

## 1. Grafische Programmierung vs. klassische Hochsprachenprogrammierung

Es ist heute unbestritten, dass die grafische Programmierung gegenüber der reinen Hochsprachenprogrammierung immense Vorteile hat. Eine bessere Übersichtlichkeit bei Hierarchien und Strukturen, sowie die grafische Darstellung eines logischen Ablaufes sind für den Menschen einfach besser nachvollziehbar.

Kann der Mensch bei einer Größenordnung von 50 Zeilen C-Code nach wenigen Minuten im allgemeinen noch eine Aussage über die Funktionalität des Codes treffen, so ist er bei einer typischen Projektgrößenordnung von 20.000 bis 500.000 Zeilen Code maßlos überfordert. Während der Compiler noch klaglos übersetzt sieht der Mensch den Wald vor lauter Bäumen nicht mehr.

Das liegt zum einen natürlich an der enormen Anzahl von Zeilen, zum anderen aber an den Ausdrucksmöglichkeiten der klassischen Hochsprachen, egal ob funktional programmiert mit C oder objektorientiert entworfen mit C++ bzw. Java.

Keine der Sprachen bietet Mechanismen um die Parallelität und Priorität von Tasks, die asynchrone Verarbeitung von Ereignissen oder einen einfaches Locking ausreichend zu beschreiben. Wir machen uns zwar die Dienste von Echtzeit-Betriebssystemen zu nutze und verwenden für die Code-Generierung Design Patterns, aber im erzeugten Code ist von der ursprünglichen Absicht nicht mehr viel zu sehen, das Debugging erweist sich als schwierig.

Es ist nicht verwunderlich, dass versucht wurde, grafische Notationen zu entwickeln, die eine beherrschbare Software-Entwicklung ermöglichen. Als Beispiele seien Flussdiagramme und Nassi-Shneiderman-Diagramme für logische Abläufe, die IEC 61131-3 für die SPS-Programmierung im Maschinenbau und die UML (Unified Modelling Language) für objektorientierte Systeme genannt.

## 2. Automaten-Programmierung heute mittels SPS

Die SPS (Speicher programmierbare Steuerung) ist heute im Maschinenbau und in der Automatisierungstechnik neben der Hochsprachenprogrammierung in C wohl die am häufigsten vertretene Art der Programmierung. Mit der IEC 61131-3 gibt es einen seit Jahren normierten und standardisierten herstellerbergreifend verfügbaren Standard für ein Programmiersystem.

Mit AWL (Anweisungsliste), KOP (Koppelplan), FBD (Funktionsblockdiagramm), FUP (Funktionsplan), ST (Strukturierter Text) und AS (Ablaufsprache) existieren mehrere sich ergänzende Eingabearten für die Programmerstellung.

Das Konzept der SPS ist sehr stark Datenorientiert. Es basiert auf der Definition von Datenbausteinen die als Eingangs- und Ausgangsgrößen der einzelnen Funktionsblöcke dienen. Einzelne Funktionsblöcke bewerten mit einstellbaren Zykluszeiten ihre Eingangsgrößen, führen logische oder arithmetische Operationen mit ihnen aus und legen das Ergebnis als Ausgangsgröße wieder ab. Jede Ausgangsgröße eines Blocks kann natürlich wieder Eingangsgröße eines anderen Blocks werden. Neben echten Speicherstellen können aber auch Hardware Ein- und Ausgänge wie Datenbausteine verknüpft werden.

Das Debugging erfolgt im allgemeinen über einen externen Datenmonitor, der die Verwendung jedes Speicherbausteins kennt und diesen auch überwachen und modifizieren kann. Jede interne Zustandsvariable kann somit global sichtbar gemacht werden. Auch Prozessvisualisierungen greifen entweder direkt oder über eine Schnittstelle auf diese globalen Datenbausteine zu.

In den SPSen selbst werden unterschiedlichste Konzepte eingesetzt. In der klassischen SPS läuft meist ein herstellereigenes Laufzeitsystem, das den erzeugten Programmcode wahlweise interpretiert oder linkt und anspringt. In den meisten Soft-SPS Systemen läuft dort ein RTOS. Aus den SPS-Automaten wird hier ein C-Code, welcher das RTOS-API verwendet, generiert und später nach erfolgreicher Compilierung als eigenständiger Prozess oder Thread des Betriebssystems gestartet.

Für Automaten begrenzter Komplexität wird dieses Konzept der globalen Datenbausteine und ihrer Zuordnung zu Funktionsblöcken gerne und erfolgreich genutzt. Soll die SPS aber nicht nur vordefinierte Abläufe steuern, sondern wie z.B. in einer PKW-Fertigungsstraße auch komplexe Produktionsdatensätze übernehmen, auswerten und auf sie reagieren, so ist sie überfordert. Dann wird entweder ein weiterer PC vorgeschaltet der derartige Datensätze auswertet und die SPS entsprechend über Datenbausteine konfiguriert, oder aber man greift zur Soft-SPS bei der man noch Zusatzfunktionen in einer Hochsprache wie C oder C++ realisieren und als weitere Task im Betriebssystem starten kann.

Einen smarten Ansatz bietet hier die UML. Bei ihr kann man ebenso einfach wie bei der SPS grafisch programmieren, dabei aber zugleich die Vorteile der Objektorientierung wie Schnittstellen, Datenkapselung und Wiederverwendbarkeit von Komponenten nutzen.

### **3. Modellbasierte Entwicklung mit UML, basierend auf Objekten, Statecharts und wiederverwendbaren Komponenten**

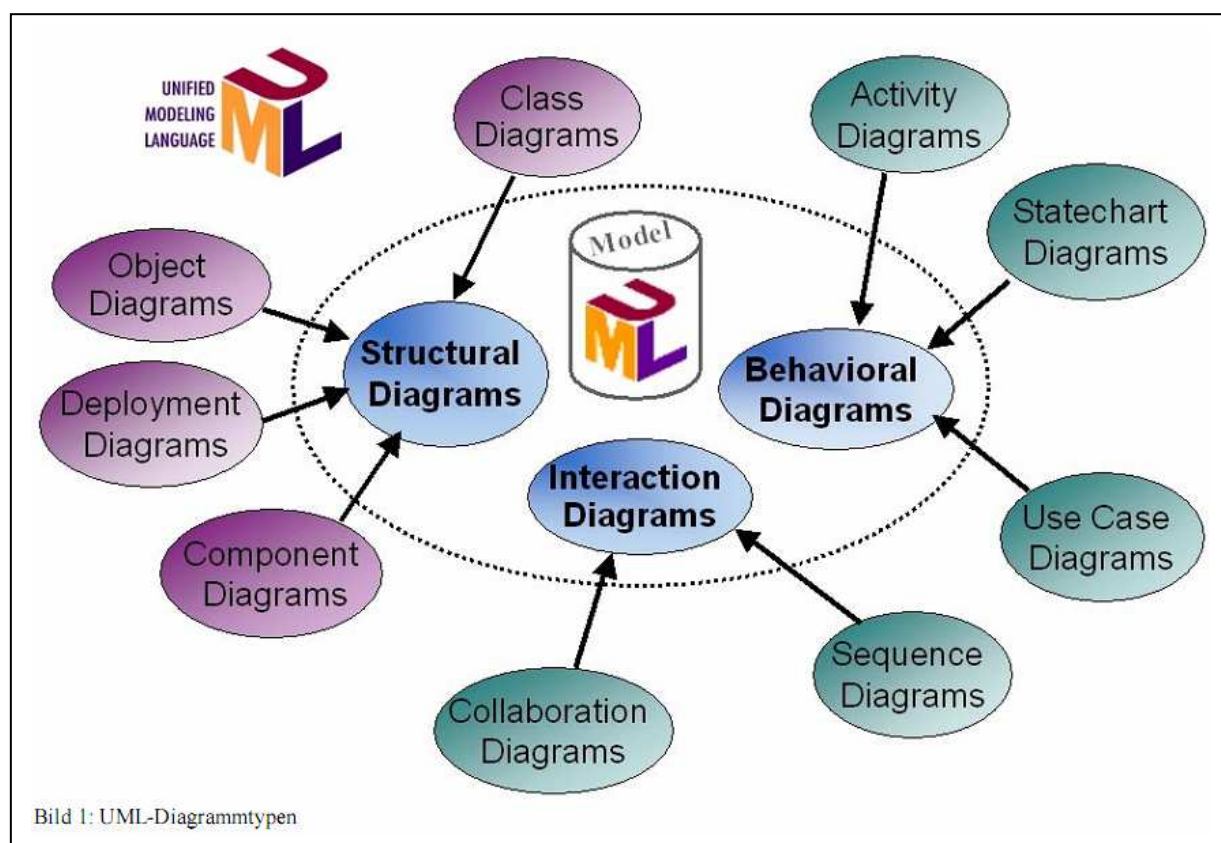
Die Unified Modelling Language ist im Bereich der IT-Software längst zur Standard-Modellierungssprache geworden, den Bereich der reaktiven Echtzeitsysteme in der

Automatisierung beginnt sie jetzt mit der Unterstützung spezialisierter Tools zu besetzen.

Die UML bietet zur Modellierung von Systemen insgesamt neun Diagrammtypen an. Sie können in drei Kategorien eingeordnet werden. Für die Beschreibung der Systemarchitektur dienen Klassendiagramm (Prototyp für die Erzeugung von Objekten, beschreibt Attribute, Methoden und Beziehungen der Klassen), Objektdiagramm (Instanziierung von Klassen und ihren Beziehungen), Komponentendiagramm (Zusammensetzung einer SW-Komponente aus Objekten) und Deploymentdiagramm (Verteilung der SW-Komponenten auf die Hardware). Das Zusammenwirken von Objekten und die Interaktionen (Nachrichtenaustausch) zwischen den Objekten werden mittels Kollaborationsdiagrammen und Sequenzdiagrammen dargestellt. Die Anwendungen des Systems und sein detailliertes Verhalten werden mit UseCase Diagrammen, Statechart Diagrammen und Aktivitätsdiagrammen beschrieben.

Im Kontext dieses Beitrages werden nur vier für die Codegenerierung und Laufzeitanalyse von Automaten wesentliche Diagramme (Klassendiagramme, Objektdiagramme, Statechart-diagramme, Sequenzdiagramme) und ihre Anwendung mit dem UML-Tool Rhapsody von Telelogic betrachtet. Für die spezifische Implementierung werden hier nur C und C++ als Programmiersprachen berücksichtigt. Rhapsody unterstützt aber auch alle anderen Diagrammtypen und bietet weiterhin noch Codegenerierung für Java und Ada.

Zum Thema generelles methodisches Vorgehen bei der Analyse und dem Design mit der UML und für eine detaillierte Beschreibung aller Diagramme möchte ich auf Standardliteratur wie „Doing Hard Time“ von Bruce Powel Douglass verweisen.



### 3.1. Architektur eines UML-basierten Automaten

Wie kann man beim Design eines objektorientierten Automaten beginnen?

Als Beispiel soll hier eine sehr simplifizierte Steuerung für eine automatische Schiebetür dienen. Die Steuerung soll auf zwei unabhängige Bewegungssensoren auf jeder Seite der Tür reagieren, einen Motor zum Öffnen und Schließen der Tür ansteuern, mittels zwei Positionssensoren Tür offen und Tür geschlossen detektieren und mit zeitlicher Verzögerung die Tür automatisch wieder schließen.

Um das Modell der Software so modular und wiederverwendbar wie möglich zu gestalten, werden im Design wiederverwendbare Objekte benutzt.

Die Basis für jede Art von Objekt ist zunächst eine Klasse. Sie definiert die Daten, die das Objekt später intern braucht, sie definiert die Methoden um darauf zuzugreifen, die Beziehungen die zu anderen Objekten hin existieren können, sowie bei Bedarf auch ein reaktives Verhalten (den Zustandsautomaten) für das Objekt mittels eines Statechartdiagramms.

Um verschiedene Mechanismen der Modellierung zeigen zu können, wurden unterschiedliche Arten von Klassen definiert. Eine Klassenübersicht wird in Bild 2 gezeigt.

Im Zentrum des Diagramms (Bild 2) steht die Klasse Door, welche die Steuerung des Systems übernimmt. Diese Klasse ist als aktiv definiert (erkennbar am dicken Rahmen) und besitzt ein eigenes reaktives Verhalten (erkennbar am State-Symbol rechts oben in der Klasse).

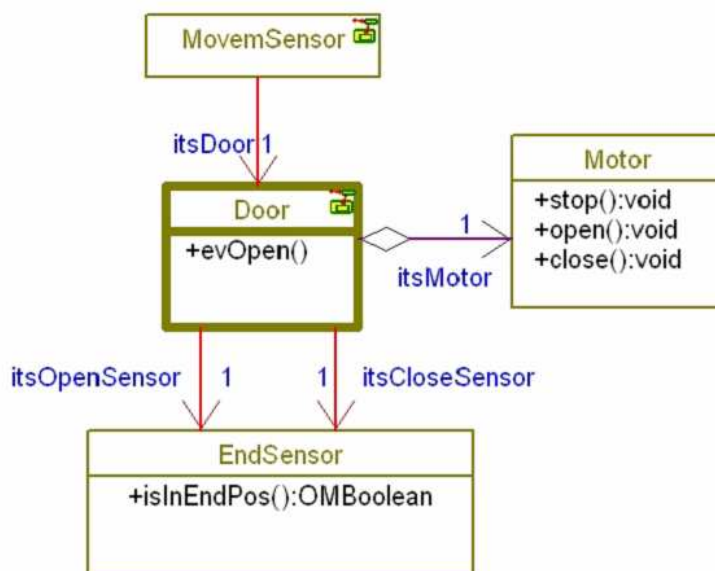


Bild 2: Klassendiagramm für die Türsteuerung

Diese Klasse Door hat Beziehungen zu Klassen vom Typ EndSensor (jeweils eine gerichtete Assoziation) und vom Typ Motor (eine gerichtete Aggregation). Für die Implementierung bedeutet dies, dass es in der Klasse Door zwei Pointer vom Typ EndSensor\* mit den Namen itsOpenSensor und itsCloseSensor und einen Pointer vom Typ Motor\* mit dem Namen itsMotor gibt. Da diese

Beziehungen gerichtet sind (am Pfeil erkennbar), existiert der Pointer in die Gegenrichtung bei den Klassen EndSensor und Motor nicht. Die Klasse Door kann auf Message Events vom Typ evOpen reagieren.

Die Klassen EndSensor und Motor im Beispiel sind beide einfache Klassen, die ein Methoden-Interface zur Verfügung stellen. Die Klasse Motor besitzt die Methoden stop(), open() und close() die keinen Rückgabewert liefern (void), die Klasse EndSensor besitzt die Methode isInEndPos(), die einen Wert vom Typ OMBoolean zurückliefert.

Die Klasse MovemSensor ist wieder eine reaktive Klasse (sie besitzt ein Statechart) und kann über die gerichtete Assoziation (Pointer vom Typ Door\* mit dem Namen itsDoor) auf die Klasse Door zugreifen. Im Unterschied zur Klasse Door ist sie aber nicht aktiv (kein dicker Rahmen).

Der dicke Rahmen der Klasse Door besagt dass jede spätere Instanz dieser Klasse (jedes Objekt vom Typ Door) in einer eigenen Thread läuft. Die Codegenerierung von Rhapsody sorgt wiederum dafür, dass dieses Verhalten ohne weiteres Zutun des Programmierers erreicht wird.

Reaktive Klassen erben von der Basisklasse OMReactive des Rhapsody Frameworks und nutzen somit über ein abstraktes Interface die Message Queues eines darunter liegenden Echtzeitbetriebssystems (RTOS) zum Queuen von ankommenden Messages Aktive Klassen erben zusätzlich automatisch von der Basisklasse OMThread des Rhapsody Frameworks und werden damit in einer eigenen Thread des darunter

liegenden RTOS gestartet.

Jede aktive Klasse erhält eine eigene Input-Queue für ihre Events. Diese wird vom Framework automatisch beim Instanzieren des aktiven Objekts zusammen mit der Thread angelegt.

Bei der Codegenerierung mit Rhapsody in C sieht das Ergebnis fast identisch aus. Da C jedoch keine Klassen kennt wird statt einer class Door ein struct Door generiert und anstatt über Vererbung wird die Funktionalität der „Basisklasse“ RiC\_Reactive über eine Komposition eingebunden.

```
/// class Door
class Door : public OMThread, public OMReactive (

/// Constructors and destructors ///
public :

    /// auto_generated
    Door(OMThread* p_thread = OMDefaultThread);

    /// auto_generated
    ~Door();

/// Relations and components ///
protected :

    EndSensor* itsCloseSensor; /// link itsCloseSensor

    Motor* itsMotor;          /// link itsMotor

    EndSensor* itsOpenSensor; /// link itsOpenSensor

/// Framework ///
protected :

    /// ignore
    /// states enumeration:
    enum Door_Enum( OMNonState=0, opening=1,
                    open=2, closing=3, closed=4 );

    int rootState_subState;          /// ignore

    int rootState_active;           /// ignore

};
```

Bild 3: Spezifikation der Klasse Door

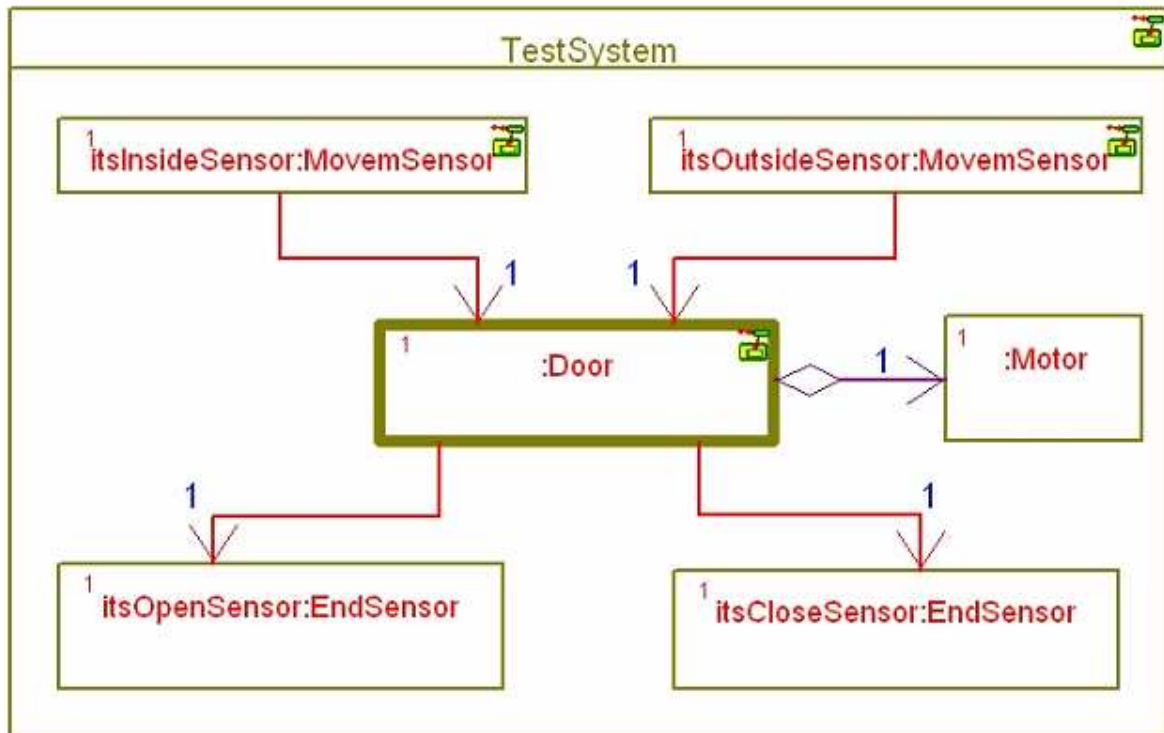


Bild 4: Objektdiagramm

Um die o.g. Klassen durch Instanziierung zu lebendigen Objekten zu machen, braucht man ein Objektdiagramm. Das Objektdiagramm zeigt in einem Testsystem, wie viele Objekte von welchem Typ (von welcher Klasse) mit welchem Namen tatsächlich instanziiert werden und welche Beziehungen realisiert werden.

Das Objektdiagramm (Bild 4) zeigt klar, dass im TestSystem sechs Objekte erzeugt werden. Wenn der gewählte Objektname vom Klassennamen abweicht (z.B. weil mehrere unterscheidbare Objekte als Instanzen der gleichen Klasse gebraucht werden), wird der Instanzname vor der Klasseninstanz (diese ist am vorangestellten Doppelpunkt erkennbar) angeführt.

Für die Generierung des erforderlichen Codes (Bild 5) zur Instanziierung der Objekte sorgt wiederum Rhapsody mit seiner Codegenerierung.

Die Code-Sequenz aus der Implementierung der Klasse TestSystem zeigt, dass Rhapsody sowohl die Objekte erzeugt als auch die Beziehungen zwischen den Objekten initialisiert (in der Methode `initRelations()`) bevor das dynamische Verhalten der Objekte (in `initStatechart()`) wirklich initialisiert wird. Lediglich die Konstruktor-Argumente müssen über entsprechende Dialoge besetzt werden.

```

//## class TestSystem

TestSystem::TestSystem(OMThread* p_thread) : ioValue(0x00) {
    setThread(p_thread, FALSE);
    initRelations();
    initStatechart();
}

TestSystem::~TestSystem() {
    cleanUpRelations();
}

void TestSystem::initRelations() {
    itsCloseSensor = newItsCloseSensor(&ioValue, 0x02);
    itsDoor = newItsDoor();
    itsInsideSensor = newItsInsideSensor();
    itsMotor = newItsMotor();
    itsOpenSensor = newItsOpenSensor(&ioValue, 0x01);
    itsOutsideSensor = newItsOutsideSensor();
    itsDoor->setItsOpenSensor(itsOpenSensor);
    itsInsideSensor->setItsDoor(itsDoor);
    itsOutsideSensor->setItsDoor(itsDoor);
    itsDoor->setItsMotor(itsMotor);
    itsDoor->setItsCloseSensor(itsCloseSensor);
}

```

Bild 5: Initialisierungscode für das Testsystem

### 3.2. Reaktives Verhalten der UML- Objekte

Das reaktive Verhalten der Klassen kann in der UML wahlweise per Aktivitätsdiagramme oder per Statechartdiagramme modelliert werden. Hierzu wird eine Kombination aus der grafischen Notation der UML Statecharts und der jeweiligen Implementierungssprache (hier C bzw. C++) als Action-Sprache in den Statecharts genutzt. Eine neue Programmiersprache muss neben der klassischen Hochsprache (C/C++), die für die Implementierung ohnehin verwendet wird, nicht separat erlernt werden. Die grafische Notation der Statecharts wird lediglich

benutzt, um die erforderlichen Aktionen den gewünschten Zustandsübergängen zuzuordnen. Die Implementierung des Zustandsautomaten übernimmt das Tool. Dies erlaubt Codegenerierung mit minimalem Overhead bei gleichzeitig sehr gut visuell wahrnehmbarer Struktur.

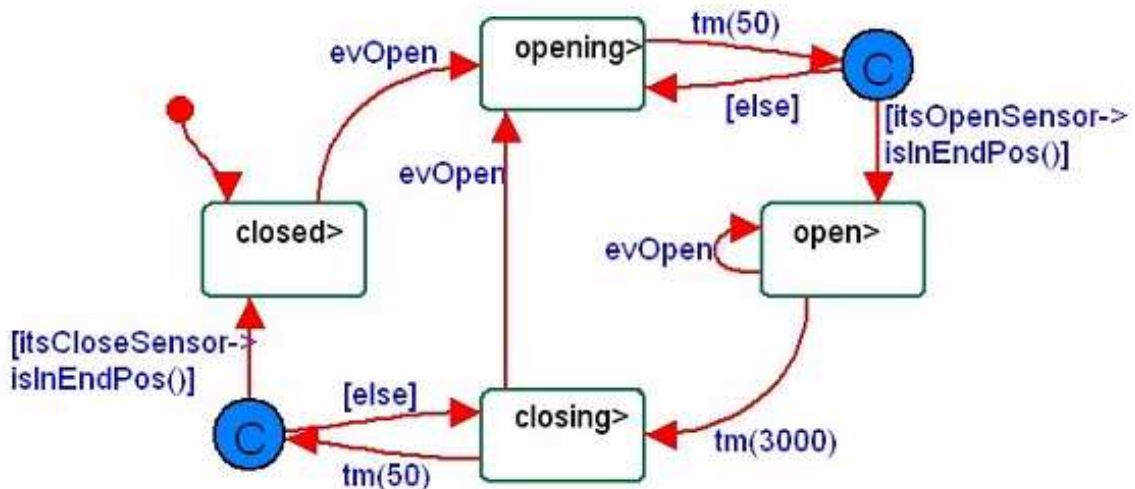


Bild 6: Statechart der Klasse Door

Wie in Bild 6 zu erkennen ist besitzt die Türsteuerung vier Zustände. Die Zustandsübergänge werden entweder durch Events (das sind asynchrone

Messages), durch Timer `tm()` oder über sogenannte Guards (das sind [Bedingungen] mit dem Wert `TRUE` oder `FALSE`) ausgelöst.

Die asynchronen Events werden von außen gesendet (hier `evOpen`, gesendet von einem der Bewegungssensoren), die lokalen Timer `tm(nnn)` werden von einem zentralen `TimerManager` ebenfalls als Events versendet. Wann immer der Automat einen Zustand betritt, von dem aus eine Timer-getriggerte Transition wegführt, wird der `TimerManager` beauftragt nach der definierten Zeit (per Default in ms) ein Timer-Event an das Objekt zu senden. Wird der Zustand wieder verlassen, so wird der Auftrag beim `TimerManager` gelöscht.

Zum Abfragen der Endschalter wird jeweils eine Methode der Klassen vom Typ `EndSensor` direkt (synchron) aufgerufen. Dafür wird der im Klassendiagramm definierte Pointer verwendet. Ein Aufruf (Bild 6) lautet also z.B. `itsOpenSensor->isInEndPos()`. Nur das über den Pointer referenzierte Objekt kümmert sich in seiner Methode `isInEndPos()` darum, was hierfür zu tun ist.

Soll mit Rhapsody Code für C generiert werden sieht der Methodenaufruf ein klein wenig anders aus. Da für eine bessere Codeperformance keine Memberfunktionen

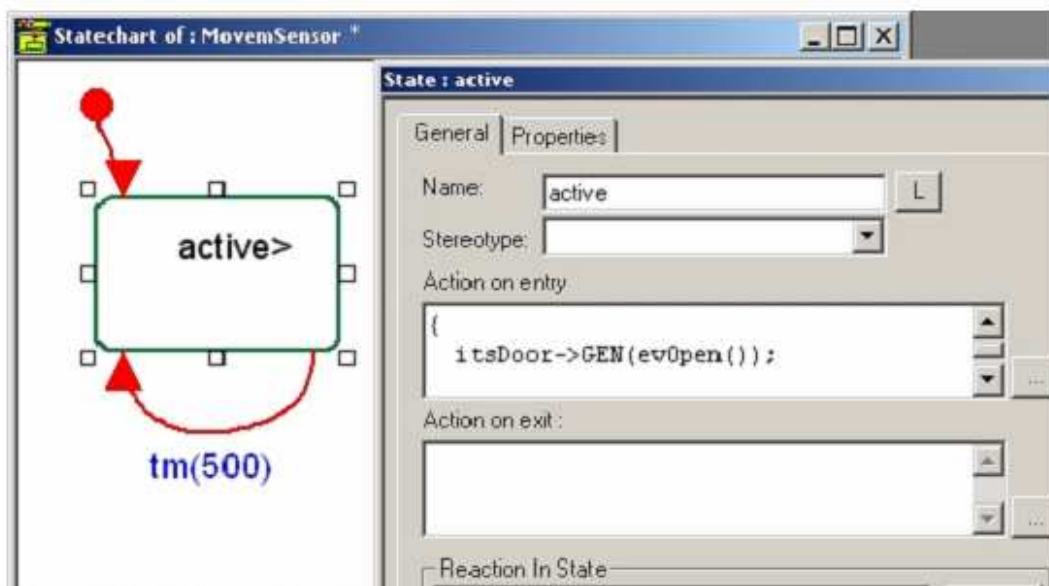


Bild 7: Statechart der Klasse `MovemSensor` und Entry Action im State `active`

wie in C++ über einen Funktionspointer aufgerufen werden, sondern stattdessen eine statische Bindungen bevorzugt wird, lautet der gleiche Aufruf hier `EndSensor::isInEndPos(me->itsOpenSensor)`. Bei der statischen Bindung in C wird der Klassename (hier `EndSensor`) dem Methodennamen (hier `isInEndPos()`) mit einem Unterstrich vorangestellt. Die Adresse der Daten des angesprochenen Objektes (hier `me->itsOpenSensor`) wird als erstes Aufrufargument explizit an die Methode übergeben. Dieses Argument entspricht dem implizit bei C++ vom Compiler übergebenen `this`-Pointer und erfüllt den gleichen Zweck. Der Aufrufer hat diese Adresse in seiner lokalen Datenstruktur gespeichert. Lokal wird die Struktur immer mittels dem Pointer `me` referenziert.

Genauso einfach wie das Aufrufen von Methoden funktioniert auch das Versenden von asynchronen Messages, den Events. In den beiden Klassen vom Typ MovemSensor wird das Event evOpen einfach ber den Pointer zum Empfängerobjekt und das Makro GEN() versendet itsDoor->GEN(evOpen()), einfacher und intuitiver geht's kaum noch (Bild 7). Ein vergleichbares Makro existiert in C, hier lautet der entsprechende Aufruf mit Übergabe der Objektadresse CGEN(me->itsDoor, evOpen()).

Rhapsody liefert die komplette Infrastruktur fr die Realisierung der notwendigen Kommunikationsmechanismen in Form eines Object Execution Frameworks (OXF) im Sourcecode mit. In diesem OXF realisiert ein Operating System Abstraction Layer (OSAL) das Interface zu nahezu jedem beliebigen Echtzeitbetriebssystem, egal ob es sich dabei um ein kommerziell vertriebenes RTOS oder um einen proprietären selbst geschriebenen Scheduler handelt. Fr die gängigsten Betriebssysteme sind fertige OSAL im Quellcode verfügbar.

### 3.3. Modellvalidierung und der Weg vom Modell zum Zielsystem

Modellierung und Codegenerierung sind zwar ein wichtiger Bestandteil eines grafischen Modellierungstools, doch echte Vorteile in der Praxis ergeben sich erst dann, wenn man seine Modelle auch grafisch validieren und direkt ins Zielsystem laden kann. Denn das schönste Werkzeug wird schnell beiseite gelegt, wenn der

generierte Code in jedem Entwicklungszyklus erst langwierig in eine entsprechende Cross-Entwicklungsplattform geladen, dort auf Sourceebene nachbearbeitet, mit einem BSP (Board Support Package) gebunden und dann zum Testen ins Zielsystem geladen werden muss.

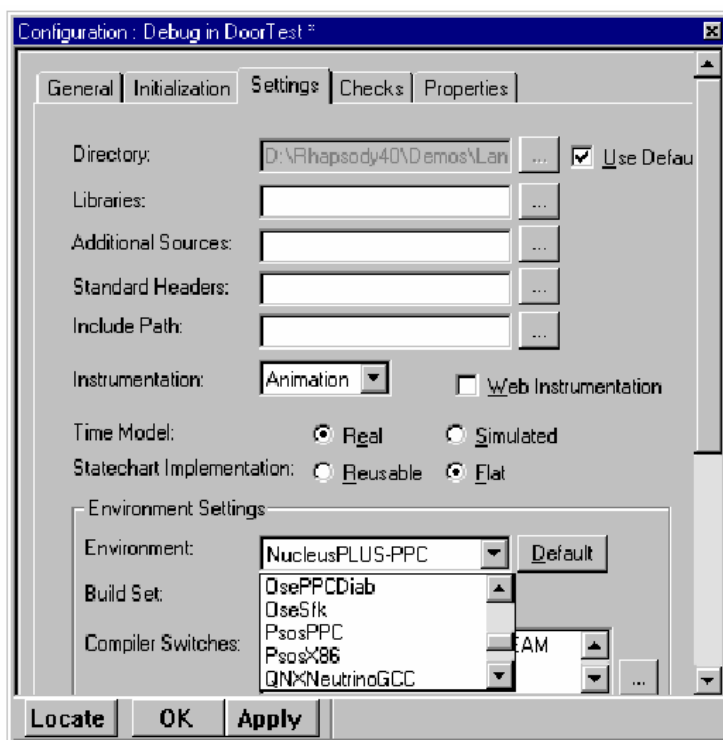


Bild 8: Selektion von Target-Environment und Code-Instrumentierung

Rhapsody bietet neben der reinen Codegenerierung auch die Möglichkeit für jedes einzelne Zielsystem ein oder mehrere Environments mit Hilfe eines individuellen Makefile-Templates und einer OSAL-Anpassung so einzurichten, da? diese

Environments über ein Pulldown Menu einfach selektierbar sind (Bild 8). Damit kann aus Rhapsody heraus nach der Makefile-Generierung auch der Makeprozess für den

Cross-Compiler gestartet werden. Selbst der Download des fertigen Codes ist möglich, wenn die Crossumgebung über eine entsprechende Schnittstelle verfügt.

Richtig interessant wird diese Entwicklungsumgebung dann, wenn man auf dem Host oder dem Target mit Design-Level Debugging beginnt. Rhapsody erlaubt es den Code auf einen einfachen Knopfdruck hin bei Bedarf so zu instrumentieren, daß die internen Attribute und Zustände der Objekte und die Kommunikation der Objekte

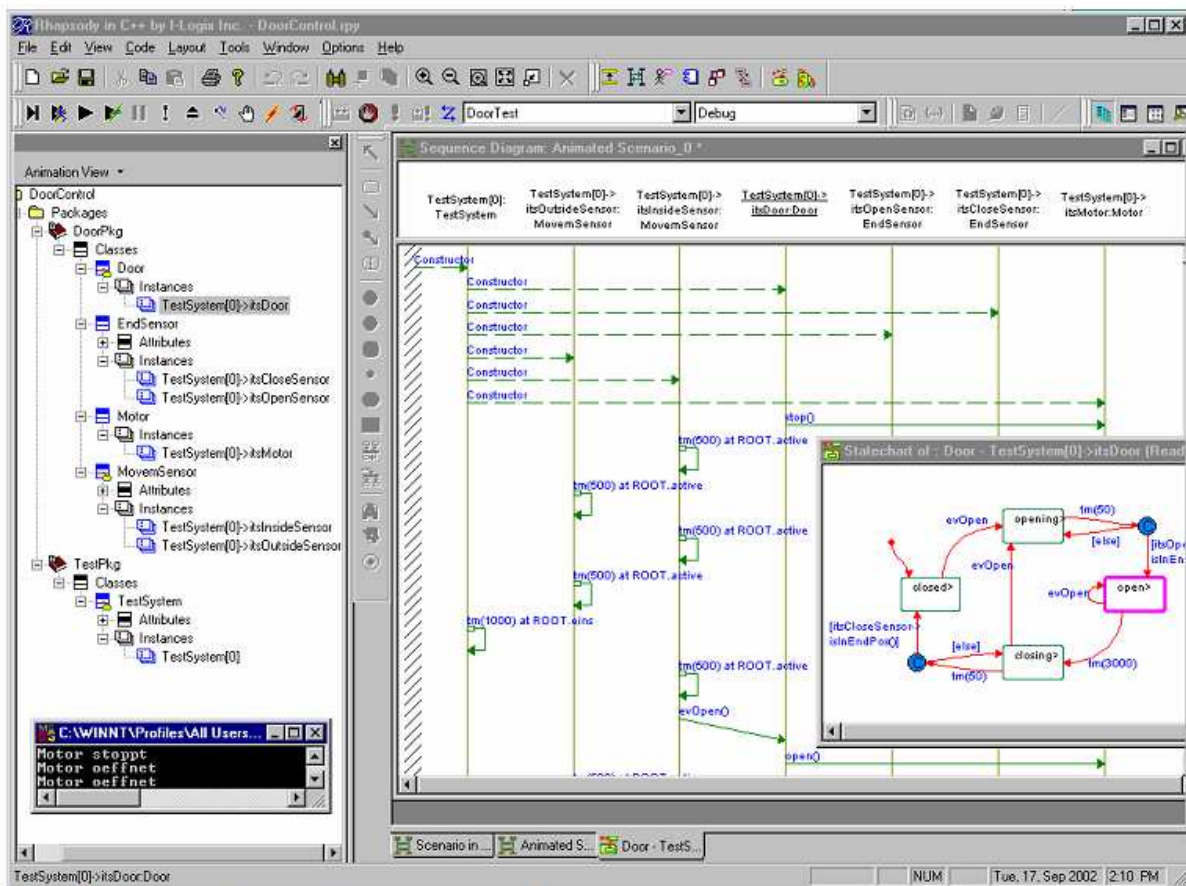


Bild 9: Design-Level Debugging auf dem Entwicklungs-Host

untereinander grafisch auf UML-Ebene visualisiert werden können.

Damit erreicht man beim Debuggen von objektorientierten Systemen bislang ungeahnte Transparenz (Bild 9). Animierte Sequenzdiagramme protokollieren ähnlich wie ein Logikanalyzer die Kommunikation zwischen Objekten, animierte Statechartdiagramme zeigen die Zustände für jedes einzelne Objekt, im Browser sieht man jede Instanz einer Klasse mitsamt ihren Laufzeitattributen. Das Modell kann interaktiv über Events stimuliert und über Breakpunkte gezielt unterbrochen werden. Bei Bedarf kann auf Knopfdruck (Bild 8) Web-Instrumentierung aktiviert werden. Danach erlaubt ein einfach konfigurierbarer, in der Applikation integrierter Webserver den Zugriff auf alle gewünschten Variablen, Events und Methoden über einen Web-Browser. Dies erlaubt die einfache Erstellung von Wartungsoberflächen und Bedienschnittstellen, die unabhängig von Rhapsody über ein Intranet oder sogar übers Internet mit der Applikation kommunizieren.

Das Design-Level-Debugging erfolgt wie gewohnt interaktiv durch den Benutzer. Bei Bedarf natürlich auch parallel oder alternativ mit einem klassischen Source-Level-Debugger wie er mit vielen Cross-Entwicklungsumgebungen mitgeliefert wird.

Daneben bietet Rhapsody mit dem TestConductor weitere Unterstützung um Tests zu automatisieren. Der TestConductor stimuliert das Modell indem er Events von außen sendet oder aber Methoden aufruft. Zur Definition der Tests, also zur Definition der Folge von Aktionen die der TestConductor anstößt und zur Definition der bei der Überwachung erwarteten Reaktionen des Systems, benutzt man Sequenzdiagramme wie oben (Bild 9) zu sehen.

## 4. Einsatz in der Praxis und Kombination mit bestehenden Lösungen

Wie muss man sich nun die praktische Arbeit mit einem derartigen Werkzeug vorstellen?

Die Arbeitsteilung ist natürlich sowohl von der Projektgröße als auch von der Teamgröße abhängig. Man kann genauso alleine mit dem Tool arbeiten, wie auch in großen verteilten Teams. In großen verteilten Teams werden meist Configuration Management Tools verwendet um die Projekte zu verwalten. Die Anbindung an diese ist für eine Vielzahl dieser Tools verfügbar, speziell wenn sie mit der Microsoft SCC (Source Code Control) Schnittstelle kompatibel sind.

Inhaltlich kann man zwei wesentliche Aufgabenstellungen beim Automatenbau sehen. Die Gruppe der Programmierer, die Klassen entwerfen und deren Methoden implementieren und die Gruppe der Konfigurierer, die existierende Klassen in ihren Steuerungen verwenden und zum Teil das Verhalten (die Ablauflogik) grafisch noch weiter spezialisieren und Bausteine miteinander kombinieren.

Für die Kommunikation und Integration mit anderen Steuerungsteilen stehen alle Mechanismen der üblichen Betriebssysteme zur Verfügung. Das kann wie in der SPS eine Kommunikation über Speicherstellen sein, aber auch eine nachrichtenbasierte Kommunikation wie in objektorientierten Systemen üblich ist möglich. Zu diesem Zweck werden meist Kommunikationsobjekte entworfen die diesen Job in einer eigenen Thread erledigen.

## 5. Lösungsbibliothek durch Wiederverwendung von Komponenten

Der Ansatz der Objektorientierung fordert natürlich geradezu dazu auf einmal entworfene Logiken und Abläufe immer wieder zu verwenden. Denn die Klassen kapseln ja alle notwendigen Daten und stellen in jeder Instanz automatisch einen eigenen Datensatz zur Verfügung. Die Initialisierung der Daten erfolgt mit Parametern des Konstruktors, eine Definition von Datenbausteinen und die Verknüpfung zu ihnen entfällt. Die Beziehung zu anderen Objekten wird, sofern Kommunikation erforderlich ist, einfach grafisch in Objektdiagrammen definiert.

Und natürlich wird die Funktionalität des neu entworfenen Systems wieder grafisch auf Designebene validiert.

Dipl.-Ing. Martin Stockl  
Telelogic Deutschland GmbH

Kontakt:  
Willert Software Tools GmbH

Tel.: +49 5722 9678  
Email: [Info@willert.de](mailto:Info@willert.de)  
WWW.willert.de