Erweiterbare Programmiersprachen und DSLs

Markus Voelter, Freiberufler/itemis
Bernhard Merkle, SICK AG

Dieser Vortrag (und dieses Paper) beschreiben einen neuartigen Ansatz für die Entwicklung eingebetteter Systeme. Die Idee ist, die Programmiersprache C inkrementell so zu erweitern, dass sie besser für die Entwicklung eingebetteter Systeme in einer bestimmten Domäne geeignet ist. Technologisch wird dies durch die Verwendung von projizierenden Editoren - konkret: JetBrains MPS - umgesetzt, die es erlauben, Sprachen zu erweitern, und zu modularisieren.

Der Status Quo

Die Entwicklung eingebetteter Systeme lässt sich - vereinfacht gesagt - in zwei Kategorien unterteilen: entweder werden die Systeme auf niedriger Abstraktionsebene mit C und Assembler entwickelt, oder es werden Modellierungswerkzeuge eingesetzt. Die Beschränkung auf C führt zu Problemen dort keine besonders mächtigen Mechanismen für Abstraktion und Modularisierung vorhanden sind. Der Code ist daher oft recht "trickreich" implementiert und folglich schlecht wart- bzw. wiederverwendbar. C umfasst außerdem einige Features die i.A. als gefährlich eingeschätzt werden; diese werden dann oft "verboten" und entsprechende Codechecker entdecken Verletzungen dieser Vorschriften.

Modellierungswerkzeuge adressieren einige dieser Probleme, bringen aber eigene Herausforderungen mit: sie sind oft nicht an die eigenen Bedürfnisse anpassbar, integrieren sich schlecht in den Entwicklungsprozess bzw. die Toollandschaft und sind auch aus Benutzersicht und Gesamtproduktivität suboptimal.

Natürlich werden auch Mischformen verwendet, also manuell geschriebener C Code in Kombination mit Modellierungswerkzeuge. Allerdings führt auch dies zu massiven Herausforderungen bzgl. der semantischen und technischen Integration zwischen den beiden Welten.

Erweiterbare Sprachen als Alternative

Wir möchten diesem Ansatz eine Alternative gegenüberstellen: die inkrementelle, domänenspezifische Erweiterung von C. Die Erweiterungen können dabei zunächst allgemeingültig sein (Zustandsmaschinen, Tasks, Komponenten) oder auch bei Bedarf auch sehr spezifisch an bestimmte Problemdomänen, Projekte oder Plattformen angepasst. Die einzelnen Erweiterungen sind modular, lassen sich aber auch kombinieren um eine für ein spezifisches Projekt genau passende Sprache zu erhalten. Aufgrund der engen semantischen und tooltechnischen Integration verspricht dies deutlich geringere Reibungsverluste als der traditionelle Ansatz.

Die Spracherweiterungen werden durch Transformation/Codegenerierung auf C abgebildet und nachfolgend von einem Ggf. Target-spezifischen Compiler übersetzt.

Projizierende Editoren und MPS

Traditionelle Texteditoren basieren auf Grammatiken und Parsern. Eine Grammatik beschreibt, wie eine Zeichenkette strukturiert sein muss, dass sie einer bestimmten Sprache entspricht. Die Aufgabe des Parsers ist zu validieren, ob ein Eingabestring der Grammatik entspricht und - üblicherweise – nachfolgend eine Datenstruktur zu erzeugen, die die relevanten semantischen Informationen des Text-Programms so repräsentiert, dass sie leicht verarbeitet werden können (Type Checkers, Generatoren). Diese Datenstruktur wird als Abstract Syntax Tree (oder Graph) bezeichnet.

Projizierende Editoren verwenden einen grundlegend anderen Ansatz, Grammatiken und Parser kommen hier nicht zum Einsatz. Stattdessen ist der Editor für die Sprache so implementiert, dass der AST direkt modifiziert wird. Jede Benutzeraktion führt *direkt* zu einer Änderung der Datenstruktur. Diese Verhalten wird bei grafischen Modellierungswerkzeugen schon lange genutzt: wird eine Klasse in ein Diagramm eingefügt, so erzeugt das Werkzeug im dahinterliegenden Modell direkt eine Instanz des Konzeptes "Klasse". Das Diagram wird nicht geparst! Projizierende Editoren verwenden diesen Ansatz auch für Sprachen mit textueller Syntax. Durch entsprechende "Tricks" wird dafür gesorgt, dass sich der projizierte Text (fast) genauso editiert wie die gewohnten Zeichenketten.

JetBrains MPS [1] ist ein Open Source Werkzeug das diesen Ansatz unterstützt. Zusammen mit den kommerziellen Werkzeug von Intentional Software [5] sind diese Werkzeuge die ersten projizierenden Editoren die man in der Praxis tatsächlich einsetzen kann.

Die Vorteile dieses Ansatzes liegen darin, dass keine Grammatiken und Parser verwendet werden: textuelle, tabellarische, grafische und andere Notationen lassen sich somit problemlos im selben "Programm" kombinieren. Ein Konzept lässt sich - je nach Kontext oder Benutzerpräferenz - anders darstellen. Auch die Modularisierung und spätere Kombination von Sprachmodulen lässt sich erheblich leichter realisieren als mit Parsertechnologie.

Die Modular Embedded Language

Auf http://mbeddr.com findet sich die Modular Embedded Language (MEL). Dies ist unser Prototyp einer Sammlung von C Erweiterungen für die Embedded Entwicklung. Die konkreten Sprachfeatures werden im Folgenden beschrieben. Zur Validierung des Ansatzes haben wir mit Lego Mindstorms, C und Osek die Steuerung für einen Roboter implementiert.

Die Core Sprache ist eine Implementierung der C Sprachfeatures; allerdings verwenden wir keine Header Files, sondern ein Modulkonzept (Header werden dann später als Compilerinput generiert). Modules entsprechen Namensräumen und können

alle anderen Abstraktionen enthalten. Sie definieren Abhängigkeiten untereinander und markieren mittels *exported* die füe andere Module sichtbaren Elemente.

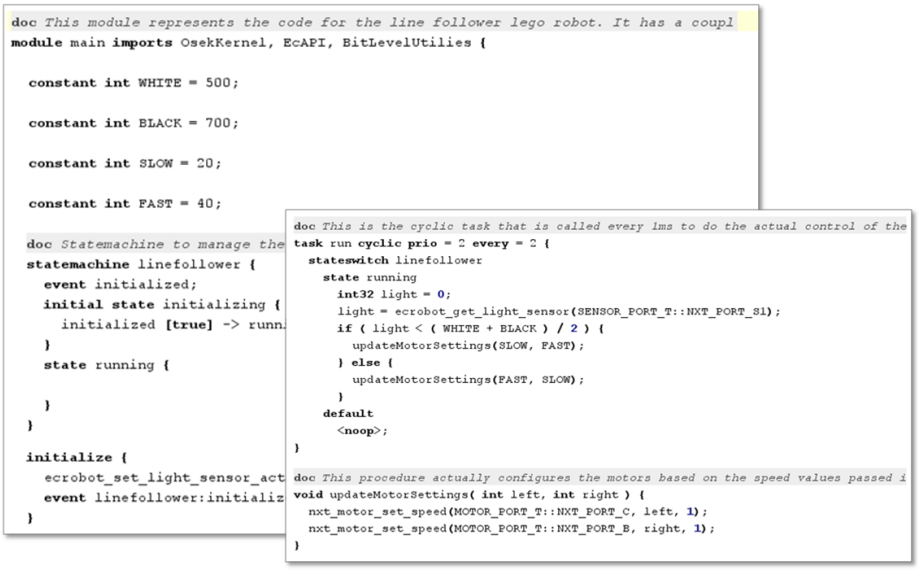

Abb 1: Der Code für einen einfachen Linefollower als MEL programm

Abb. 1 zeigt ein einfaches Linefollower-Programm. Das Module *main* enthält Konstanten und Funktionen, aber auch bereits eine Zustandsmaschine (siehe unten). Diese verwaltet die verschiedenen Phasen im Programmablauf. Das Modul definiert auch einen Task für das periodische Abfragen der Sensoren. Sein Verhalten ist abhängig von der Phase im Programmablauf. Ein sogenanntes *stateswitch* implementiert die Abhängkeit von der weiter oben definierten Zustandsmaschine.

Tasks kapseln Verhalten das zu bestimmten Zeiten (beim Programmstart, zyklisch) abgearbeitet werden soll. Tasks erinnern an Funktionen, haben aber keinen Rückgabetyp. Durch den Codegenerator werden sie letztendlich auf C Funktionen sowie diverse Einträge im OSEK Konfigurationsfile (OIL) abgebildet.

Ein weiteres Sprachmodul unterstützt die Deklaration von **Zustandsmaschinen**, das Feuern von Events sowie die Definition von zustandsabhängigem Verhalten (das *stateswitch* von oben).

Zustandsmaschinen werden durch Transformationen abgebildet auf geschachtelte Switch/Case Statements, die im Rahmen des Core Moduls verfügbar sind. Das Zustandsmaschinenmodul erweitert das Core Modul und Zustandsmaschinen implementieren *IModuleContent*, sodass sie innerhalb von Modulen deklariert werden können.

linefollower	initializing	paused	running	crash
initialized	true running			
bumped			true crash	
blocked			true paused	
unblocked		true running	true crash	

Abb. 2: Editieren von Zustandsmaschinen als Tabelle

Wie oben beschrieben, lassen sich mit Projizierenden Editoren sehr einfach verschiedene Notationen für dasselbe Konzept realisieren. Abb. 2 zeigt eine Zustandsmachine als Tabelle. Weitere Darstellungmöglichkeiten z.B eine grafische Notation sind bereits geplant.

Feingranulare Erweiterungen, wie bspw. die **Erweiterungen des Typsystems** sind auch auf möglich. Beispielsweise können neue Datentypen eingeführt werden. Das Beispiel hier zeigt Variablen für die automatische Berechnung des Durchschnitts. In Abb. 3 A deklarieren wir einen *avg(int,10)*. Dies bedeutet, dass wenn wir einer Variable dieses Typs mit dem =/ Operator einen Wert zuweisen (Abb. 3 B), enthält die Variable immer den Durchschnitt der letzten 10 Zuweisungen (Abb. 3 C). Dies ist beispielsweise für sehr empfindliche Sensoren wichtig. Ein anderes implementiertes Beispiel sind Variablen die bei Zuweisung automatisch eine Überlaufsprüfung durchführen (zur Laufzeit).

A
```
{sonar} var avg(int, 10) currentSonar = 250;
```

B
```
{sonar} task sonartask cyclic prio = 2 every = 100 {
          currentSonar =/ ecrobot_get_sonar_sensor(SENSOR_PORT_T::NXT_PORT_S2);
          {debugOutput} debugInt(2, "sonar:", currentSonar);
        }
```

C
```
{sonar} if ( currentSonar < 150 ) {
  event linefollower:blocked
  terminate;
}
```

Abb. 3: Durchschnittsbildende Variablen

Komponenten sind ein gerne verwendetes Strukturierungsmittel. Auch hierfür bietet C kein direktes Sprachkonstrukt. MEL bietet hier Support, wie Abb. 4 zeigt. Eine Schnittstelle (Menge von Operationen), eine Komponente (bietet und nutzt Schnittstellen) und eine Komponentenimplementierung (fügt das Verhalten zu einer Komponente hinzu). MEL Komponenten unterstützen "statischen Polymorphismus". Der Client-Code hat nur Abhängigkeiten zu den Schnittstellen und Komponenten können jederzeit ausgetauscht werden (durch ein Mapping von Schnittstelle auf Komponente). Transformationen bilden dies auf normale C Funktionsaufrufe ab - ein Laufzeit-overhead ist daher nicht vorhanden.

```
exported interface MotorControl {
  void stop(   );
  void setLeftSpeed( int8 speed );
  void setRightSpeed( int8 speed );
}

exported component Motors {
  provides motorControl : MotorControl;
}
```

```
exported component implementation MotorsNXT : Motors {

  procedure void motorControl.stop( ) {
    nxt_motor_set_speed(MOTOR_PORT_T::NXT_PORT_B, 0, 1);
    nxt_motor_set_speed(MOTOR_PORT_T::NXT_PORT_C, 0, 1);
  }

  procedure void motorControl.setLeftSpeed( int8 speed ) {
    nxt_motor_set_speed(MOTOR_PORT_T::NXT_PORT_C, speed, 1);
  }

  procedure void motorControl.setRightSpeed( int8 speed ) {
    nxt_motor_set_speed(MOTOR_PORT_T::NXT_PORT_B, speed, 1);
  }
}
```

Abb. 4: Eine Schnittstelle, eine Komponente und eine Komponentenimplementierung

Alle bisher beschriebenen C-Erweiterungen sind allgemeingültig. Im Folgenden zeigen wir die Integration einer **Domänenspezifischen Sprache (DSL)**, die einen sehr begrenzten Fokus hat, dafür aber Sachverhalte in diesem Fokus sehr knapp, präzise und verständlich ausdrücken kann. Als Beispiel verwenden wir die Steuerung des Roboters (Abb. 5). Man beachte, dass auch Roboter-Routing Scripte in Module eingebettet sind und bspw. Einfache C Funktionen aufrufen können. Damit entfällt der Unterschied - und der Integrationsaufwand - zwischen Programmierung (C Code) und Modellierung (DSLs, aber auch Zustandsmaschinen, Komponenten) komplett. Der DSL Code wird per Transformation automatisch auf die vorhandenen C-Erweiterungen (hier Zustandsmaschinen und Tasks) abgebildet.

```
module impl imports <<imports>> {

  int speed( int val ) {
    return 2 * val;
  }

  robot script stopAndGo
    block main on bump block retreat on bump <no bumpReaction>
                          stop
                          accelerate to 0 - 30  within 2000
                          drive on for 2000
                          decelerate to 0  within 1000
                          stop
      accelerate to speed(25) within 3000
                                                    }
                                                          drive on for 2000
                                                          turn left for 2000
                                                          block driveMore on bump <no bumpRe
                                                            accelerate to 80  within 2000
                                                            turn right for 3000
                                                          decelerate to 0  within 3000
                                                          stop
```

Figure 5: A robot routing script embedded in a module

Anforderungsverfolgung und Produktlinienvariabilität werden ebenfalls unterstützt, können hier aber aus Platzgründen nicht weiter beleuchtet werden.

Zusammenfassung

Der hier beschriebene Ansatz führt zu kompaktem, übersichtlichen, lesbaren und semantisch reichen und damit gut analysierbaren Code. Die Integrationsproblematik zwischen Modellen und Code fällt weg, alles passiert in einem integrierten Werkzeug. Unter http://mbeddr.com gibt es eine Reihe von Screencasts und weiteren Papieren [2,3,4]. Die Webseiten enthält auch Links zu MPS sowie dem MEL Code.

Referenzen

1. JetBrains, *Meta Programming System*, http://jetbrains.com/mps
2. Voelter, M., *Product Line Engineering with Projectional Language Workbenches*, http://voelter.de/data/pub/Voelter-ProductLineEngWithProjectionalLanguageWorkbenches.pdf
3. Voelter, M., Solomatov K., *Language Modularization and Composition with Language Workbenches*, http://voelter.de/data/pub/VoelterSolomatov_SLE2010_Language ModularizationAndCompositionLWBs.pdf
4. Voelter, M., *Embedded Software Development with Projectional Language Workbenches*, Proceedings of MODELS 2010, can also be found at mbeddr.com
5. Intentional Software, http://intentsoft.com