Zimmermann, Frank

Working Paper

Modellgetriebene Softwareentwicklung für Lego NXT

Arbeitspapiere der Nordakademie, No. 2008-01

Provided in cooperation with:
Nordakademie - Hochschule der Wirtschaft

Arbeitspapiere der Nordakademie

ISSN 1860-0360

Nr. 2008-01

Modellgetriebene Softwareentwicklung für Lego Mindstorms NXT

Prof. Dr. Frank Zimmermann

Februar 2008

Eine elektronische Version dieses Arbeitspapiers ist verfügbar unter:
http://www.nordakademie.de/index.php?id=ap

NORDAKADEMIE
HOCHSCHULE DER WIRTSCHAFT

Köllner Chaussee 11
25337 Elmshorn
http://www.nordakademie.de

Modellgetriebene Softwareentwicklung für Lego NXT*

Frank Zimmermann

20.2.2008

Zusammenfassung

Model Driven Software Development (MDSD) soll die Hemmschwelle für die Anwendung von komplexen Frameworks reduzieren und den Weg für fachliche Modellierung frei machen. Aus Modellen werden dabei fehlerfreie, architekturkonforme Programme generiert. Die verwendete Generatortechnologie jedoch stellt häufig aufgrund komplizierter Buildvorgänge selbst ein Lernhindernis dar, das die Nutzerakzeptanz modellgetriebener Entwicklungsumgebungen reduziert, in einer Erprobungsphase vielleicht sogar verhindert. In dieser Arbeit wird der Aufbau einer vollintegrierten Entwicklungsumgebung für Lego Mindstorms Roboter beschreiben. Die Benutzergruppe der Lego Robotikprogrammierer stellt aufgrund ihrer intrinsischen Motivation besonders hohe Anforderungen an die Benutzerfreundlichkeit. Ein Vergleich mit der Lego eigenen Sprache NXT-G soll die Vorteile der modellgetriebenen Entwicklung zeigen.

1 Lego Mindstorms NXT

Der Lego Mindstorms NXT [10] ist 2006 als Nachfolger von Lego Mindstorms RCX der Öffentlichkeit vorgestellt worden. Durch seinen Preis unter 300 € ist der NXT im Hobbybereich gerne genutzt, wird aber auch von Hochschulen in der Lehre eingesetzt. Die NXT Generation verfügt über 256 KB flash memory und 64 KB RAM. Verglichen mit aktuellen Handys ist das ein sehr kleiner Speicher. Im Basispaket sind 3 Motoren und 4 Sensoren enthalten.

Lego Mindstorms NXT werden mit einem eigenen Betriebssystem ausgeliefert. Programme werden in der vom MIT mitentwickelten Programmiersprache NXT-G geschrieben. Die Programmierumgebung erlaubt die visuelle Kombination und Konfiguration von Programmblöcken. Erweiterungen sind in Form von angepassten Programmblöcken im Internet verfügbar oder mit der kommerziellen Umgebung LabView selbst programmierbar.

Abbildung 1: NXT Line Follower

Als Beispiel für eine NXT-G Anwendung soll hier der Line Follower (siehe Abbildung 1) genannt werden. Ein mobiler Robot soll einer schwarzen Linie folgen (siehe Abbildung 2). Dazu nutzt er einen

*Dieses Dokument wird mit dem Koautor Peter Friese als Beitrag beim Eclipse Magazin eingereicht.

Lichtsensor, der die Helligkeit unter dem Sensor misst. Ist der Helligkeitswert niedrig, so befindet sich der Robot über der Linie, und der Robot kann vorwärts fahren. Ist der Helligkeitswert hoch, so ist der Robot nicht über der Linie. Allerdings weiss man nicht, ob man rechts oder links von der Linie ist, so dass man sich mit sich aufschaukelnden Rechts-/Linksbewegungen solange um die eigene Achse dreht, bis man die Linie wieder gefunden hat. Abbildung 2 zeigt das Programm in NXT-G. Man erkennt die Verarbeitungsblöcke, die ähnlich zu Makroaufrufen über Datenflüsse miteinander verbunden sind. Man kann mit NXT-G Unterprogramme erstellen, allerdings sind der Verwendung von Parametern Grenzen gesetzt, wie man an dem Links/Rechts Sweep Block erkennt. Der Abstraktionsgrad der NXT-G Sprache wird anhand des Double Arc Blocks deutlich, der die Verdopplung des Werts einer Variablen enthält.

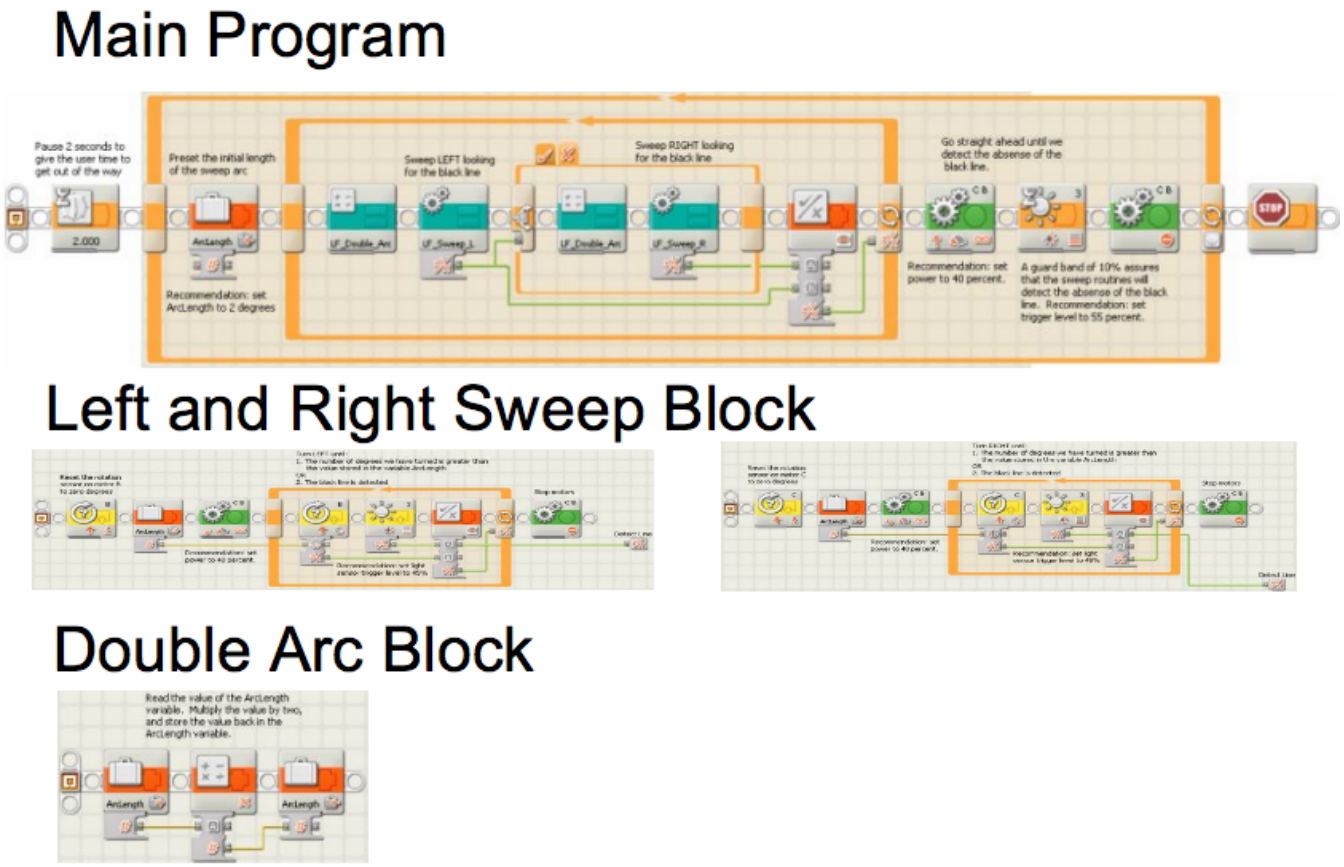

Abbildung 2: Line Follower mit NXT-G [5]

2 leJOS Java Operating System

Neben der Lego eigenen Umgebung gibt es noch zahlreiche Ansätze, Lego NXT mit herkömmlichen Programmiersprachen C, C#, Phyton,... den NXT zu programmieren. Für die Programmiersprache Java gibt es das Betriebssystem leJOS [6], das eine virtuelle Java Maschine auf dem NXT samt zugehöriger Infrastruktur wie zum Beispiel angepasste Klassenbibliotheken und native Methoden zum Ansteuern der Hardware umfasst. Der prominenteste Einsatz von leJOS ist wahrscheinlich der Lego Roboter Jitter, der 2001 auf der ISS die Erde umrundete und in der Lage war, Gegenstände in der Schwerelosigkeit einzusammeln.

Als typsichere objektorientierte Sprache eignet sich Java besonders gut für die Programmierung von autonomen Robotern, denn das Fehlen einer geeigneten Testinfrastruktur kann zum Teil durch die starke Typisierung kompensiert werden. Andererseits stellt Java als objektorientierte Programmiersprache umfangreiche Mechanismen zur Entwicklung komplexer Anwendungen zur Verfügung. Diese

Eignung ist nicht zufällig, sondern entspringt der ursprünglichen Intention, Java als Programmiersprache für eingebettete Systeme zu nutzen.

Aufgrund der geringen Speichergröße ist zu Zeit unter leJOS nur eine eingeschränkte Klassenbibliothek verfügbar. Weil keine Collection Klassen vorhanden sind, muss man mit Arrays vorlieb nehmen. Auch gibt es in leJOS das Java Konzept der Classloader nicht, also Klassen, die dynamisch Klassen nachladen können. Statt dessen müssen die .class Dateien durch einen statischen Link Prozess in eine .nxj Datei gebunden werden.

3 Zustandsautomaten

In der Entwicklung von embedded Systems haben sich Zustandsautomaten als besonders geeignete Technik erwiesen [Kapitel 13 aus 8]. In der Informatikausbildung stellen Automatenmodelle die Grundlage für viele Anwendungen dar, von der theoretischen Informatik bis hin zur Spezifikation von Kommunikationsprotokollen in der Nachrichtentechnik. Automatenmodelle bestechen durch ihre intuitive Verständlichkeit und durch ihre Fähigkeit, komplexe Probleme zu strukturieren. In der Umsetzung von Automaten jedoch zeigt sich schnell, dass die zweidimensionale graphenbasierte Struktur von Automaten nur unter deutlichem Verlust an Ausdruckskraft in die eindimensionale Struktur von Programmtext übertragen werden kann. Um das Automatenmodell für die Programmierung von Lego Mindstorms sinnvoll nutzen zu können, ist eine Entwicklungsumgebung wünschenswert, die die Automatenmodelle direkt in Programmcode übersetzt. Im Rahmen von modellgetriebener Softwareentwicklung werden in der Regel nur die architekturzentrierten Bestandteile generiert [9]. Aufgrund der geringen Komplexität der Lego Anwendungen kann man in Betracht ziehen, dass die gesamte Anwendung aus dem Modell generiert wird. Dazu müssen die modellierten Artefakte um Java Codeschnipsel (Snippets) ergänzt werden. In diesen Snippets werden Variablen deklariert, die Aktoren und Sensoren abgefragt oder eigene Klassen eingebunden. Letzteres stellt sicher, dass komplexe Codeteile in die Anwendung mit einbezogen werden können, ohne das Modell unnötig aufzublähen. Für das EMF Modell sind die Snippet nur Texte, erst ein Generator kopiert sie an die richtige Stelle in einer Java Klasse. Fehler werden so zwar erst bei der Generierung erkannt, da die Generierung jedoch nahtlos in den Modellierungsprozess eingebaut ist, erscheint dieses Vorgehen völlig ausreichend.

Der Vorteil der höheren Abstraktionsstufe wird anhand der Lösung des Line Follower Problems deutlich. Abbildung 3 zeigt die Version der Robotersteuerung mit Zustandsautomaten. Der Zustandsautomat ist im wesentlichen selbsterklärend, das Modell dient gleichzeitig zur Dokumentation und Ausführung. Die verwendeten Snippets sind einerseits Java Anweisungen wie $pilot.forward()$ zum Vorwärtsfahren oder $pilot.stop()$ zum Stoppen beider Motoren und andererseits Bedingungen wie $light.readValue() > 40$ zum überprüfen der vom Lichtsensor gemessenen Helligkeit.

4 MDSD Werkzeugkette

Zur Entwicklung eines Development Toolkits mit MDSD kann man auf eine Reihe von Open Source Werkzeugen zurückgreifen. Auf die Evaluation verschiedener Alternativen soll hier nicht eingegangen werden. Zur Speicherung der Modelle dient das Eclipse Modeling Framework [1]. Statt auf die naheliegende Möglichkeit der Nutzung der UML Statecharts zurückzugreifen, wurde eine Domänenspezifische Sprache entwickelt. Das reduziert den Aufwand für die Generierung erheblich, denn das Metamodell der UML ist für die Generierung von NXT Anwendungen zu komplex. Abbildung 4 zeigt das domänenspezifische Metamodell. Für die Erstellung der Modelle wird ein GMF [2] basierter graphischer Editor benutzt. oAW [7] stellt mit xPand und check geeignete Werkzeuge zur Validierung und Generierung von Java Anwendungen aus EMF Modellen.

5 leJOS Statemachine Develoment Toolkit

Für die Entwicklung von Statemachine basierten Robots soll es ausreichen, ein Zustandsdiagramm wie in Abbildung 3 zu zeichnen und zu speichern. Das Ausführen eines modellierten Zustandsautomaten für den Mindstorms Brick erfolgt in fünf Schritten

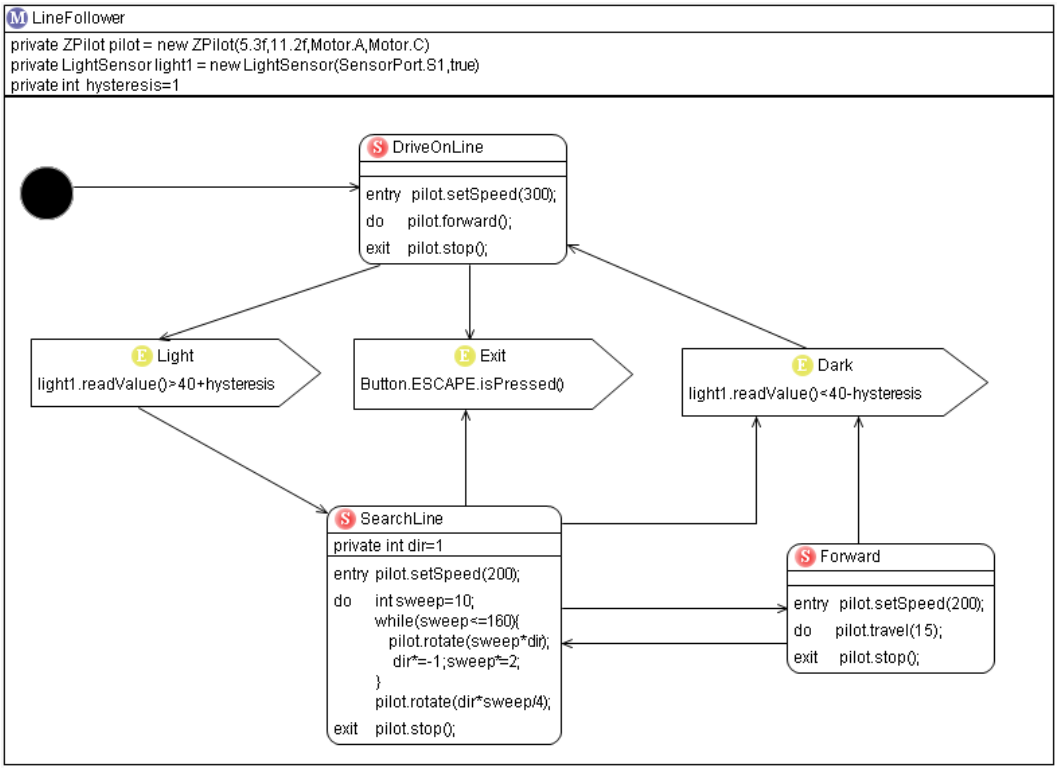

Abbildung 3: Line Follower mit Statemachines

1. Generieren des Java Programms (.java Datei) aus dem Modell
2. Übersetzen des Java Programms (.class Datei)
3. Binden des Java Programms (.nxj)
4. Upload der .nxj Datei auf den Brick
5. Starten des Programms

Punkt 1 bis 3 können so weit automatisiert werden, dass der leJOS Statemachine Entwickler nichts mehr davon mitbekommt. Punkt 4 und 5 müssen explizit vom Entwickler angestoßen werden. Für ein Eclipse Plugin bedeutet das also, dass 1 bis 3 durch einen Builder und 4 und 5 durch Actions oder Commands ausgelöst werden. Abbildung 5 gibt einen Eindruck von der Anwendung.

6 Nahtlose Integration durch Builder

Incremental Builder sind eine bekannte Möglichkeit [3], automatisierbare Buildprozesse in die Eclipse Umgebung nahtlos zu integrieren. Die Auswahl der Builder, die für ein gegebenes Projekt verwendet werden, wird in Eclipse über die Natures festgelegt. Projekte ihrerseits können mehrere Natures haben und damit einen komplexen Buildprozess abdecken.

Im gegebenen Problem werden korrespondierend zu den Schritten 1 bis 3 drei Builder benötigt. Während zur Umsetzung von Schritt 2 der JavaBuilder mit der JavaNature assoziiert ist, werden Schritt 1 und 3 mit einer neuen LeJOSNature verknüpft. Das Erstellen einer neuen Nature mit einem neuen Builder ist durch die vorgefertigten Templates des New Plug-in Project Wizards sehr einfach. In dem Template „Plug-in with an inremental project builder“ist eine SampleNature, ein neuer SampleBuilder und eine SampleAction zum Aktvieren der Nature enthalten. Die erforderlichen Klassen

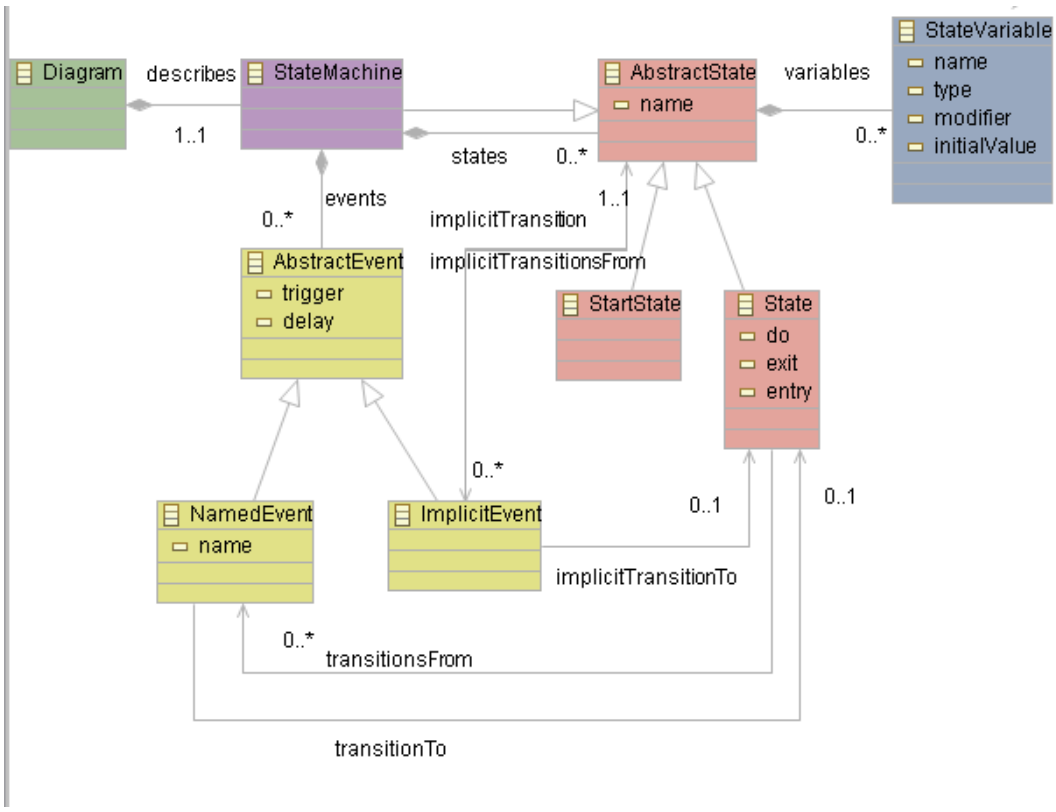

Abbildung 4: Metamodell NXT Zustandsautomaten

und plugin.xml Einträge werden generiert und können leicht an die eigene Situation angepasst werden. Listing 1 zeigt die Deklaration der Nature in der plugin.xml.Die Builder werden in Zeile 6 und 7 definiert.

Listing 1: Eclipse Nature Deklaration

```
1  <extension id="lejosnature" name="leJOS Nature"
2        point="org.eclipse.core.resources.natures">
3  <runtime>
4     <run class="de.nordakademie.lejos.core.nature.LeJOSNature"/>
5  </runtime>
6     <builder id="de.nordakademie.lejos.core.lejosbuilder"/>
7     <builder id="de.nordakademie.lejos.core.lejoslinker"/>
8     <requires-nature id="org.eclipse.jdt.core.javanature"/>
9   </extension>
```

Beim Konfigurieren der leJOS Nature sind nun drei Dinge zu erledigen. Zum ersten sind die erforderlichen Bibliotheken in den Classpath des Projekts zu bringen. Unter anderem ist das Java Runtime Environments durch das NXT spezifische Runtime Enironment zu ersetzen. Zweitens müssen Verzeichnisse für die Modelle und für die generierten Klassen angelegt werden. Und schließlich müssen die Builder in der richtigen Reihenfolge hinzugefügt werden. Listing 2 zeigt die configure Methode aus der LeJOSNature Klasse, in dem deutlich wird, wie man Builder hinzufügen kann.

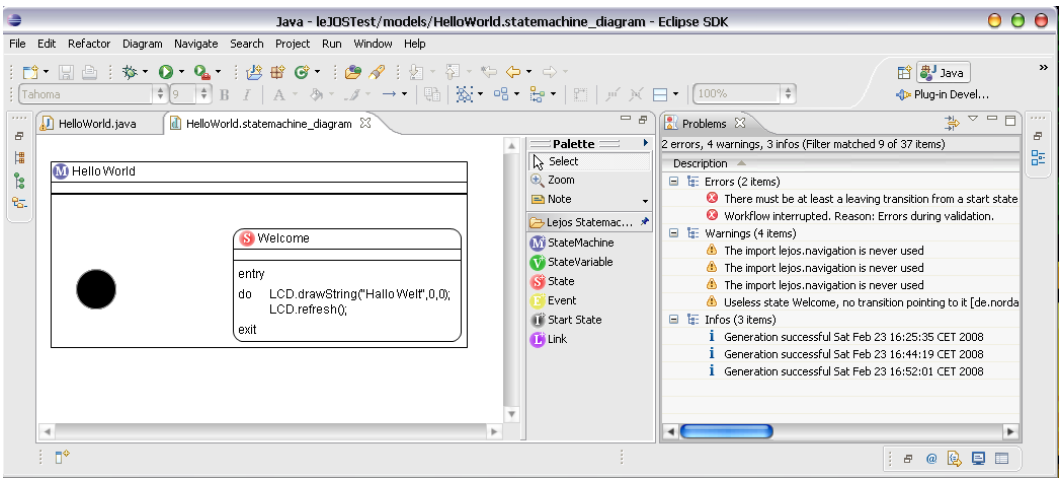

Abbildung 5: Integrierte modellbasierte Entwicklung. Die vollständige Anwendung wird im Diagramm erstellt. Domänenspezifische Fehler sowie Fehler im generierten Code werden beim Abspeichern im Problems view dargestellt. Für Upload und Starten der Anwendung steht ein Kontextmenü zur Verfügung.

Listing 2: Eclipse Nature Deklaration

```
1  public void configure() throws CoreException {
2          prepareClassPath();
3          createDirectories();
4          addBuilders();
5  }
6
7  private void addBuilders() throws CoreException {
8          IProjectDescription desc = project.getDescription();
9          ICommand[] commands = desc.getBuildSpec();
10         ICommand[] newCommands = new ICommand[commands.length + 2];
11         System.arraycopy(commands, 0, newCommands, 1, commands.length);
12         // Insert generator as first
13         ICommand command = desc.newCommand();
14         command.setBuilderName(LejosBuilder.BUILDER_ID);
15         newCommands[0] = command;
16         // Insert linker as last
17         command = desc.newCommand();
18         command.setBuilderName(LejosLinker.BUILDER_ID);
19         newCommands[newCommands.length - 1] = command;
20         desc.setBuildSpec(newCommands);
21         project.setDescription(desc, null);
22 }
```

Nun geht es an das Anpassen des generierten LeJOSBuilders. Wenn die Datei mit dem Zustandsautomaten neu angelegt oder geändert wurde, dann ist die Generierung mit o18e [1] anzustoßen. Der Builder filtert die Dateien mit der Endung .statemachine, mit der die Modelldateien des EMF Models enden. Das Starten des Workflows (Zeile 16 Listing 3) erfordert Zugriff auf die aktuelle Resource (Zeile 7) und das Setzen des Arbeitsverzeichnisses (Zeile 8).

[1] o18e steht als Abkürzung für openarchitectureware.

Listing 3: Starten eines oAW Workflows aus einem Incremental Builder

```
1  private void runWorkflow(IResource resource) throws CoreExeption{
2          Map slotMap = new HashMap();
3          Map properties = new HashMap();
4
5          // configure properties passed to the workflow engine
6          properties.put("basedir", getProject().getLocation().toOSString());
7          properties.put("model", resource.getLocation().toOSString());
8
9          // handling error messages
10         prepareLogger(resource);
11         WorkflowRunner runner = new WorkflowRunner();
12         runner.run("workflow/generator.oaw",
13                 new NullProgressMonitor(), properties, slotMap);
14         resetLogger();
15 }
```

Der Workflow aus Listing 4 Zeile 11 enthält nun einerseits den Aufruf der nötigen Check Anweisungen, um die Konsistenz der Modelle zu gewährleisten. Modelchecks, die Fehler schon in der Konzeptionsphase erkennen lassen, sind ein oft zitierter Vorteil modellgetriebener Entwicklung mit Domänenspezifischen Sprachen [9]. Andererseits wird im Falle einer fehlerfreien Überprüfung des Modells Java Code erzeugt.

Listing 4: oAW Workflow

```
1  <workflow>
2          <property name="metaModel" value="lejos.stateMachine.StateMachinePackage" />
3
4          <!-- load model and store it in slot 'model' -->
5          <component class="org.eclipse.mwe.emf.Reader">
6                  <uri value="file://${model}" />
7                  <modelSlot value="model" />
8          </component>
9
10         <!-- semantical checks -->
11         <component class="org.o18e.check.CheckComponent">
12                 <metaModel id="mm"
13                         class="org.o18e.type.emf.EmfMetaModel">
14                         <metaModelPackage value="${metaModel}" />
15                 </metaModel>
16                 <checkFile value="check::statemachine" />
17                 <expression value="model.eAllContents" />
18         </component>
19
20         <!--  generate code -->
21         <component class="org.o18e.xpand2.Generator">
22                 <metaModel id="mm"
23                         class="org.o18e.type.emf.EmfMetaModel">
24                         <metaModelPackage value="${metaModel}" />
25                 </metaModel>
26
27                 <expand value="template::statemachine::main FOR model" />
28                 <outlet path="${basedir}/ssrc-gen"/>
29         </component>
30 </workflow>
```

Besondere Aufmerksamkeit in Listing 3 ist der Fehlerbehandlung in Zeile 10 und 14 zu widmen. In einem oAW Check File und während der Generierung könnten Fehler auftauchen, die dem Entwick-

ler mitgeteilt werden müssen. oAW nutzt Logger, um die Anwender des Workflows vom Fortschritt der Arbeit zu informieren. Hier sollten Fehlerinformationen als Marker im Problems View angezeigt werden, ein Entwickler sie dort erwarten würde. Um dies zu erreichen muss ein angepasster Logger geschrieben werden, der die Fehler und Warnmeldungen als Filemarker hinterlegt. Listing 5 zeigt eine Möglichkeit. Dabei wird ein Handler mittels einer anonymen inneren Klasse (Zeile 3) erzeugt, der dem vom oAW verwendeten java.util.logging Logger hinzugefügt wird. Der Handler schreibt nun die Warnungen und Fehlermeldungen als Marker zum aktuell bearbeiteten File(Zeile 8 und 10). Dazu steht die vom Template generierte Methode addMarker zu Verfügung. Da unterschiedliche File nacheinander generiert werden können, kann dieser Handler zur Laufzeit nicht wiederverwendet werden. Die Methode resetLogger löst den Handler wieder vom Logger.

Listing 5: Einrichten der Fehlerbehandlung für einen oAW Workflow

```
1  \samepage
2  private void prepareLogger(final IResource resource) {
3    Logger l = Logger.getLogger("org.o18e.workflow.WorkflowRunner");
4    h = new Handler() {
5      public void close() throws SecurityException {}
6
7      public void publish(LogRecord record) {
8        if (record.getLevel() == Level.SEVERE) {
9          addMarker((IFile) resource, record.getMessage(),1,IMarker.SEVERITY_ERROR);
10       } else if (record.getLevel() == Level.WARNING) {
11         addMarker((IFile) resource,record.getMessage(),1,IMarker.SEVERITY_WARNING);
12       }
13     }
14
15     public void flush() {}
16   };
17   l.addHandler(h);
18 }
```

7 Nutzen von MDSD

Es soll eine Gegenüberstellung der NXT-G und leJOS Statemachine Entwicklungsumgebung gemacht werden, aus denen die unterschiedlichen Ansätze der beiden visuellen Entwicklungsumgebungen deutlich werden. Gemessen an den Vorteilen, die man von MDSD [9] erwartet, werden die beiden visuellen Umgebungen eingeschätzt.

Nutzen	NXT-G	Statemachine
Geschwindigkeitssteigerung durch Automation	Kaum erkennbar, da der Abstraktionsgrad den einer gewöhnlichen Makrosprache nicht überschreitet.	Vorhanden. Übersichtlichkeit und Konzentration auf die wesentlichen Bestandteile erhöht die Geschwindigkeit beträchtlich.
Mehr Interoperabilität und Funktionalität	Nicht Vorhanden	Nicht vorhanden; Testtreiber könnten generiert werden. Dieses Potential st zur Zeit noch nicht genutzt.
Bessere Softwarequalität	Nicht erreicht.	Erreicht, durch Generierung von 100% architekturkonformem und fehlerfreien Code.
Erhöhung der Wiederverwendbarkeit	Gering durch Makro Technik	Zur Zeit genauso gering wie bei NXT-G. Modularität und Hierarchie ist ein Ansatz, die Harel [4] in die Modellierung von Zustandsautomaten eingebracht hat. Zur Zeit nicht implementiert.
Verbesserung der Wartbarkeit durch Ablegen von Querschnittsfunktionalität in den Transformationsregeln	Nicht relevant	Zu Zeit nicht sichtbar.
Hoher fachlicher Abstraktionsgrad; Explizierung des fachlichen Expertenwissens	Abstraktionsgrad nicht erreicht.	Hoher Abstraktionsgrad. Kapselung technischer Schwierigkeiten durch Framework und Generator.
Besere Dokumentation	Dokumentation nur durch zusätzliche Beschreibung möglich.	Diagramme sin selbstdokumentierend.

8 Fazit

Durch das dargestellte Vorgehen lässt sich mit relativ geringem Aufwand eine integrierte Entwicklungsumgebung aufbauen, deren Bedienung so intuitiv ist, dass auch ungeübte Entwickler schnell den Einstieg finden. Trotz der komplexen Vorgänge (Generierung, Compilierung und Binden) können kurze Turnaround Zeiten sichergestellt werden, so dass eine effiziente Entwicklung sichergestellt ist. Erweiterungen, wie zum Beispiel das automatisierte Generieren von Testrahmen für Zustände, sind ohne großen Aufwand integrierbar. Nicht alle Vorteile der Modellgetriebenen Softwareentwicklung können an dem Werkzeug zur Zeit nachvollzogen werden.

Literatur

[1] Eclipse Foundation: *EMF Project Page*. 2 2008. – URL `http://www.eclipse.org/modeling/emf`. – Zugriffsdatum: 25.2.2008

[2] Eclipse Foundation: *GMF Project Page*. Februar 2008. – URL `http://www.eclipse.org/modeling/gmf`. – Zugriffsdatum: 25.2.2008

[3] Friese, Peter: *Entwicklungsturbo; Incremental Project Builder in Eclipse*. 2004

[4] Harel, David: Statecharts: A Visual Formalism for Complex Systems. In: *Science of Computer Programming* 8 (1987), June, Nr. 3, S. 231–274. – URL `citeseer.ist.psu.edu/article/harel87statecharts.html`

[5] Jeff: *Line Follower Statemachine with NXT-G*. Februar 2008. – URL `http://home.earthlink.net/~xaos69/NXT/Line_Follower/Line_Follower.html`. – Zugriffsdatum: 25.2.2008

[6] Open Source: *leJOS Project Page*. Februar 2008. – URL `http://lejos.sourceforge.net`. – Zugriffsdatum: 25.2.2008

[7] openArchitectureWare.org: *Project Page*. 2 2008. – URL `http://www.openarchitectureware.org`. – Zugriffsdatum: 25.2.2008

[8] Sommerville, Ian: *Software Engineering*. 6 th Edition. Addison Wesley, 2001 (Pearson Studium)

[9] Stahl, Thomas ; Völter, Markus: *Modellgetriebene Softwareentwicklung*. 1. dpunkt.verlag, 2005

[10] The LEGO Group: *Lego Mindstorms Homepage*. February 2007. – URL `http://mindstorms.lego.com`