die per Handle aufgerufene Operation enthält dann also nicht die Implementierung, sondern nur die Organisation des Aufrufes der in den Instanzdaten enthaltenen Operation.

7 Zusammenfassung und Ausblick

Der vorliegende Artikel zeigte Möglichkeiten und Anwendung der Objektorientierten Denkweise in Simulink-Modellen. Es wurde damit eine andere ergänzende Herangehensweise an die Modellierung vorgestellt. Es wurde gezeigt, dass dies für den Regelungstechniker transparent nachvollziehbar und verständlich sein sollte, um die notwendige Akzeptanz zu erzeugen. Für den objektorientiert denkenden Informatiker bietet Simulink damit eine neue nutzbare Plattform.

Es wurden Details wie die Verbindung von sogenannten Function-FB und Operation-FB mit Handles, deren Datenflussrichtung und der notwendige Typtest gezeigt und diskutiert. Selbst abgleitete Objekte und überladene Operationen lassen sich mit Simulink-Mitteln darstellen.

Insgesamt dürfte diese ergänzende Objektorientierte Herangehensweise die Übersicht in den Modellierungen verbessern. Weiteres Potenzial kann es auch geben, wenn von Haus aus objektorientiert denkende Informatiker mehr zur Nutzung von Simulink übergehen.

8 Literatur, Querverweise

Die Simulink-Funktionen sind über die Simulink-Toolhilfe in einer lokalen Installation und im Internet auffindbar und sind daher nicht extra als Literaturquelle extra aufgeführt.

Ebenfalls als bekannt und verbreitetet oder auffindbar vorausgesetzt sind die Theorien und Praktiken der UML (Unified Modelling Language) und der OOP (ObjectOriented Programming).

Die konkrete Implementierung des hier gezeigten Beispieles getestet mit Mathworks, Simulink Version 2016a sind auf der Internetseite *www.vishia.org/smlk/Download* enthalten, Erläuterungen dazu auf *www.vishia.org/smlk/index.html*. Diese Seite wird vom Verfasser erstellt und gepflegt.