

ENTWICKLUNG VON EMBEDDED REAL-TIME SYSTEMEN MIT DER UML UND QUANTUM

Model Driven Development (MDD) mit der UML wird auch in der Embedded-Entwicklung immer populärer. Kein Wunder: Die grafische Visualisierung in UML-Modellen bietet zum einen einen anschaulichen und leicht verständlichen Überblick über das zu entwickelnde System. Zum anderen besteht die Möglichkeit, direkt aus den Modellen plattformspezifischen Code zu generieren. Mehr Effizienz bei der Entwicklung, höhere Softwarequalität und leichtere Wartbarkeit der realisierten Anwendung – die Vorteile von MDD liegen auf der Hand. Kombiniert man die modellgetriebene Softwareentwicklung zusätzlich mit dem Einsatz von Frameworks, können Effizienz und Qualität noch einmal deutlich gesteigert werden.

Dieser Artikel stellt die Grundidee von MDD kurz vor. Anschließend beschreibt er eine praxiserprobte Lösung für die modellgetriebene Entwicklung von Embedded Real-Time Systemen mit der UML und dem Application Framework Quantum.

Model Driven Development – ein Überblick

MDD stellt die Architektur der zu erstellenden Anwendung in den Mittelpunkt des Entwicklungsprozesses. Ziel ist es, die Fachlichkeit von allen anderen Aspekten der Anwendung, wie z.B. dem Datenzugriff und der Autorisierung, klar zu trennen. Damit soll der Softwareentwickler von lästiger und fehleranfälliger Routinearbeit entlastet und die Softwareentwicklung durch Automatisierung nachhaltig verbessert werden. Erreicht wird dies, indem die Erstellung des Infrastrukturcodes ganzheitlich automatisiert und redundanter, technischer Code in der Anwendungsentwicklung minimiert wird. Dazu sind im Kern die folgenden Schritte nötig:

- Definition und Abgrenzung der Domäne für das bzw. die zu erstellenden Softwaresysteme.
- Entwurf einer passenden Architektur. Dazu gehört die Definition der grundlegenden Architekturmuster für die einzelnen Aspekte der Anwendungen, d.h. der Datenhaltung, Kommunikation, Steuerung, Konfiguration und Schichten. Weiterhin müssen die Basistechnologie festgelegt, notwendige

Frameworks bzw. Fremdkomponenten ausgewählt und Vorgaben für die Implementierung der fachlichen Datenelemente, Container und Business-Objekte getroffen werden. Die Architektur sollte in jedem Fall durch eine so genannte Referenzimplementierung für ausgewählte Anwendungsfälle aus der Domäne validiert werden.

- Entwurf einer passenden, d.h. auf die Domäne und die Architektur zugeschnittenen, Modellierungssprache. Die UML hat sich hierfür bewährt, eine Alternative bieten aber auch domänenspezifische Sprachen (DSL).
- Definition der Transformationsregeln und Erstellung der Transformatoren, falls zusätzliche Modelltransformationen notwendig sind.
- Bereitstellung von Generatoren, die aus dem Modell die Implementierungsartefakte erzeugen.
- Erstellung des Modells.
- Transformation des Modells und Generierung der Implementierungsartefakte.

Das Vorgehen basiert auf dem Entwurf einer stabilen, validierten Architektur als wesentliche Voraussetzung generierbarer Implementierungsartefakte. Die Definition einer klar umrissenen Domäne ist ebenfalls unabdingbar für den Erfolg der modellgetriebenen Entwicklung. Damit bleibt der Umfang der Modellierungskonstrukte überschaubar und die Semantik der Modellierungssprache eindeutig.



Matthias Ehlert (Dipl.-Ing.) (M.Ehlert@microTOOL.de) konnte seit 1985 umfangreiche Erfahrungen in der Software-Entwicklung auf den verschiedensten Plattformen sammeln. Seit 1992 ist er in der Tool-Entwicklung bei microTOOL tätig und leitet seit vielen Jahren verteilte Projekte im In- und Ausland. Matthias Ehlert unterstützt microTOOLs Kunden als Senior Consultant bei dem Einsatz der UML und der Einführung von Entwicklungsprozessen mit dem Ziel, die Effizienz zu verbessern, die Qualität zu sichern und die Risiken zu minimieren.

MDD mit der UML

Der vorangehende Abschnitt zeigt: MDD-Modelle dienen nicht allein der Dokumentation, sie sind vielmehr eine Abstraktion des Programmcodes. Für die Modellierung der Architektur bietet sich die UML an: Klassendiagramme, Aktivitätsdiagramme und Zustandsdiagramme besitzen eine hinreichend formale Präzision, um sie durch vollautomatische Transformation auf technische Klassendiagramme mit Implementierungsklassen abzubilden. Die Implementierungsklassen enthalten bereits generierten Source Code, in dem das Wissen aus dem fachlichen Modell steckt.

Modellieren von Zustandsautomaten

Speziell die Modellierung von Zustandsautomaten stellt für die Embedded Entwicklung eine interessante Basis für die Code-Generierung dar. Zustandsautomaten bilden das dynamische Verhalten eines Systems ab. Mit der Spezifikation der möglichen Zustände, Zustandsübergänge, Ereignisse und Aktionen im „Leben“ eines Systems, geben Zustandsautomaten Antwort auf die Frage: „Wie verhält sich das System in einem bestimmten Zustand, wenn gewisse Ereignisse eintreten?“

Die UML kennt alle benötigten grafischen Konstrukte für den Entwurf von hierarchischen Zustandsautomaten mit parallelen Regionen. Sie definiert Konformitätsregeln für Zustandsdiagramme und bietet damit die Möglichkeit, die Diagramme auf formale Korrektheit zu überprüfen. So muss zum Beispiel bei kom-

plexen geschachtelten Zuständen sichergestellt werden, dass keine Nachrichtenbrüche/-verluste über die Ebenen hinweg modelliert werden. Diese Prüfungen sollten idealerweise durch das für die grafische Spezifikation eingesetzte UML-Tool erfolgen. Die Praxis zeigt, dass die von der UML definierten Regeln sogar teilweise projektspezifisch angepasst bzw. erweitert werden müssen.

Die Zustandsübergänge können mit komplexen Guards versehen werden und die Aktionsaufrufe und Entry- bzw. Exit-Aktionen mit konkreten Parametern. Hier sollte das eingesetzte UML-Tool nicht nur reinen Fließtext unterstützen, sondern den konsistenten Bezug zu Attributen und Methoden der Klassen aus dem UML-Modell herstellen.

Entwurfsrichtlinien für Zustandsautomaten

Der Entwurf der Zustandsautomaten sollte einheitlich in der Semantik von Mealy-Automaten oder Moore-Automaten erfolgen. Mealy-Automaten geben vor, dass die Aktionen beim Zustandsübergang erfolgen, in Moore-Automaten werden die Aktionen im Zustand selbst durchgeführt. Die beiden Automaten-Interpretationen sind äquivalent. Jeder Moore-Automat kann durch einen Mealy-Automaten beschrieben werden – und umgekehrt. Die UML erlaubt beide Automatenformen. Die hier vorgestellte Lösung basiert auf der Automaten-Definition von Mealy.

Weiterhin sollte beim Entwurf der Zustandsautomaten die „Run to Completion“-Regel beachtet werden. Sie besagt, dass während eines Zustandswechsels keine Unterbrechung der Abarbeitung im Automaten erfolgen darf. Der Automat behandelt Ereignisse erst, wenn alle mit einem Zustandsübergang verbundenen Aktionen ausgeführt sind und er sich wieder in einem definierten Zustand befindet. Tritt also während einer Transition ein Ereignis auf, das behandelt werden muss, bedeutet das, dass der Quell- und/oder Zielzustand dieser Transition weiter verfeinert werden muss. Der Vorteil: Die Synchronisation der Ereignisse kann getrennt von der Zustandslogik des Automaten implementiert werden.

Für die einfache Darstellung komplexer reaktiver Systeme eignet sich die Modellierung von hierarchischen Zustandsautomaten. Eine für alle Subzustände gültige Ereignisbehandlung kann dann übersicht-

lich in einem Superzustand erfolgen und muss nicht mehrfach in den Subzuständen spezifiziert werden. Enthält ein Zustandsautomat parallele Abläufe oder parallele Zustandsebenen, sollten diese in getrennten Regionen spezifiziert werden. Die Anzahl der Zustände bleibt auf diese Weise überschaubar. Die Guards an den Zustandsübergängen können aus komplexen Ausdrücken zusammengesetzt werden. Da die Reihenfolge für die Abarbeitung dieser Ausdrücke nicht definiert ist, ist darauf zu achten, dass die Abarbeitung der Teilausdrücke das System nicht verändert.

Die Vorteile modellgetriebener Softwareentwicklung lassen sich erst dann vollständig ausschöpfen, wenn die Modell-Transformation und Code-Generierung wiederholbar ist. Modell-Transformatoren und Code-Generatoren können allerdings nur für die Generierung der Zustandslogik eingesetzt werden, die Implementierung der Fachlichkeit liegt beim Entwickler. Für den Entwurf der Zustandsautomaten bedeutet das: Zustandslogik und fachliche Logik müssen sauber voneinander getrennt werden. Die Zustandslogik, dazu gehören alle Zustände, Zustandsübergänge, die Bedingungen für die Zustandsübergänge und die fachlichen Aktionen, die bei einem Zustandsübergang ausgeführt werden, lässt sich dann jederzeit komplett aus dem Modell generieren. Für die Implementierung der fachlichen Logik empfiehlt es sich, diese aus atomaren Aktionen zusammensetzen, die gut kombiniert werden können. Beim Zustandsübergang können die atomaren Aktionen dann nacheinander ausgeführt werden.

MDD mit dem Quantum Framework

Besonders effizient wird die modellgetriebene Softwareentwicklung, wenn die Code-Generierung aus Zustandsautomaten mit einem Application Framework kombiniert wird. Bei der Transformation können die Implementierungsklassen dann automatisch als anwendungsspezifische Spezialklassen der abstrakten Superklassen des verwendeten Frameworks erzeugt werden. Der stabile Entwurf und die ausgereifte Implementierung des Frameworks stehen dem Projekt so unmittelbar zur Verfügung. Für die Entwicklung von Embedded Real-Time Systemen hat sich das Open-Source-Framework Quantum bewährt.

Quantum ist ein wiederverwendbares, ereignisgesteuertes Application Framework

zur Ausführung paralleler Zustandsautomaten. Mit Quantum realisierte Anwendungen bestehen aus aktiven, multithreading-fähigen Objekten, die gemeinsam die gewünschte Funktionalität zur Verfügung stellen. Aktive Objekte sind zustandsbehaftet und kommunizieren miteinander asynchron, indem sie Ereignisse senden und empfangen. Aktive Objekte enthalten eine eigene Queue zur Serialisierung der Ereignisse. Innerhalb eines aktiven Objekts werden Ereignisse sequentiell nach der Run-to-Completion-Regel abgearbeitet. Quantum bringt alle Funktionen zum Steuern und Kontrollieren der einzelnen Threads, zum Verteilen der Ereignisse sowie zur Serialisierung der Ereignisse in den einzelnen aktiven Objekten mit. Darüber hinaus liefert Quantum die Funktionalität zur Implementierung der zugrunde liegenden Zustandsautomaten der einzelnen Objekte. Die theoretische Grundlage bilden Harel-Automaten. Quantum unterstützt die Implementierung in C/C++ und Java.

Modell-Transformation mit Quantum

Bei der Modell-Transformation und Code-Generierung aus Zustandsautomaten unter Verwendung von Quantum müssen folgende Konventionen des Frameworks berücksichtigt werden: Ereignisse werden in Quantum durch Objekte übermittelt. Die Klassen dieser Objekte werden alle von einer gemeinsamen Superklasse abgeleitet. Die Aktionen, die in Reaktion auf ein entsprechendes Ereignis ausgeführt werden, müssen diese Ereignisobjekte verarbeiten, d.h. übernehmen können.

Werden diese Konventionen erfüllt, können aus Zustandsautomaten entsprechende Design- und Implementierungsstrukturen erzeugt werden. Wird beispielsweise im Zustandsautomaten ein neues Ereignis angelegt, werden für dieses Ereignis automatisch eine entsprechende Subklasse und ein eindeutiger Schlüssel generiert. Ein weiteres Beispiel ist das Anlegen neuer Aktionen: Aus diesen Aktionen werden in der Klasse, die den Zustandsautomaten definiert, Methoden inklusive der benötigten Signatur erzeugt.

Bei der Transformation aus dem Modell wird die Klasse, die den Zustandsautomaten definiert, automatisch aus der von Quantum geforderten Superklasse abgeleitet. Damit bringt sie alle Eigenschaften für ein aktives Objekt mit. Zusätzlich wird



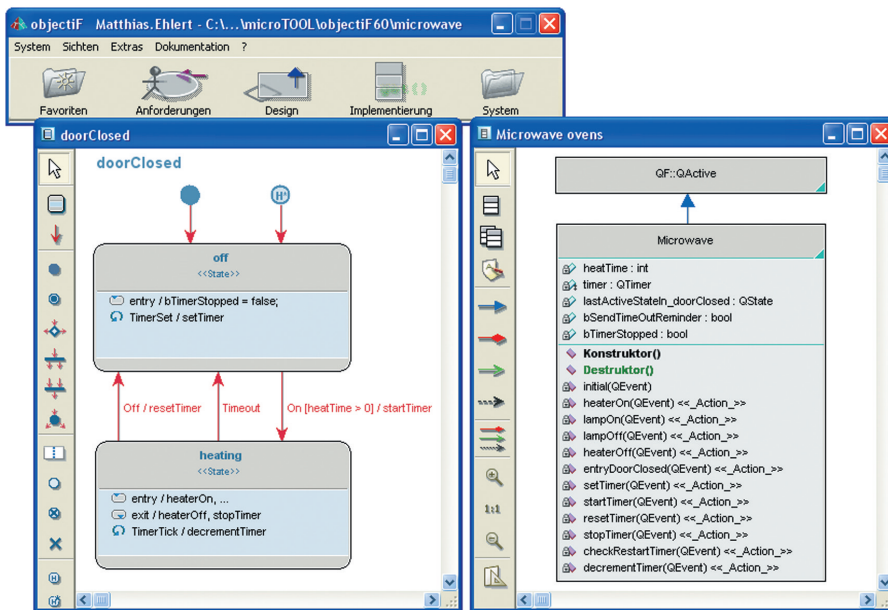


Abbildung 1: Das Zustandsdiagramm zeigt einen Ausschnitt aus dem Zustandsautomaten einer Mikrowelle, das Klassenmodell und das Design der implementierenden Klasse

eine Methode zur Initialisierung des Objekts – Registrierung des Startzustandes und der Ereignisse, die das Objekt verarbeiten kann – automatisch generiert. Ebenso werden für alle Zustände Methoden angelegt und mit einem Stereotyp versehen. Hierbei gilt folgende Regel: Nur Methoden mit dem Stereotyp „State Handler“ werden bei der Transformation aktualisiert und ihr gesamter Inhalt generiert. Methoden, die mit dem Stereotyp „Action“ gekennzeichnet sind, verwalten applikationsspezifischen Code, der in der Verantwortung des Entwicklers liegt und deshalb bei der Transformation unberührt bleibt.

Fazit

Model Driven Development erzwingt eine stabile Architektur mit einer klaren Trennung von fachlicher Logik und Zustandslogik. Für den Entwurf der Architektur bietet sich die UML an. Das Verhalten der zu realisierenden Anwendung lässt sich übersichtlich in Zustandsautomaten modellieren.

Mithilfe von Modell-Transformatoren und Code-Generatoren kann aus der grafischen UML-Spezifikation eine ablauffähige Implementierung nach einem passenden Entwurfsmuster generiert werden. Für die Entwicklung von Embedded Real-Time

Systemen hat sich das Quantum-Framework von Miro Samek bewährt. Voraussetzung für die automatische Transformation und Generierung ist, dass die oben beschriebenen Entwurfsrichtlinien eingehalten werden.

Der Vorteil modellgetriebener Generierung von Code aus UML-Zustandsautomaten für Quantum entsteht vor allem durch die saubere Trennung von Fach- und Zustandslogik. Sie ermöglicht, dass sich der Entwickler vollständig auf die fachliche Modellierung seines Systems konzentrieren kann. Die Zustandslogik lässt sich jederzeit komplett aus dem Modell generieren – und ist somit leichter zu überblicken und besser zu warten. Die Applikationslogik wird von der Transformation nicht betroffen. Finden Änderungen an der Applikationslogik, also beispielsweise beim Testen außerhalb des Modells, statt, kann der Code anschließend ohne Auswirkungen auf die Zustandslogik ins Modell zurückgeführt werden.

Die Anwendung von Modell-Transformatoren und Code-Generatoren bietet darüber hinaus die Möglichkeit, jederzeit im Projektverlauf, das Entwurfsmuster und die Implementierung an neue Zielplattformen anzupassen. Durch die Anwendung von UML in Verbindung mit Code-Generatoren wird die Entwicklung zustandsbehaltender Systeme robuster, flexibler und effizienter. ■