Modellgetriebene Entwicklung von OSEK Applikationen mit UML 2

Markus Schmidt
Technische Universität Darmstadt
FG Echtzeitsysteme, D-64283 Darmstadt
markus.schmidt@es.tu-darmstadt.de

Abstract: Eingebettete Systeme werden zunehmend komplexer und immer häufiger in sicherheitsgerichteten Systemen eingesetzt. Die Softwareentwicklung für diese Systeme ist traditionell textbasiert, wobei hauptsächlich hardwarenahe Sprachen wie Assembler oder C verwendet werden. Bedingt durch den geringen Abstraktionsgrad solcher Sprachen, werden zunehmend auch visuelle Modelle bei der Entwicklung eingebetteter Systeme eingesetzt. Im Vergleich zu anderen Domänen gibt es für die eingebetteten Systeme keine einheitliche visuelle Sprache, sondern jedes Entwicklungswerkzeug verwendet seine eigene Sprache. Im Bereich der objektorientierten Softwareentwicklung hat sich die UML als Standard etabliert. Wir werden UML benutzen, um eingebettete Systeme zu entwickeln, die das Betriebssystem OSEK-OS verwenden. OSEK-OS ist ein standardisiertes Betriebssystem, welches besonders im Automotive Bereich eingesetzt wird. Wir zeigen eine Abbildung von bestimmten UML Modellelementen auf die wichtigsten OSEK-OS Objekte. Weiterhin wird dargestellt, wie der gesamte Entwicklungsprozess durch die Benutzung eines UML kompatiblen Werkzeugs verbessert werden kann.

1 Einleitung

Die klassische Softwareentwicklung beschäftigt sich mit einer Menge von Dateien, die den Quelltext in textueller Form enthalten. Dieses Vorgehen ist bei kleineren bis mittleren Projekten akzeptabel, aber bei großen Projekten stößt dieses Vorgehen an seine Grenzen.

Ein aktueller Trend ist die modellgetriebene Softwareentwicklung. Visuelle Modelle ermöglichen eine höhere Abstraktion und bieten umfangreiche Notationen. Die modellgetriebene Softwareentwicklung wird besonders im Bereich der objektorientierten Programmierung eingesetzt, da es einfache Abbildungen von visuellen Modellelementen auf Elemente der Programmiersprache (z.B. `C++`) gibt.

Es gibt aber immer noch Softwareprojekte, die mit hardwarenahen Programmiersprachen realisiert werden müssen. Wegen der beschränkten Ressourcen müssen eingebettete Systeme sehr nah zur unterliegenden Hardware entwickelt werden. Die Betriebssysteme aus diesen Bereichen bieten zwar nicht den Komfort den man von modernen Mehrbenutzersystemen kennt, aber sie ermöglichen zumindest eine einfache Abstraktion von der unterliegenden Hardware.

OSEK-OS ist ein Betriebssystem, welches speziell für eingebettete Systeme entwickelt

wurde und hauptsächlich im Automotive Bereich eingesetzt wird. Es besteht aus einigen Kernel Objekten und einer C API zum Aufruf der Betriebssystemroutinen.

Diese Arbeit beschreibt einen Ansatz zur modellgetriebenen Entwicklung von OSEK Applikationen. OSEK Applikationen werden durch UML Modelle beschrieben, wobei die Bedeutung spezieller Modellelemente durch eine Abbildung auf OSEK Objekte definiert wird. Aus diesem Modellen wird durch Modell-zu-Text Transformation eine Menge von Textdateien erzeugt, die von den gängigen Werkzeugen zur Erzeugung der ausführbaren Applikation verwendet werden.

Bevor der Ansatz im Detail beschrieben wird, erfolgt im nächsten Abschnitt eine Zusammenfassung verwandter Arbeiten. Die Grundlagen von OSEK werden in Abschnitt 3 beschrieben. Anschließend werden die wichtigsten Grundlagen von UML dargestellt. Die Abbildung von UML Modellelementen auf OSEK Objekte wird in Abschnitt 5 definiert. Abschnitt 6 faßt diese Arbeit zusammen und bietet einen Ausblick auf weiterführende Aktivitäten.

2 Verwandte Arbeiten

Es gibt nur sehr wenige Arbeiten die sich mit OSEK und modellgetriebener Entwicklung beschäftigen. Nach unserem Kenntnisstand gibt es nur zwei Arbeiten die eine Abbildung von UML auf OSEK beschreiben. Beide Arbeiten verwenden das UML RT-Profil. Dieses Profil ermöglicht die Annotation von UML Modellen mit Informationen zum Scheduling, Performanz und Zeit.

Alan Moore [Moo02] hat als erster eine Erweiterung des RT-Profils beschrieben. Die grundlegenden OSEK Objekte werden im UML Modell durch annotierte Modellelement dargestellt. Die Annotation erfolgt auf Grundlage der UML Stereotypen, wodurch auch eine Zuweisung von Eigenschaftswerten an Tagged Values möglich ist. Als Beispiel ist eine OSEK Applikation als Kollaborationsdiagramm angegeben. Jedes OSEK Objekt ist durch ein annotiertes UML Element dargestellt. Eigenschaften der OSEK Objekte erfolgen durch Zuweisungen an die Tagged Values. Dadurch ist das resultierende Diagramm nicht besonders übersichtlich.

Die Arbeit von Zonghua Gu et al. [GWS03] verfolgt einen ähnlichen Ansatz. Auch dort werden Stereotypen verwendet, um die spezielle Bedeutung von Modellelementen zu beschreiben. Im Gegensatz zur Arbeit von Moore wird das RT-Profil aber nicht erweitert. Die schon definierten Stereotypen des RT-Profil werden auf Objekte von OSEK abgebildet.

Beide Arbeiten verwenden Stereotypen um die UML auf den speziellen Anwendungsfall (hier OSEK) zu erweitern. Auch der Ansatz dieser Arbeit basiert auf der Verwendung von Stereotypen. Im Gegensatz zu den Vorgängerarbeiten wird in dieser Arbeit aber die Version 2 von UML verwendet. Dadurch ergibt sich eine bessere Strukturierung der Erweiterung von UML, da die Erweiterungsmöglichkeit von Version 1.x zu Version 2 eine erhebliche Verbesserung erfahren hat.

3 OSEK/VDX

Die OSEK Vehicle Distributed eXecutive (OSEK/VDX) ist ein Gemeinschaftsprojekt der Automobilindustrie zur Definition eines offenen Standards zur Entwicklung verteilter Steuergeräte. OSEK/VDX wurde 1983 gegründet und ist heute das am häufigsten eingesetzte Betriebssystem im Automotive Bereich. Der gesamte OSEK/VDX Standard umfasst vier Bereiche:

- OSEK-OS: Betriebssystem
- OSEK-OIL: Strukturbeschreibung einer Applikation(textuell)
- OSEK-COM: Kommunikation
- OSEK-NM: Netzwerk Management

3.1 OSEK-OS

OSEK-OS ist ein statisches Betriebssystem, wobei jedes Kernel Objekt schon zur Übersetzungszeit bekannt sein muß. Die wichtigsten Kernel Objekte sind: Task, Event, Ressourcen und Interrupt Service Routine (ISR). Es gibt drei verschiedene Ausführungseinheiten - Task, ISR, und Callback Funktionen. Tasks können durch explizite Aktivierung, oder durch den Empfang von Events zur Ausführung kommen. Konkurrierender Zugriff auf gemeinsame Ressourcen wird durch OSEK Ressourcen geregelt. OSEK verwendet ein Scheduling mit festen Prioritäten. Dadurch muß jeder Task schon zur Übersetzungszeit eine feste Priorität besitzen.

3.2 Task

Ein Task ist die grundlegende Ausführungseinheit in OSEK. Tasks können durch Senden und Empfangen von Events miteinander kommunizieren. Ein einfacher Task (standard) kann keine Events empfangen, wohingegen ein erweiterter Task (extended) Events empfangen kann. Diese Fähigkeit muß zur Übersetzungszeit festgelegt werden, damit für jeden erweiterten Task ein notwendiger Empfangspuffer angelegt werden kann.

Abbildung 1 zeigt das Zustandsmodell eines OSEK Task und die zugehörige Darstellung als UML Klasse. Ein erweiterter Task (*extended*) hat einen zusätzlichen Zustand **waiting**, in dem er inaktiv auf empfangene Events wartet. Weiterhin muß für jeden Tasks festgelegt werden, ob er unterbrechbar (*preemptive*) ist und somit jederzeit von einem Task mit höhere Priorität unterbrochen werden kann.

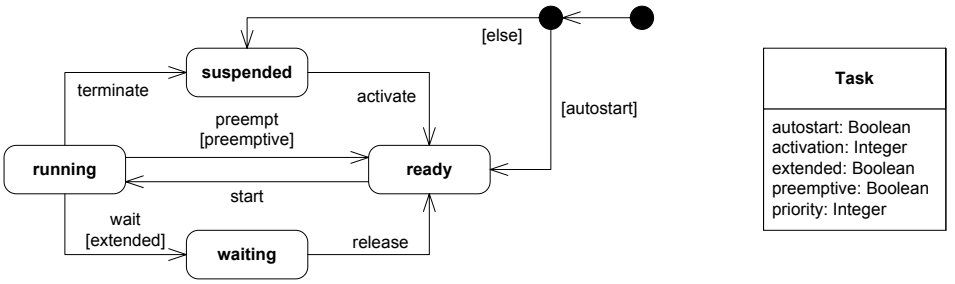


Abbildung 1: Zustandsautomat eines OSEK Task und zugehörige UML Klasse

3.3 Event

OSEK Events dienen zur Synchronisation zwischen erweiterten Tasks. Jeder Task, auch ein einfacher Task, kann Events senden, aber nur erweiterte Tasks können Events empfangen. Da ein OSEK Event als Bit realisiert ist, ist es möglich mehrer Events auf einmal zu schicken oder auf mehrere Events zu warten.

Abbildung 2 zeigt den C Quelltext des Tasks **Sample**, der auf zwei Events wartet und einen Event sendet. Bei Empfang des Events `E1` wird der Event `E3` an den Task **Worker** geschickt. Ein Empfang des Events `E2` beendet die Schleife und auch den Task.

```
TASK (Sample) {
  EventMaskType eMask;
  bool bRunning = true;

  while (bRunning) {
    WaitEvent(E1 | E2);
    GetEvent(Sample, &eMask);
    ClearEvent(eMask);

    if (eMask & E1)
      SetEvent(Worker, E3);
    if (eMask & E2)
      bRunning = false;
  }

  TerminateTask();
}
```

Abbildung 2: Senden und Empfangen von Events innerhalb eines Tasks

```
TASK Sample
{
  AUTOSTART = FALSE;
  PRIORITY = 2;
  SCHEDULE = FULL;
  ACTIVATION = 1;
  EVENT = E1;
  EVENT = E2;
};
```

Abbildung 3: Beschreibung des Task **Sample**

3.4 Zugriff auf gemeinsame Betriebsmittel

OSEK verwendet logische Ressourcen, um den konkurrierenden Zugriff auf gemeinsame Betriebsmittel zu verwalten. Vor dem ersten Zugriff muß die entsprechende Ressource reserviert werden und nach dem Zugriff wieder freigegeben werden. Leider gibt es keine direkte Verbindung zwischen einem gemeinsamen Betriebsmittel und eine logischen Ressource. Vielmehr ist es Aufgabe des Programmierers Zugriffe auf ein gemeinsames Betriebsmittel mit entsprechendem Reservieren bzw. Freigeben einer logische Ressource zu versehen. Weiterhin verwendet OSEK das priority ceiling protocol (PCP) [RSL89], um das Problem der Prioritätenumkehr zu vermeiden.

3.5 Strukturbeschreibung

Zusätzlich zur C API, ist eine textuelle Strukturbeschreibung der Applikation notwendig. Dies erfolgt in der OSEK Implementation Language (OIL). Jedes Kernel Objekt muß mit seinen Eigenschaften dort beschrieben sein. Diese Beschreibung wird während der Übersetzung verwendet, um die ausführbare Applikation zu erzeugen. So wird beispielsweise für jeden erweiterten Task (spezieller Eintrag in OIL-Datei) ein Empfangspuffer generiert. Abbildung 3 zeigt die Beschreibung des Task **Sample**, dessen Verhalten in Abbildung 2 als C Quelltext angegeben ist. Jede Eigenschaft eines Tasks wird als Name-Wert Paar angegeben. Die ersten vier Eigenschaften beschreiben einen Task der nicht automatisch aktiviert wird, eine Priorität von 2 hat, unterbrechbar ist, und nur einmal aktiviert werden kann. Die beiden letzten Eigenschaften müssen auch in der Strukturbeschreibung aufgeführt werden, obwohl sie aus dem Verhalten (C Quelltext) abgeleitet werden können. Alle Events die ein Task empfangen kann, müssen in der Strukturbeschreibung aufgeführt werden. In diesem Fall sind das die Events `E1` und `E2`.

3.6 Entwicklungsprozess

Der Entwicklungsprozess für OSEK Applikationen unterscheidet sich grundlegend von dem Prozess für mächtigere Betriebssysteme (z.B. Linux). Es gibt kein fertiges Betriebssystem welches Applikationen ausführen kann. Bei OSEK wird ein spezifisches Betriebssystem zur Übersetzungszeit erzeugt und mit der Applikation vereint. Der typische Prozess ist in Abbildung 4 dargestellt.

Beide Eingabemengen, Strukturbeschreibung und Quelltext, sind Textdateien, wobei die Strukturbeschreibung (OIL Datei) auch von einem Werkzeug erzeugt werden kann. Die Strukturbeschreibung wird während der Übersetzung ausgelesen und die ermittelten Informationen (z.B. erweiterter Task braucht Empfangspuffer) werden zur Generierung des ausführbaren Applikation verwendet. Der gesamte Quelltext, handgeschrieben oder generiert, wird dann übersetzt und mit vordefinierten Bibliotheken kombiniert. Die resultierende Datei ist ein einzelnes Programm, wobei Betriebssystem und Applikation fest

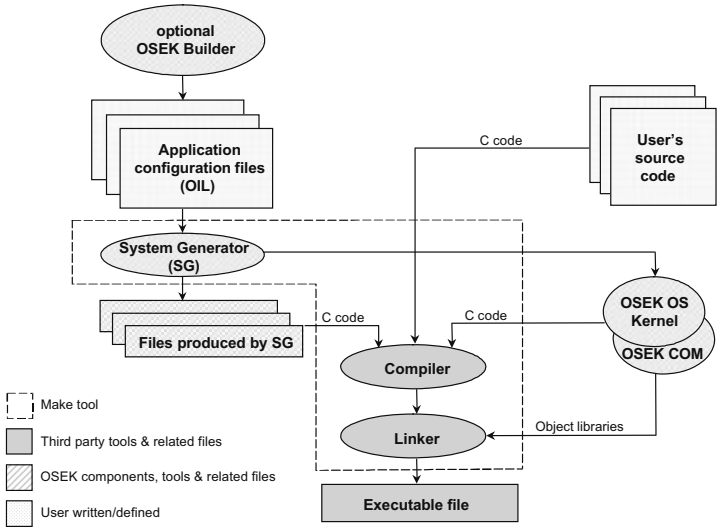


Abbildung 4: Entwicklung mit OSEK nach [OSE04]

miteinander verbunden sind.

4 UML 2

Dieser Abschnitt bietet eine kurze Einführung in die Grundlagen von UML, die für den vorgestellten Ansatz notwendig sind. Ausführliche Informationen findet man in der Spezifikation zur aktuellen Version 2.1 [OMG07].

4.1 Struktur und Verhalten

UML hat dreizehn verschiedene Diagrammarten die in Strukturdiagramme und Verhaltensdiagramme unterteilt werden können. Diagramme bestehen aus Modellelementen, die Instanzen von Klassen aus dem Metamodell sind. Für den vorgestellten Ansatz ist eine Beziehung zwischen Struktur und Verhalten wichtig, um aus dem Verhalten die Struktur zu beschreiben. Abbildung 5 stellt den Ausschnitt aus dem UML Metamodell dar, der die Beziehung zwischen Struktur und Verhalten realisiert.

Die wichtigen Klassen sind **BehavioredClassifier** und **Behavior**, wobei die erste die Oberklasse aller Strukturelemente mit Verhalten ist und die zweite die Oberklasse aller Verhaltenselemente ist. Einem Strukturelement (**BehavioredClassifier**) kann über die Assoizationen *ownedBehavior* und *classifierBehavior* eine Verhaltensspezifikationen zugewiesen werden. Eine Verhaltensspezifikationen (**Behavior**) kann über die Assoziation *context* auf

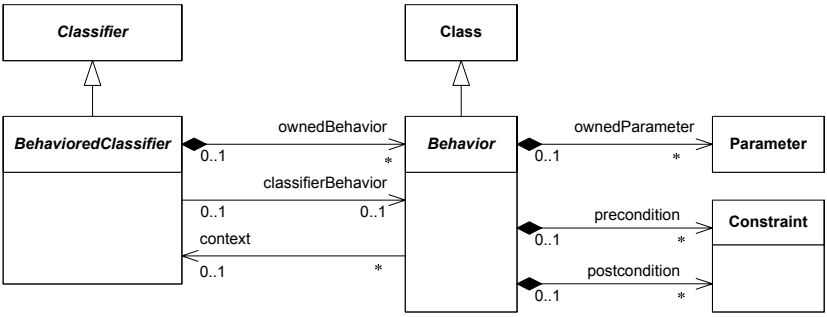


Abbildung 5: Beziehungen zwischen Struktur und Verhalten in UML

ein Strukturelement zugreifen. Die Verhaltensspezifikationen kann eine Menge von Parametern (*ownedParameter*) enthalten, wobei diese Argumente und Rückgabewert beschreiben. Zusätzlich können Bedingungen angegeben werden, die vor (*precondition*) oder nach (*postcondition*) der Ausführung gelten müssen.

Auf Modellebene kann nur mit Instanzen von konkreten Metaklassen gearbeitet werden. Diese konkreten Metaklassen sind in Abbildung 6 dargestellt.

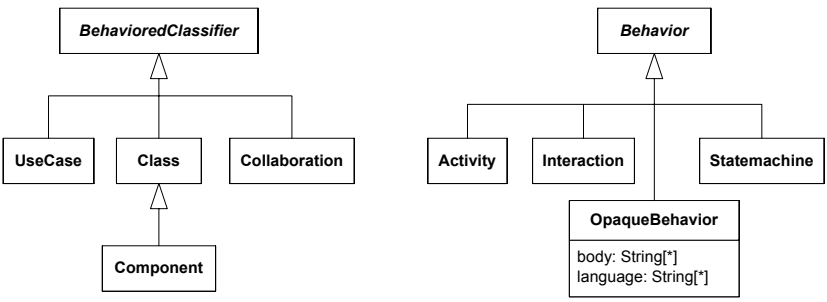


Abbildung 6: Konkrete Klassen zur Modellierung von Struktur und Verhalten

Als Struktruelement können auf Modellebenen Anwendungsfall, Klasse, Kollaboration, und Komponente verwendet werden. Diesen Elementen kann ein Verhalten in Form von Aktivität, Interaktion oder Zustandsmaschine zugewiesen werden. Zusätzlich kann Verhalten in textueller Form einer speziellen Sprache angegeben werden.

4.2 UML Profile

Obwohl die UML eine sehr umfangreiche Sprache ist, können nicht alle möglichen Anwendungsbereiche berücksichtigt werden. Deshalb gibt es einen Erweiterungsmechanismus, damit die UML an bestimmte Domänen angepasst werden kann. Eine Erweiterung ist als UML Profile definiert. Das grundlegende Konstrukt einer Erweiterung ist ein Ste-

reotyp, der eine bestehende Metaklasse erweitert. Ein Stereotyp kann Attribute haben und zusätzliche Constraints tragen.

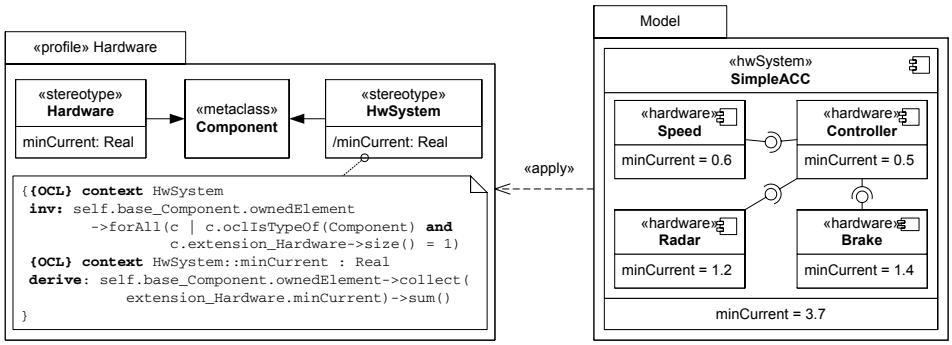


Abbildung 7: Ein UML Profil und seine Anwendung

Abbildung 7 zeigt eine Beispiel dieser Erweiterung. Das Profile **Hardware** definiert zwei neue Stereotypen **Hardware** und **HwSystem**, die beide eine Komponente erweitern. Beide Stereotypen haben eine Attribut, welches den minimalen Stromverbrauch der Komponente enthält. Beim Stereotyp **HwSystem** ist dieses Attribut aber abgeleitet und die Berechnung in Form eines OCL Constraint angegeben. Ein weiteres Constraint sorgt dafür, dass ein **HwSystem** nur Komponenten vom Typ **Hardware** enthält.

5 Abbildung von UML 2 auf OSEK

Dieser Abschnitt beschreibt die Abbildung von UML Modellelementen auf bestimmte OSEK Objekte. Dabei beschränken wir uns auf die drei wichtigsten Objekte - Task, Event und Resource - und Funktionen auf diesen. Die Abbildung des Verhaltens wird zuerst angegeben und danach die Abbildung der Struktur.

5.1 Abbildung von Verhalten

Eine sinnvolle Abbildung des Verhaltens ist besonders wichtig, da der überwiegende Teil einer OSEK Applikation aus Verhaltensbeschreibung besteht. Nach Abschnitt 4 gibt es drei wesentliche Elemente zur Beschreibung von Verhalten. Für jedes dieser Elemente gibt es eine zugehörige Diagrammart, die eine spezifische Notation zur Verfügung stellt. Daher können folgende Diagrammarten verwendet werden:

- Zustandsdiagramm
- Aktivitätsdiagramm
- Interaktionsdiagramm

Ein Zustandsdiagramm ist sehr gut geeignet um das Verhalten in Form von Zuständen und Übergängen zwischen diesen zu beschreiben. Aber zur Darstellung von Tasks, die Events senden und empfangen, ist es weniger geeignet.

Mit Aktivitätsdiagrammen kann Verhalten viel allgemeiner beschrieben werden und man kann Kontrollfuss und Datenfluss beschreiben. Selbst für das Senden und Empfangen existierten eigene Notationen, die aber leider zu restriktiv sind. So kann beim Senden kein Empfänger angegeben werden.

Daher verwendet dieser Ansatz Interaktionsdiagramme zur Darstellung des Verhaltens einer OSEK Applikation. In den folgenden Unterabschnitten werden Betriebssystemfunktionen durch eine bestimmte Art von Interaktionsdiagrammen, den Sequenzdiagrammen, dargestellt. Dabei wird zu einem Sequenzdiagramm auch der äquivalente C Quelltext angegeben. Die einzelnen OSEK Objekte werden dabei durch Lebenslinien repräsentiert und der Aufruf von Betriebssystemfunktionen durch Nachrichten zwischen den Lebenslinien.

5.1.1 Tasks

Im Grunde gibt es nur zwei Arten von Aktionen auf Tasks, die über Betriebssystemfunktionen aufgerufen werden - das explizite Beenden und das explizite Aktivieren.

Ein Task kann nicht von außerhalb beendet werden, sondern nur durch sich selbst. Da wir Sequenzdiagramme zur Darstellung verwenden, benötigen wir eine Notation, die das Beenden auf einer Lebenslinie darstellt. Dafür bietet sich die **DeleteMessage** an. Sie wird als dickes Kreuz am Ende der Lebenslinie dargestellt.

Abbildung 8 zeigt im Sequenzdiagramm **Terminate** einen Task (Stereotype **task**), der nach Abarbeitung seiner eigentlichen Arbeit (Interaktionsreferenz **Work**), sich selbst beendet. Der zugehörige C Quelltext ist im rechten oberen Bereich angegeben. Die durchgeführte Arbeit ist dabei nur durch einen Kommentar angegeben. Eine **DeleteMessage** am Ende einer Lebenslinie repräsentiert daher den Aufruf von **TerminateTask** am Ende des Quelltextes eines Tasks.

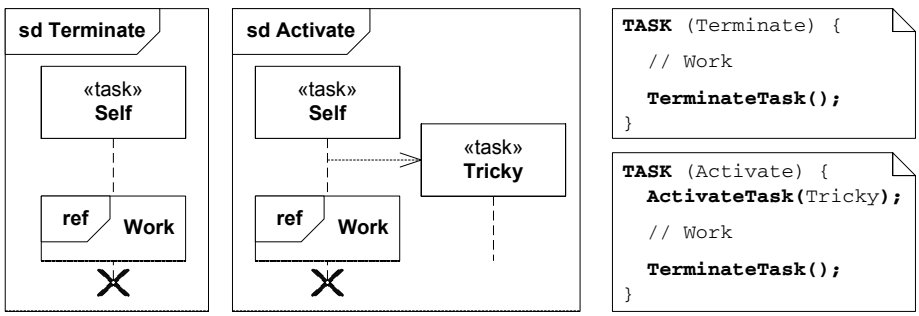


Abbildung 8: Aktivieren und Beenden eines Tasks

Da das Aktivieren die Umkehrung des Beendens ist, verwenden wird dafür auch die entsprechende Notation der Sequenzdiagramme. Damit wird die Aktivierung eines Tasks

durch eine **CreateMessage** dargestellt, die vom aufrufenden Task zum aktivierenden Task zeigt. In Abbildung 8 zeigt das Sequenzdiagramm **Activate** die Aktivierung des Tasks **Tricky** und den zugehörigen C Quelltext im rechten unteren Bereich.

5.1.2 Events

Events sind in OSEK-OS als einfache Bits realisiert und die Empfangspuffer werden durch Bitfelder dargestellt. Dabei wird beim Senden das entsprechende Bit im Empfangspuffer des Empfänger Tasks gesetzt und beim Empfang wird eine Bitmaske mit dem Inhalt des Empfangspuffers verglichen. Für das Senden ist nur die Betriebssystemfunktion `SetEvent` zuständig, während zum Empfang die Funktionen `WaitEvent`, `GetEvent` und `ClearEvent` benötigt werden.

Beim Senden wird neben dem Empfänger noch eine Menge von Events (Bitmuster) angegeben, wodurch mehrere Events gleichzeitig gesendet werden können. Abbildung 9 zeigt das Senden als spezielle Nachricht mit dem Namen **Events** und den einzelnen Events als Parameter der Nachricht.

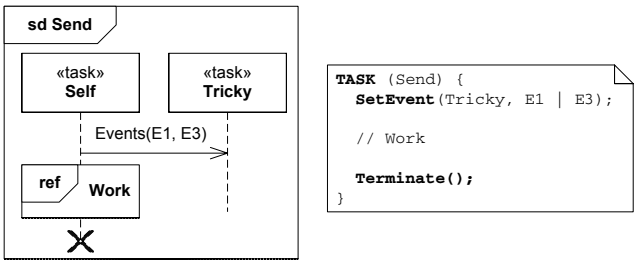


Abbildung 9: Senden von Events

Im zugehörigen C Quelltext auf der rechten Seite der Abbildung sind die Parameter in eine bitweise Verknüpfung überführt worden.

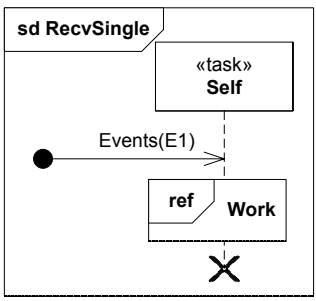
Der Empfang von Events ist wesentlich komplizierter als das Senden. Die gesamte Empfangssequenz ist in die folgenden Schritte aufgeteilt.

1. Warten auf eine Menge von Events `WaitEvent` - blockierend!
2. Holen aller empfangenen Events `GetEvent`
3. Löschen der bearbeiteten Events `ClearEvent`

Da der Aufruf von `WaitEvent` blockierend ist, wird der aktuelle Task in den Zustand *waiting* gesetzt und erst wenn mindestens einer der angegebenen Events empfangen wurden, wird der Zustand auf *ready* gesetzt und der Task kann mit der Bearbeitung der Events fortfahren. War beim Senden die Angabe des Empfängers notwendig, so ist beim Empfang der Absender eines Events unerheblich. Der Empfang von Events erfolgt daher ohne Angabe eines Absenders. Da die Notationen zum Senden und Empfangen möglichst ähnlich

aussehen sollten, wird für den Empfang die Darstellung einer Nachricht ohne Absender benötigt. Diese Darstellung gibt es in Form einer *unvollständigen* Nachricht erst seit UML 2.0.

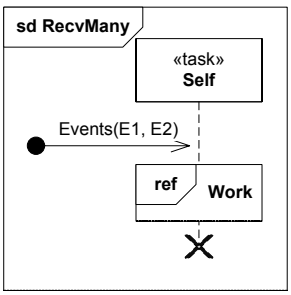
Der Empfang eines einzelnen Events ist in Abbildung 10 dargestellt. Der Empfang ist wieder als Nachricht mit dem Namen **Events** dargestellt, wobei statt eines Absenders, ein Punkt angegeben ist. Hier wird ein *found signal* verwendet, das den Empfang einer Nachricht repräsentiert, bei der der Absender unwichtig oder unbekannt ist. Der zugehörige C Quelltext auf der rechten Seite zeigt den vollständigen Empfang durch drei Aufrufe der entsprechenden Betriebssystemfunktionen.




```
TASK (RecvSingle) {
  EventMaskType eMask;

  WaitEvent(E1);
  GetEvent(RecvSingle, &eMask);
  ClearEvent(eMask);

  // Work

  TerminateTask();
}
```

Abbildung 10: Empfang eines einzelnen Events

Der Empfang von mehreren Events ist bei der klassischen Programmierung in C etwas komplizierter. Dabei ist es notwendig die tatsächlich empfangenen Events zu ermitteln und diese aus dem Empfangspuffer zu löschen. Erst wenn alle angegebenen Events empfangen worden sind, kann die Bearbeitung fortgesetzt werden.




```
TASK (RecvMany) {
  EventMaskType eMask;
  bool bIncomplete = true;

  while (bIncomplete) {
    WaitEvent(E1 | E2);
    GetEvent(Simple, &eMask);
    bIncomplete = (eMask &(E1 | E2)) != (E1 | E2);
  }
  ClearEvent(E1 | E2);

  // Work

  TerminateTask();
}
```

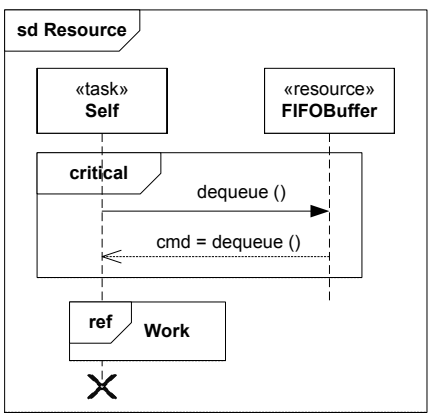
Abbildung 11: Empfang von mehreren Events

Wie Abbildung 11 zeigt, ist die Darstellung als Sequenzdiagramm nicht komplizierter als beim Empfang eines Events. Der entsprechende C Quelltext ist hingegen um einiges komplizierter geworden. Da die Funktion `WaitEvent` schon beim Empfang eines einzigen Events die Unterbrechung des Tasks aufhebt, muß immer überprüft werden, ob tatsächlich

alle (hier `E1` und `E2`) Events empfangen worden sind. Dazu ist eine zusätzliche Schleife nötig, deren Abbruchbedingung in jedem Durchlauf in der Variablen `bIncomplete` aktualisiert wird.

5.1.3 Verwaltung gemeinsamer Betriebsmittel

Der Zugriff auf gemeinsame Betriebsmittel wird in OSEK-OS durch logische Ressourcen kontrolliert. Vor dem Zugriff muß die Ressource reserviert werden (`GetResource`) und nach dem Zugriff wieder freigegeben (`ReleaseResource`) werden.




```
TASK (Resource) {
  char cmd;

  GetResource(FIFOBuffer);
  cmd = dequeue();
  ReleaseResource(FIFOBuffer);

  // Work

  TerminateTask();
}
```

Abbildung 12: Zugriff auf gemeinsame Betriebsmittel

Dies kann man in Sequenzdiagrammen durch *kritische Bereiche* darstellen. Kritische Bereiche sind spezielle Fragmente (**CombinedFragments**) vom Typ **critical**. Die Abbildung 12 zeigt, wie der Zugriff auf gemeinsame Ressourcen modelliert wird. Ein gemeinsames Betriebsmittel wird durch den Stereotyp **resource** gekennzeichnet. Jeder Zugriff darauf muß von einem kritische Bereich umgeben sein, der durch ein Fragment mit dem Namen **critical** dargestellt wird.

5.2 Abbildung der Struktur

Die Struktur der bisher behandelten OSEK Objekte wird auf naheliegende Weise auf Klassen abgebildet. Diese Klassen können dann innerhalb von Sequenzdiagrammen verwendet werden. Tasks haben Eigenschaften, die durch entsprechende Attribute einer Klasse repräsentiert werden. Ressourcen haben zwar keine Eigenschaften, sind aber als Lebenslinien direkt an Interaktionen beteiligt und benötigen daher eine Strukturdarstellung als Klasse. Events sind zwar auch in Sequenzdiagrammen vorhanden, dort aber nur als Nachrichten und benötigen somit keine Strukturdarstellung.

Die notwendige Strukturdarstellung ist in Abbildung 13 als UML Profil angegeben.

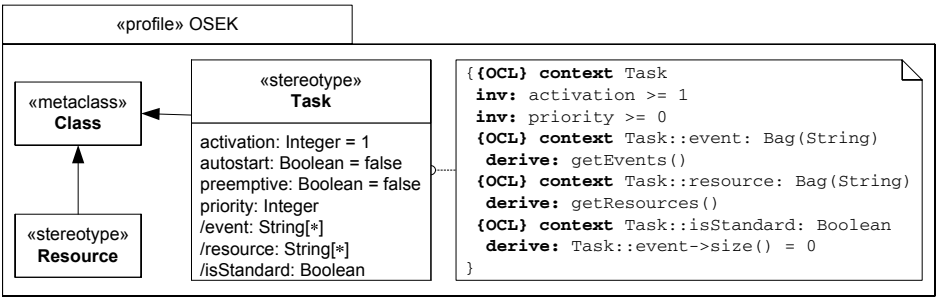


Abbildung 13: UML Profil für OSEK

Eine Ressource ist eine spezielle Klasse die keine weiteren Attribute hat. Der entsprechende Stereotyp **resource** wird zur Unterscheidung von Ressourcen und Tasks verwendet. Ein Task ist auch eine spezielle Klasse, wobei zusätzliche Attribute benötigt werden. Die ersten vier Attribute muß der Entwickler angeben, wobei die Standardwerte schon vorgegeben sind. Die letzten drei Attribute sind abgeleitet und werden aus dem Verhalten des Tasks berechnet. Für das Attribut **isStandard** ist die Berechnung in Form eines OCL Constraints angegeben. Ein OSEK Task ist genau dann ein Standard Task, wenn er keine Events empfängt. Die beiden anderen Attribute werden durch die OCL Funktionen **getEvents()** bzw. **getResources()** berechnet. Diese Funktionen sind aus Platzgründen nicht angegeben, ermitteln aber durch eine Navigation zum zugewiesenen Verhalten (siehe Abbildung 5), die Menge der empfangenen Events bzw. verwendeten Ressourcen.

5.3 Erweiterungen zum Standard

Die bisherigen Abschnitte beschrieben die Modellierung der wichtigsten OSEK Objekte und deren Verhalten mit UML. Neben der Modellierung des OSEK Standards, lassen sich aber auf einfache Weise auch sinnvolle Erweiterungen darstellen.

Der Empfang von Events wird durch ein *found signal* dargestellt. Es gibt auch das zugehörige Gegenstück *lost signal*, welches eine Nachricht ohne Empfänger kennzeichnet. Zwar gibt es in OSEK keine Möglichkeit ein Event ohne Empfänger zu senden, aber die Modellierung solch eines *Broadcast* ist durchaus sinnvoll.

Abbildung 14 zeigt einen Task, der den Event `E1` sendet ohne einen Empfänger angegeben zu haben. Damit soll ein Senden an alle Tasks beschrieben werden, die diesen Event auch empfangen wollen. Bei der Modell-zu-Text Transformation wird dann aus einem Broadcast eine Menge von Sendeaktionen generiert, wobei die jeweiligen Empfänger explizit angegeben sind.

Tasks sind entweder unterbrechbar oder nicht unterbrechbar. Diese Eigenschaft muß zur Übersetzungszeit festgelegt werden und gilt für das gesamte Verhalten eines Tasks. Es gibt in OSEK leider keine nicht unterbrechbare Regionen. Dies kann mit einigem Auf-

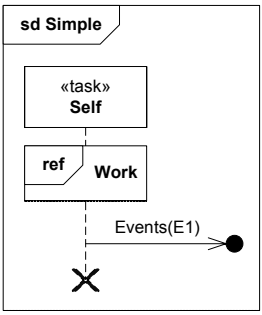


Abbildung 14: Broadcast von Events

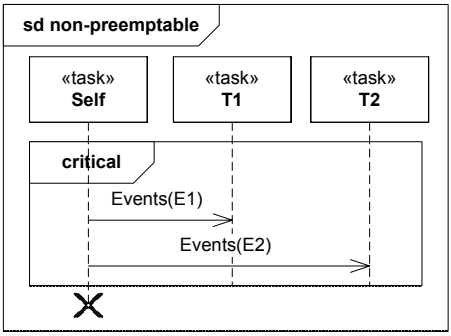


Abbildung 15: Task mit nicht unterbrechbarer Region

wand durch die Verwendung von logischen Ressourcen erreicht werden. Darunter leidet dann aber die Lesbarkeit des entsprechenden C Quelltextes. Abbildung 15 zeigt, wie eine nicht unterbrechbare Region modelliert werden kann. Wie bei dem Zugriff auf gemeinsame Betriebsmittel wird ein kritischer Bereich modelliert. Da in diesem Bereich aber kein Zugriff auf eine Ressource erfolgt, sondern nur Events an Tasks gesendet werden, handelt es sich hierbei um eine nicht unterbrechbare Region.

5.4 Vorteile des modellgetriebenen Ansatzes

Der vorgestellte Ansatz bietet einige Vorteile gegenüber dem klassischen Ansatz. Die beiden wichtigsten Vorteile werden im Folgenden kurz erläutert.

5.4.1 Entwicklungsprozess

Der klassische Entwicklungsprozess für OSEK Applikationen besteht aus zwei separaten Eingabemengen - die Strukturbeschreibung als OIL Datei und das Verhalten in Form von C Quelltexten. Dabei müssen diese Eingabemengen in gewisser Weise kompatibel sein. So muß für einen Task, dessen C Quelltext den Empfang des Events `E1` vorsieht, auch in der Strukturbeschreibung der entsprechende Event aufgeführt sein. Fehlt dieser Event, wird kein Empfangspuffer angelegt und der Task produziert zur Laufzeit Fehler.

Ein Sequenzdiagramm wird verwendet, um das Verhalten eines einzelnen Tasks zu beschreiben. Dadurch kann mit einer Menge von Sequenzdiagrammen das Verhalten einer gesamten Applikation beschrieben werden. Einen Überblick über die gesamte Kommunikation ermöglichen die Kommunikationsdiagramm, die vom exakten Verhalten der einzelnen Bestandteile abstrahieren. Die Kommunikation von drei Tasks ist in Abbildung 16 dargestellt. Von den fünf verschiedenen Events schickt nur Task3 den Event `E5` an Task2.

Der modellgetriebene Ansatz vereint Struktur und Verhalten in einem UML Modell und vermeidet dadurch Unstimmigkeiten zwischen Struktur und Verhalten. Aus diesem Modell

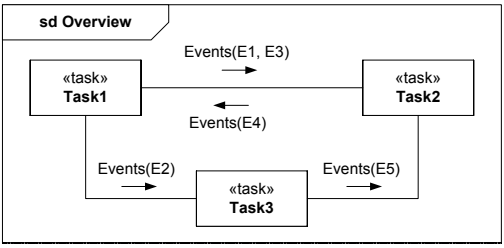


Abbildung 16: Überblick der Kommunikation von drei Tasks

wird durch Modell-zu-Text Transformation sowohl die Struktur als OIL Datei, als auch eine Menge von C Quelltexten erzeugt. Diese erzeugten Dateien können dann an ein bestehendes Werkzeuge übergeben werden, welches die ausführbare Applikation generiert. Somit kann der modellgetriebene Ansatz als Front-End des klassischen Entwicklungsprozesses angesehen werden.

5.4.2 Vermeidung von Laufzeitfehlern

Die Aufrufe von Betriebssystemfunktionen sind durch Nachrichten in Sequenzdiagrammen modelliert. Dabei sind die Parameter nicht explizit angegeben, sondern durch den Kontext implizit festgelegt. Zur Laufzeit können Aufrufe von Betriebssystemfunktionen in Abhängigkeit vom aktuellen Zustand und den Parametern auch scheitern. Interessant ist dabei, inwieweit die implizite Angabe der Parameter bestimmte Laufzeitfehler verhindert.

Als Beispiel wird hier die Funktion `ActivateTask(void)` betrachtet. Abbildung 17 zeigt die vollständige Spezifikation dieser Funktion aus [OSE05].

`StatusType ActivateTask(void)`

Status	Description
`E_OK`	no error
`E_OS_ID`	task id is invalid
`E_OS_LIMIT`	too many task activations

Abbildung 17: Spezifikation der Funktion `ActivateTask`

Der erste Rückgabewert (`E_OK`) kennzeichnet eine erfolgreiche Abarbeitung und ist somit kein Fehlerfall. Ein ungültiger Task wird durch `E_OS_ID` signalisiert. Dieser Fehlerfall kann mit dem modellgetriebenen Ansatz aber nie auftreten, da die Modellierung der Funktion `ActivateTask` immer als Nachricht zwischen zwei Tasks erfolgt. Wenn ein Task zu oft aktiviert wurde, liefert die Funktion den Wert (`E_OS_LIMIT`). Dieser Fehlerfall ist abhängig vom Laufzeitverhalten und kann zur Modellierungszeit nicht entschieden werden. Generell kann fast die Hälfte aller Fehlerfälle zur Modellierungszeit vermieden werden, da die meisten Parameter durch die Modellierung korrekt sind so zur Laufzeit keine Fehler produzieren können.

6 Zusammenfassung und Ausblick

In diese Arbeit wurde ein Ansatz zur modellgetriebenen Entwicklung von OSEK Applikationen beschrieben. Der Kern des Ansatzes ist die verwendete Modellierungssprache UML. Diese visuelle Sprache ist, ähnlich wie OSEK, ein offener Standard, wodurch zahlreiche Werkzeuge zur Modellierung vorhanden sind. Dadurch kann der beschriebene Ansatz auch mit einer Vielzahl von Werkzeugen realisiert werden.

Die Abbildung von OSEK Objekten auf Modellelemente wurde durch ein UML Profil realisiert. Die Struktur einer OSEK Applikation wurde auf naheliegende Weise auf ein UML Klassendiagramm abgebildet. Der wesentliche interessante Fall ist die Abbildung des Verhaltens. Hierfür wurden UML Sequenzdiagramme gewählt, wodurch die Aufrufe von Betriebssystemfunktionen als Nachrichten modelliert werden konnten.

Weiterhin ergeben sich durch bestimmte Einschränkungen in der Notation von UML einige positive Konsequenzen für die Validierung zur Modellierungszeit. So können in Sequenzdiagrammen die kritischen Bereiche nicht überlappen. Da ein kritischer Bereich dem Zugriff auf eine OSEK Ressource entspricht, wird der überlappende Zugriff allein durch die Notation vermieden und kann daher zur Laufzeit nicht auftreten.

Zur Zeit wird der beschrieben Ansatz mit dem UML Werkzeug Visual Paradigm [VP007] erprobt. Der aktuelle Prototyp erzeugt aus einem UML Modell die notwendigen Dateien für die OSEK Implementierung RTA-OSEK [RTA07].

Literatur

[GWS03] Zonghua Gu, Shige Wang und Kang G. Shin. Issues in Mapping from UML Real-Time Profile to OSEK. In *Workshop on Specification and Validation of UML models for Real Time and Embedded Systems (SVERTS)*, 2003.

[Moo02] Alan Moore. Extending the RT Profile to Support the OSEK Infrastructure. In *Symposium on Object-Oriented Real-Time Distributed Computing*, Seiten 341–347, 2002.

[OMG07] OMG. *UML 2.1.1 Superstructure Specification*. OMG, 2007. `http://www.omg.org/cgi-bin/apps/doc?formal/07-02-05.pdf`.

[OSE04] OSEK/VDX. *OSEK OIL Version 2.5*, 2004. `http://www.osek-vdx.org`.

[OSE05] OSEK/VDX. *OSEK OS Version 2.2.3*, 2005. `http://www.osek-vdx.org`.

[RSL89] R. Rajkumar, L. Sha und J. P. Lehoczky. An experimental investigation of synchronization protocols. *IEEE Real-Time Syst. Newsl.*, 5(2-3):11–17, 1989.

[RTA07] ETAS Group. *RTA-OSEK v5.0.0*, 2007. `http://www.etasgroup.com`.

[VP007] Visual Paradigm Intl. *Visual Paradigm 6.0*, 2007. `http://www.visual-paradigm.com/`.