Entwurf einer domänenspezifischen Sprache für elektronische Stellwerke*

Wolfgang Goerigk[1], Reinhard von Hanxleden[4], Wilhelm Hasselbring[3], Gregor Hennings[1], Reiner Jung[3], Holger Neustock[2], Heiko Schaefer[2], Christian Schneider[4], Elferik Schultz[2], Thomas Stahl[1], Steffen Weik[1] und Stefan Zeug[1]

[1] b+m Informatik AG, 24109 Melsdorf
[2] Funkwerk Information Technologies GmbH, 24145 Kiel
[3] Arbeitsgruppe Software Engineering, Universität Kiel
[4] Arbeitsgruppe Echtzeitsysteme und Eingebettete Systeme, Universität Kiel

Abstract: Die Entwicklung elektronischer Stellwerke für den Bahnbetrieb ist ein aufwändiges Unterfangen, welches sich besonders für die zahlreichen Nebenstrecken und andere kleinere Bahnanlagen häufig als unrentabel erweist. Um in Zukunft einerseits mehr Verkehr auf die Schiene zu bringen und zudem die Kosten für den Betrieb der Infrastruktur zu senken, müssen die Hardware-Komponenten günstiger werden, aber auch die Entwicklung der darauf laufenden Software produktiver erfolgen, ohne Abstriche bei der Sicherheit zu machen.

Bisher werden für elektronische Stellwerke Prozessrechner eingesetzt, welche speziell auf das jeweilige Stellwerk zugeschnitten sind. Ebenso wird die Software speziell für die jeweilige Anlage entwickelt. Beide Komonenten müssen für den Betrieb zugelassen werden. Unser Ansatz zur Produktivitätssteigerung setzt einerseits auf den Einsatz standardisierter Hardware-Komponenten aus der Industrieautomation, hier konkret speicherprogrammierbarer Steuerungen, und andererseits auf eine Verbesserung des Softwareentwicklungsprozesses durch den Einsatz modellgetriebener Softwareentwicklung mit domänenspezifischen Sprachen und dazu passenden Werkzeugen.

In diesem Beitrag stellen wir aus dem Verbundprojekt MENGES (Modellbasierte Entwurfsmethoden für eine neue Generation elektronischer Stellwerke) den Entwurf und die Implementierung einer domänenspezifischen Sprache für die Programmierung elektronischer Stellwerke vor. Die MENGES-Sprache besteht aus einer Menge textueller Teilsprachen, für deren effiziente Benutzung eine leistungsfähige Werkzeugumgebung zur Analyse der Spezifikationen, zur Code-Generierung und zur zweckgerichteten grafischen Repräsentation von Spezifikationsteilen entwickelt wird.

1 Einleitung

Der Verkehrsträger Schiene soll in der Zukunft ein wichtigere Rolle in der Verkehrslandschaft einnehmen. Um den heutigen Anforderungen bezüglich seiner Wirtschaftlichkeit

*Diese Arbeit wird im Projekt MENGES aus dem Zukunftsprogramm Wirtschaft des Landes Schleswig-Holstein, kofinanziert mit Mitteln aus dem Europäischen Fonds für regionale Entwicklung (EFRE), gefördert.

gerecht zu werden müssen die bestehenden Bahnanlagen modernisiert werden. Wesentliche Bedeutung kommt dabei der Modernisierung der Nebenstrecken zu, die zurzeit noch mechanisch oder elektromechanisch betrieben werden. Sie sollen zukünftig elektronisch gesteuert werden.

Die Entwicklung elektronischer Stellwerke ist bisher ein sehr aufwändiges Unterfangen. Die Hardware wird speziell entwickelt, die Software wird für diese Anlagen in der Regel re-implementiert bzw. angepasst. Alle Komponenten müssen darüber hinaus verifiziert und durch staatliche Behörden zugelassen werden. Für Nebenstrecken ist dieses Vorgehen häufig unrentabel. Im Rahmen des Forschungsprojekts „Entwicklung von modellbasierten Entwurfsmethoden für eine neue Generation elektronischer Stellwerke“ (MENGES) entwickeln wir eine domänenspezifische Sprache (DSL) für die Beschreibung der Steuerungssoftware, sowie die passende Werkzeuginfrastruktur für diese Sprache.

In Abschnitt 2 stellen wir die Domäne und das Anwendungsszenario Lindaunis vor. Abschnitt 3 beschreibt die konkrete Architektur einer Steuerungseinheit in Hard- und Software. Den Kernbeitrag dieser Publikation stellen die in Abschnitt 4 beschriebene DSL und die in Abschnitt 5 beschriebene Werkzeuginfrastruktur für diese DSL dar. Verwandte Arbeiten werden in Abschnitt 6 diskutiert. Abschließend wird in Abschnitt 7 eine Zusammenfassung und ein Ausblick für das Projekt gegeben.

2 Die Domäne „elektronische Stellwerke für Regionalstellwerke“

Elektronische Stellwerke sind komplexe technische Systeme bestehend aus einer Vielzahl von bahntechnischen Anlagen wie Weichen oder Signalen und aus einer komplexen Logik zur Steuerung und Überwachung dieser Anlagen. In MENGES konzentrieren wir uns auf elektronische Regionalstellwerke (ESTW-R). Als Evaluationsszenario wird das Pilotprojekt Lindaunis genutzt, welches die Umstellung der Strecke Kiel–Flensburg von mechanischem auf elektronischen Betrieb verwirklicht.

Die fachliche Domäne Die Domäne der Regionalstellwerke beinhaltet viele Fachtermini, wovon hier einige zentrale Begriffe kurz eingeführt werden (siehe auch [Pac11]). Das Gesamtkonstrukt aus Bahnanlagen und der gesamten Steuerungs- und Überwachungstechnik, welche sich über mehrere Bahnhöfe und Streckenabschnitte erstrecken kann, wird in MENGES *Stellwerk* genannt. Es gliedert sich in *Lokalstellwerke*, welche nur einen Bahnhof bedienen, und die *Bedienzentrale*, dem Arbeitsplatz des Fahrdienstleiters. Die Lokalstellwerke werden in weitere Teilbereiche, sogenannte *Freimeldeabschnitte* (FMA), unterteilt. Diese FMA bestehen aus einer Weiche oder einem Stück Gleis und dienen der Unterteilung der Strecke in diskrete Bereiche, welche, so sie befahren werden sollen, frei sein müssen. FMA werden durch *Achszähler* voneinander abgegrenzt. Achzähler registrieren in welche Richtung ein Rad bzw. eine Achse an der FMA-Grenze rollt. Für eine Zugbewegung muss ein Streckenabschnitt exklusiv reserviert und gesichert sein. Diese reservierte Strecke setzt sich aus FMA zusammen und wird *Fahrstraße* genannt. Fahrstraßen werden durch ein Hauptsignal und einen Zielpunkt begrenzt.

Die Kontrolle eines Lokalstellwerks übernimmt ein lokaler Rechner, den wir *Stellwerks-*

kern nennen. Die einzelnen Stellwerkskerne werden über die Bedienzentrale gesteuert. Die Kerne ihrerseits überwachen die Signale, Weichen, Achszähler sowie weitere Aktuatoren und Sensoren des Stellwerks. Diese überwachten Elemente werden unter dem Begriff *Stellwerkselemente* zusammengefasst.

Beispiel-Kommando „Weiche umstellen“ In diesem Papier wird das Kommando *Weiche umstellen* (WU) als durchgängiges Beispiel genutzt, um zu zeigen, wie ein Kommando von der Bedienzentrale kommend in Steuerimpulse für die Aktuatoren umgesetzt wird, und wie die Sensordaten für die Bedienzentrale aufbereitet werden. Nehmen wir an die Weiche mit der Bezeichnung 83W1 läge links und sei gegen ein Umstellen gesichert (bahntechnisch *verschlossen*). Die Weiche soll nun nach rechts umgestellt werden. Dazu sendet die Bedienzentrale ein Kommando WU(83W1) an den verantwortlichen Stellwerkskern. Dieser dekodiert das Kommando und übergibt es der Steuerungskomponente für die Weiche 83W1. Abhängig vom internen Zustand der Komponente führt sie den Befehl aus oder, so die Weiche von einem anderen Vorgang genutzt wird, weist ihn ab. Ist die Weiche nicht anderweitig reserviert oder beansprucht akzeptiert die Komponente das Kommando und leitet den Umstellvorgang ein. Dazu instruiert sie den Weichentreiber den Verschluss zu lösen und den Stellantrieb zu aktivieren. Sobald der Treiber erkennt, dass die Weiche rechts liegt oder nach einer maximalen Laufzeit wird der Antrieb ausgeschaltet und dies der Steuerungskomponente mitgeteilt. Diese wiederum liefert der Bedienzentrale das Ergebnis der Aktion.

Speicherprogrammierbare Steuerungen Der Stellwerkskern wird bei großen Projekten durch spezielle Prozessrechner realisiert. Für die eingeschränkte Domäne der elektronischen Regionalstellwerke (ESTW-R) können jedoch einfachere Produkte genutzt werden. Im Lindaunisprojekt werden hierfür speicherprogrammierbare Steuerungen (SPS) verwendet. SPS sind Steuerungscomputer aus der Industrieautomation, die dort zum Steuern von Produktionsanlagen eingesetzt werden. SPS arbeiten zyklisch. Sie lesen zu Beginn eines jeden Zyklus Werte von den Eingängen ein, verarbeiten diese und geben dann die Ergebnisse an den Ausgängen am Ende des Zyklus aus. Als Befehlssatz kommen Operatoren, Zähler, Timer, Schleifen und Verzweigungen vor, welche mit Hilfe von verschiedenen im Standard IEC 61131-3 definierten Sprachen beschrieben werden [JT09]. Die meisten SPS kennen keine parallele Verarbeitung. Selbst jene, die mehrere Threads unterstützen, behandeln die einzelnen Pfade wie einzelne separate SPS. Dies erlaubt es SPS-Programme als streng sequenziell anzusehen.

3 Architektur eines Stellwerkskerns

Die Architektur eines Stellwerkskerns besteht aus einer Reihe von Hardwarekomponenten, welche das Steuerungssystem und dessen Anbindung an die Stellwerkselemente realisieren, sowie einer Software, welche die Steuerungslogik und eine Laufzeitumgebung umfasst.

Die Hardware des Kerns besteht aus einer Reihe von speicherprogrammierbaren Steuerungen (SPS) und digitalen Ein- und Ausgabekomponenten (DIO), wobei die SPS die Steue-

rungssoftware beherbergen und die verteilten DIO zur Ansteuerung der Stellwerkselemente dienen. Die Software umfasst Laufzeitkomponenten und generierten Code, welche in Abbildung 1 als Schichtenmodell dargestellt sind.

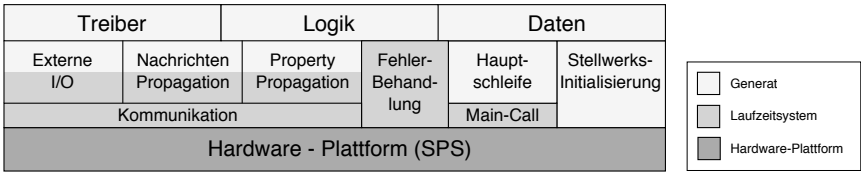

Abbildung 1: Schichtenarchitektur für die Menges Plattform

Treiber, Logik und Daten bilden die oberste Schicht, welche Struktur und Verhalten des Kerns implementiert. Darunter folgt die Anbindung der Struktur- und Verhaltensschicht an die Hardware. Diese Schicht besteht aus modellunabhängigen Komponenten (Laufzeitsystem) und aus projektspezifischen Komponenten, die generiert werden müssen.

4 Die MENGES-DSL für elektronische Stellwerke

Die DSL für Stellwerke ist in mehrere Teilsprachen unterteilt, welche im rechten Teil von Abbildung 2 dargestellt sind. Eine weitere DSL wurde für die Beschreibung von Topologie- und Projektierungsinformationen entwickelt, die jedoch austauschbar ist. Auf diese Weise kann der Heterogenität der Topologie-Domänen, die je nach Land und Anwendungsgebiet durch unterschiedliche Begrifflichkeiten geprägt sind, Rechnung getragen werden. Die Kernsprache selbst trennt die Deklaration von Schnittstellen und Typen von der Implementierung (Logik). Die Instantiierung wird mittels einer separaten Sprache beschrieben, da diese einerseits für unsere sicherheitskritische Anwendung statisch erfolgt und andererseits auf konkreten topologischen Daten beruht. Die letztendliche Abbildung auf reale Hardware wird in der Hardware-Aufbau-Sprache spezifiziert.

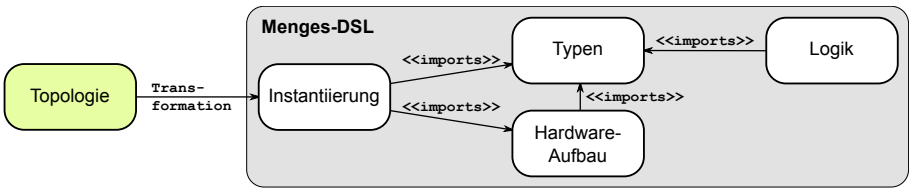

Abbildung 2: Teilspachen und Metamodelle der DSL für elektronische Stellwerke

Im Folgenden stellen wir die Beschreibung der Topologie (Abschnitt 4.1), der Typen für Stellwerkselemente (Abschnitt 4.2), der Stellwerkslogik (Abschnitt 4.3), der Hardware-Konfiguration (Abschnitt 4.4) und der Instantiierung (Abschnitt 4.5) vor.

4.1 Topologie der Gleise

Die Topologie- und Projektierungssprache erlaubt die Beschreibung von Gleistopologien und weiterer betrieblicher Strukturen. Die Basis der Beschreibung sind Gleise und Weichen, welche auf ein Knoten-Kanten-Modell abgebildet werden. Diesem Gleisgraph werden die weiteren Stellwerkselemente einbeschrieben.

Die Anzahl der Kanten pro Basisknoten hängt vom konkreten Knotentyp ab und wird durch eine Menge vordefinierter Kindstrukturen, den *Ports*, bestimmt. Ports tragen Namen und implizieren eine bestimmte Traversierungs-Semantik der Knoten. Eine Weiche ist ein spezieller Basisknoten mit den Ports head, diversion und main. Erlaubte Routen über den Basisknoten Weiche führen von head zu main bzw. head zu diversion. Diese internen Traversierungen erlauben es verschiedene Prüfungen auf dem Graphen durchzuführen, wie z.B. die Zulässigkeit der Route einer Fahrstraße. Neben Weichen kennt die Sprache noch Stumpfgleisknoten und Einbruchsstellen, die jeweils nur einen Port besitzen. Gleiskanten haben keine räumliche metrische Ausdehnung. Die Einbeschreibung von Stellwerkselementen wie Signalen oder Achszählern erfolgt als ordinale Liste. Diese Elemente können eine Richtung aufweisen (Signale) oder ohne Richtung sein (Achszähler). Basierend auf den Achszählern werden Freimeldeabschnitte definiert, welche dann zur Beschreibung der Fahrstraßen genutzt werden. Eine Weiche (mit Traversierungskanten), Stumpfgleise, und Gleiskanten sind schematisch in Abbildung 3 dargestellt, Signale und Achszähler durch schmale Pfeile bzw. Rechtecke auf den Gleiskanten angedeutet.

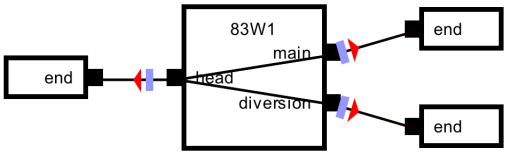

Abbildung 3: Knoten-Kanten-Darstellung einer mit Stumpfgleisen verbundenen Weiche.

4.2 Typen der Stellwerkselemente

Klassen-Schnittstellen, Konnektoren, Aufzählungstypen und Zustandsmengen für Stellwerkselemente werden in der Typensprache beschrieben. Sie ist das Bindeglied für die anderen Teilsprachen. Die Klassen-Schnittstellen heißen **interlocking element**. Sie kömmem Strukturen wie Datenfelder, Zustandsvariablen, Timer, Prädikate und Rollen enthalten. Datenfelder können von einem einen primitiven Datentyp oder einer Klasse sein. Zustandsvariablen dagegen sind immer vom Typ Zustandsmenge (nominale Werte und eine Zustandsübergangsrelation). Andere nominale Begriffe werden in der DSL mittels Aufzählungstypen realisiert. Für Timer müssen nur Namen angegeben werden. Sie erhalten zur Laufzeit ihren Startwert. Prädikate dienen dazu, komplexere logische Ausdrücke auf verständliche Begriffe abzubilden um so die Logik lesbar zu halten. Listing 1 zeigt eine einfache **interlocking element**-Definition.

```
interlocking element GaWeiche {
    properties
        Weichenlage lage;
    statevars
        WeichenumstellungStates wuMaschine;
    roles
        WeichenKommando.WeichenLogik interpreter;
        WeichenTreiberCom.WeichenLogik treiber;
}
```

Listing 1: Typdeklaration der Weichen-Logik

Für die Kommunikation zwischen Komponenten wird ein Rollenmodell genutzt. Klassen füllen diese Rollen aus. Die Kommunikation zwischen den einzelnen Rollen wird mit dem Konstrukt **communication description** beschrieben. Es ermöglicht den zyklischen Austausch von Eigenschaftswerten, wie auch das ereignisbasierte Übertragen von Nachrichten gemäß vorgegebenen Protokollen. Das Propagieren erfolgt gerichtet. Das heißt, nur eine Rolle darf schreiben, während die andere Rolle nur lesen darf. Die Richtung wird, wie in Listing 2 gezeigt, durch einen Pfeil zwischen den Rollen angegeben.

```
communication description WeichenTreiberCom : WeichenLogik, WeichenTreiber {
    properties WeichenTreiber -> WeichenLogik:
        boolean li;
    properties WeichenLogik -> WeichenTreiber:
        boolean motor_li;
}
```

Listing 2: Kommunikation zwischen Logik und Treiber für die Weiche (Auszug)

Das Protokollmodell basiert auf Nachrichten, die in unserem Fall durch einen Bezeichner benannt werden. Ferner können sie über ein Payload, beschrieben durch Parameter, verfügen. Aus den Rollen und Nachrichten lassen sich Protokolle zusammenstellen, was in Listing 3 dargestellt ist. Das Beispiel definiert, dass auf eine Nachricht WU der Rolle KommandoInterpreter die Rolle WeichenLogik mit einem akzeptiert oder abgelehnt antworten muss. Mit Hilfe dieser Protokolle soll es ermöglicht werden, Verklemmungen zwischen den Automaten und Prozessen automatisch zu ermitteln.

```
KommandoInterpreter WU -> WeichenLogik (akzeptiert,(erfolgreich|fehler))|abgelehnt -> KommandoInterpreter ;
```

Listing 3: Kommunikation zwischen Kommandointerpreter und Weichen-Logik im ESTW-R

Die Laufzeitumgebung stellt für beide Kommunikationsarten sicher, dass die Daten ihr Ziel in einem bestimmten Zeitfenster erreichen. So dies nicht gelingt, greift die Fehlerbehandlung und das System wechselt in einen Fehlerzustand.

4.3 Logik und Verhalten

Das Verhalten der Komponenten wird in der Logik-Sprache spezifiziert. Sie umfasst Konstrukte für die Beschreibung von betrieblichen und technischen Prozessen, Zustandsautomaten und speziellen bedingten und unbedingten Aktionen. Da die Beschreibung des Verhaltens eines **interlocking element** sehr groß werden kann, kann die Logik über mehrere Spezifikationsdokumente verteilt werden.

Die einfachsten Verhaltensbeschreibungkonstrukte sind *Aktionen*, welche Datenfelder verändern, Nachrichten versenden und Timer starten, und *Entscheidungen*, welche Datenfelder auslesen, Nachrichten empfangen und Timer abfragen (siehe Listing 4). Entscheidungen können geschachtelt werden, wobei Aktionen nur in den Blättern der so entstehenden Entscheidungsbäume stehen dürfen. Die Semantik von Entscheidungen ist wie folgt definiert: Die einzelnen Fälle werden der Reihe nach evaluiert bis eine Bedingung erfüllt ist. Existiert ein solcher Fall, wird dessen Aktion ausgeführt und die Evaluation beendet.

```
action weichenMotorEinschalten {
    this.motor = LINKS;
    start timerMotor(7000);
}
conditional prüfeVerschlossen() {
    case this.verschlossen:
        verschlussLösen();
    default:
        weichenMotorEinschalten();
}
```

Listing 4: Implementierung des Verhaltens mit Aktionen und Entscheidungen

Zustandsautomaten erlauben es komplexeres zustandsbehaftetes Verhalten zu beschreiben. Jeder Automat referenziert hierzu eine Zustandsvariable, für die er Transitionen definiert. So ist es möglich für verschiedene Zustandsvariablen verschiedene Automaten zu spezifizieren, welche dieselbe Zustandsmenge nutzen. Die Transitionen bestehen aus einem Guard und einer Aktion. Ist der Guard wahr, wird die Aktion ausgeführt. Die Priorität der Transitionen wird explizit durch eine Präzedenznummer angegeben. Des Weiteren können Transitionen wahlweise entweder direkt nach dem Betreten ihres Ursprungsszustandes oder erst in künftigen Ausführungszyklen evaluiert werden. Dieses Prinzip ist Andrés synchronem StateChart-Formalismus *SyncCharts* entnommen [And03].

Neben Automaten lassen sich betriebliche und technische Vorgänge auch mit an UML-Aktivitätsdiagrammen angelehnten Prozessen ausdrücken (siehe Listing 5). Diese Prozesse kennen nur Entscheidungen und Aktionen. Sie werden entweder durch Nachrichten angestoßen oder von einer Aktion aufgerufen. Die Semantik dieser *Prozessaktionen* ist weitestgehend gleich der oben beschriebenen Aktionen. Zusätzlich können sie noch einen Verweis (**continue**) auf die nächste Aktion oder eine Entscheidung enthalten. Wird dieser Verweis weggelassen, terminiert der Prozess in der Aktion. Entscheidungen haben, anders als oben beschrieben, ein blockierendes Verhalten, falls kein Fall wahr ist. Soll der Prozess nicht blockieren, muss ein **default**-Fall angegeben werden.

4.4 Hardware-Konfiguration

Die Hardware-Konfiguration umfasst die Beschreibung der Hardware bzgl. ihrer verfügbaren Anschlüsse. Daneben werden in dieser Beschreibung die *Typen*, welche auf das entsprechende Gerät ausgebracht werden, und die maximale Anzahl ihrer Instanzen benannt.

```
process WU { start prüfeVerschlossen;

    condition prüfeVerschlossen {
        case this.verschlossen: verschlussLösen();
        default: weichenMotorEinschalten();
    }

    action weichenMotorEinschalten {
        ...
        continue prüfeUmlauf;
    }
}
```

Listing 5: Prozess für das Kommando Weiche umstellen (Auszug)

4.5 Instantiierung

In sicherheitskritischen Systemen ist es notwendig, alle benötigten Ressourcen statisch zu allozieren [Hol06]. Dem Rechnung tragend führt die Instantiierungssprache die Beschreibungen der Soft- und Hardware zusammen. Spezifikationen in dieser Sprache instantiieren Geräte (**devices**) und Klassen (**deployment**), und legen die Verbindungen der Hard- und Softwarekomponenten sowie der externen Anschlüsse fest. Darüber hinaus belegen sie die Parameter der verwendeten Softwarekomponenten, definieren konkrete Kommunikationspartner und bestimmen die zyklisch auszuführenden Verhaltenskonstrukte (vgl. Listing 6).

```
interlocking unit AEC {
    devices
        HIMatrix−F30−S aec−s;

    deployment
        aec−s.GaWeiche = { 83W1 { lage = links }, 83W2 }
        aec−s.GpWeiche = { T83W1, T83W2 }

    connections
        WeichenTreiberCom<GaWeiche, GpWeiche>:
            81W1, T81W1; 81W2, T81W2;

    tasks
        GALogik = { GaWeiche.weichenMotorEinschalten();
                    GaWeiche.prüfeVerschlossen(); }
}
```

Listing 6: Auszug aus der Instantiierungsbeschreibung für das Beispielstellwerk Lindaunis

5 Integrierte Entwicklungsumgebung

Die Eclipse-RCP[1] stellt die Basis für unsere domänenspezifische Entwicklungsumgebung bereit. Editoren für die oben beschrieben textuellen Teilsprachen wurden mit Hilfe des Xtext-Frameworks[2] erzeugt. Ausgehend von einer formalen Beschreibung der konkreten Syntax generiert Xtext entsprechende Parser, Serialisierer und zugeschnittene Editoren, welche das Spezifizieren auf vielfältige Weise unterstützen.

[1] http://www.eclipse.org

[2] http://www.xtext.org

5.1 Textuelle vs. Grafische Darstellungen

Textuelle, konkrete Syntaxen können sehr kompakt und präzise sein, wie sich anhand der Teilsprache *Typen* (vgl. Abschnitt 4.2, Listings 1 bis 3) beobachten lässt. Zudem lassen sich textuelle Spezifikationen leicht versionieren und effizient bearbeiten. Werden diese Spezifikationen jedoch umfangreicher, kann die gewonnene Kompaktheit zu Lasten der Übersichtlichkeit und der Verständlichkeit gehen. Dies betrifft besonders Beschreibungen von Verhalten (Dynamik), wie anhand von Listing 7 deutlich wird. Grafische Repräsentationen sind in solchen Fällen oft das geeignetere Darstellungsmittel, vgl. Abbildung 4. Sie sind durch die geeignete Wahl von Symbolen zumeist sprechender und einfacher zu erfassen. Ihre Erstellung und Bearbeitung ist häufig jedoch sehr zeitaufwändig [FvH10].

```
state machine Gleisabschnitt_beansprucht_SM {
 start nicht          –> DWeg, FWZ, FLR, FWR;
 DWeg                 –> nicht, DWeg_FLR, FWZ_DWeg;
 FLR                  –> nicht, DWeg_FLR, FLR_FLR,
                         FWR_FLR, FWZ_FLR;
 FWR                  –> nicht, FWR_FLR;
 FWZ                  –> nicht, FWZ_FLR, FWZ_DWeg;
 DWeg_FLR             –> DWeg, FLR, DWeg_FLR_FLR;
 FLR_FLR              –> FLR, DWeg_FLR_FLR,
                         FWR_FLR_FLR,
                         FWZ_FLR_FLR;
 FWR_FLR              –> FLR, FWR, FWR_FLR_FLR;
 FWZ_DWeg             –> DWeg, FWZ;
 FWZ_FLR              –> FLR, FWZ, FWZ_FLR_FLR;
 DWeg_FLR_FLR         –> DWeg_FLR, FLR_FLR;
 FWR_FLR_FLR          –> FWR_FLR, FLR_FLR;
 FWZ_FLR_FLR          –> FWZ_FLR, FLR_FLR; }
```

Listing 7: Zustände und gültige Übergänge

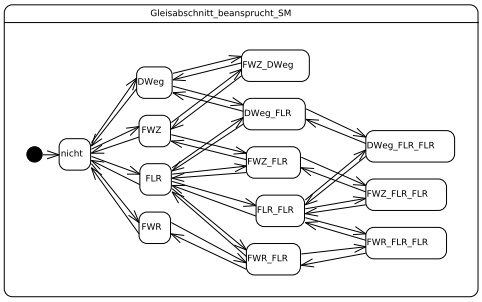

Abbildung 4: Grafische Darstellung zu Listing 7

Aus obigen Überlegungen leitet sich das Ziel ab, Vorteile beider Paradigmen zu vereinigen. Nachteile wie der Bedarf nach händischem Arrangieren von Symbolen können dabei durch Werkzeuge ausgeglichen werden. Fuhrmann und von Hanxleden fassen solche Aspekte unter dem Begriff der *Pragmatik von Modellierungssprachen* zusammen [FvH10]. Der Begriff Pragmatik ist dabei angelehnt an Morris' Auffassung von der Pragmatik natürlicher Sprache [Mor38]. Gemeinsam mit der Syntax und der Semantik bestimmt sie, wie Bedeutung in Sprache ausgedrückt und verstanden wird.

Als Basis für die Entwicklung effizienter Spezifikations- und Browsingtechniken dient das KIELER-Projekt,[3] dessen Kernkomponenten das automatische Positionieren von Diagrammelementen (KIELER Meta Layout) und das Management verschiedenartiger Sichten auf Modelle (KIELER View Management) sind [FvH10].

5.2 Transiente grafische Darstellungen

In MENGES liegen die Spezifikationen stets in textueller Form vor. Um diese grafisch darzustellen, wird der im Rahmen des KIELER-Projektes entwickelte Mechanismus der transi-

[3] http://www.informatik.uni-kiel.de/rtsys/kieler

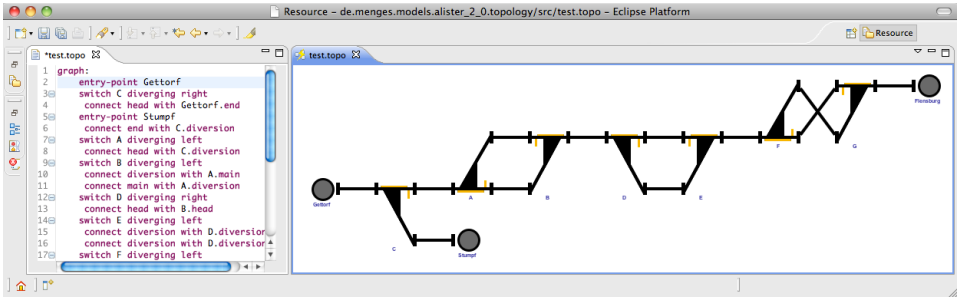

Abbildung 5: Textuelle Topologie-Spezifkation (links) und grafische Darstellung (rechts)

enten Sichten genutzt. Im Gegensatz zur händischen Erzeugung von Diagrammen ermöglicht er die Synthese sprechender Darstellungen. Dies kann sowohl automatisch wie auch *on-demand* geschehen. So wurde die grafische Darstellung in Abbildung 5 ohne das explizite Anfordern durch den Modellierer aus der im linken Bereich gezeigten Spezifikation erzeugt. Etwaige Änderungen wie die Korrektur der Lage der rechts oben positionierten Weiche werden unmittelbar in die Grafik übernommen. Der Inhalt der Grafik muss dabei nicht auf eine Teilspezifikation beschränkt sein. Dies ermöglicht unter anderem das Expandieren von referenzierten und in anderen Dokumenten definierten Elementen. Zudem kann die grafische Darstellung als Navigationshilfe genutzt werden. Die Selektion eines Diagrammelements rückt automatisch das zugehörige Element in der Quelle in den Fokus.

5.3 Code-Generierung und Simulation

Aus Spezifikationen in der textuellen DSL wird mittels Xpand/Xtend[4] Code entsprechend dem Standard IEC 61131-3 [JT09] generiert, welcher zur Simulation mit CoDeSys[5] ausgeführt und mit dem Monitoring-Framework Kieker[6] [vHRH+09] analysiert wird.

6 Verwandte Arbeiten

Unser Ansatz zielt darauf die Entwicklung der Software für Stellwerkskerne mit einem generativen Ansatz und einer zugeschnittenen DSL effizienter zu gestalten. Der Ansatz in [SOE+08] verfolgt das gleiche Ziel, ist aber weitestgehend komplementär. Er geht davon aus, dass es zu einzelnen Elementen der Topologie vorgefertigte Softwarekomponenten gibt. Die Gesamtkomposition erfolgt hier durch die Beschreibung der Topologie, woraus ein Generator Zielcode für ein Lokalstellwerk erzeugt. Die verwendete Topologiesprache *Train Control Language* (TCL) könnte so auch mit der MENGES-DSL genutzt werden. Der

[4] http://eclipse.org/modeling/m2t/?project=xpand
[5] http://www.3s-software.com/
[6] http://kieker.sourceforge.net/

Zielcode wäre in diesem Fall die Instantiierungssprache. Aufbauend auf der TCL wird in [SZLT+11] eine Fallstudie beschrieben, welche mit Produktlinien und Topologiemustern arbeitet, um so die Komplexität von Topologien zu reduzieren. Dieses Vorgehen ist in MENGES ebenso möglich, steht aber nicht im Fokus dieser Arbeit.

Haxthausen und Peleska beschreiben in [HP02] eine DSL mit dem Fokus auf Routenplanung und sicheren Zuglenkungsbetrieb. ESTW-R hingegen enthalten keine automatische Zuglenkung, Fahrwege werden nach wie vor durch Fahrdienstleiter eingestellt.

Das Anliegen der Arbeit von Thomas et al. [T+05] ist vergleichbar mit unserer Situation: Die Aufgabe des vielfachen Erstellens von SPS-Code soll mit einem generativen Ansatz bewältigt werden. Die verwendeten Technologien unterscheiden sich jedoch grundlegend. Neben der reinen Entwicklung von DSL & Code-Generierung steht in MENGES jedoch auch Technologietransfer von der Wissenschaft in die mittelständische Wirtschaft im Fokus.

7 Zusammenfassung und Ausblick

In dieser Arbeit wird der Entwurf einer domänenspezifischen Sprache als Menge textueller Teilsprachen für die Programmierung elektronischer Stellwerke vorgestellt. Diese wurden in einer etablierten Entwicklungsumgebung implementiert und um weitere pragmatische Funktionen (Modellsichten, Navigationshilfen) ergänzt. Mit dem so entstandenen Werkzeug kann modellgetrieben SPS-Code [JT09] generiert werden.

Die bisherigen Anwendungen auf Testszenarien haben gezeigt, dass die Sprache gut zur Beschreibung von Steuerungsprozessen geeignet ist. Im Vergleich zu den bisherigen Spezifikationen der Funkwerk Information Technologies GmbH mit sehr großen Zustandsdiagrammen ist die textuelle Beschreibung, unterstützt durch zweckgerichtete grafischen Modellansichten, wesentlich kompakter und gleichzeitig übersichtlicher und handlicher.

Zurzeit evaluieren wir die Sprache an größeren Beispielen aus dem Projektgeschäft der Funkwerk Information Technologies GmbH systematisch mit Hilfe der GQM-Methode [BCR94, SB99]. Konkret werden wir Verhalten mit Prozessen und Automaten spezifizieren und überprüfen, für welche Problemstellungen sich die jeweiligen Sprachkonstrukte optimal eignen. Ferner werden wir die Werkzeuge mit weiteren grafischen Sichten versehen und Model-Checking-Methoden [BBB+04] für die Entwickler bereitstellen. Für die dynamische Analyse der erstellten Steuerungssysteme wird modellgetriebene Instrumentierung genutzt [BH09, vHKGH11].

Literatur

[And03] Charles André. Semantics of SyncCharts. Bericht ISRN I3S/RR–2003–24–FR, I3S Laboratory, Sophia-Antipolis, France, April 2003.

[BBB+04] R. Buschermöhle, M. Brörkens, I. Brückner, W. Damm, W. Hasselbring, B. Josko, C. Schulte und T. Wolf. Model Checking — Grundlagen und Praxiserfahrungen.

Informatik-Spektrum, 27(2):146–158, April 2004.

[BCR94] Victor R. Basili, Gianluigi Caldiera und H. Dieter Rombach. The Goal Question Metric Approach. In *Encyclopedia of Software Engineering*. Wiley, 1994.

[BH09] Marko Boskovic und Wilhelm Hasselbring. Model Driven Performance Measurement and Assessment with MoDePeMART. In *Proceedings of the ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems (MODELS '09)*, Jgg. 5795 of *LNCS*, Seiten 62–76. Springer, Oktober 2009.

[FvH10] Hauke Fuhrmann und Reinhard von Hanxleden. Taming Graphical Modeling. In Dorina C. Petriu, Nicolas Rouquette und Øystein Haugen, Hrsg., *Proceedings of the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS '10)*, LNCS. Springer, Oktober 2010.

[Hol06] Gerard J. Holzmann. The Power of 10: Rules for developing Safety-Critical Code. *IEEE Computer*, 39(6):95–97, Juni 2006.

[HP02] Anne E. Haxthausen und Jan Peleska. A domain specific language for railway control systems. In *Proceedings of the sixth biennial World Conference on Integrated Design & Process Technology (IDPT '05)*, Juni 2002.

[JT09] Karl Heinz John und Michael Tiegelkamp. *SPS-Programmierung mit IEC 61131-3: Konzepte und Programmiersprachen, Anforderungen an Programmiersysteme, Entscheidungshilfen*. Springer, 4. Auflage, 2009.

[Mor38] Charles William Morris. *Foundations of the theory of signs*, Jgg. 1 of *International encyclopedia of unified science*. The University of Chicago Press, Chicago, 1938.

[Pac11] Jörn Pachl. *Systemtechnik des Schienenverkehrs: Bahnbetrieb planen, steuern und sichern*. Vieweg+Teubner, 6. Auflage, 2011.

[SB99] R. Solingen und E. Berghout. *The Goal/Question/Metric method: a practical guide for quality improvement of software development*. McGraw-Hill, 1999.

[SOE+08] Andreas Svendsen, Gøran K. Olsen, Jan Endresen, Thomas Moen, Erik Carlson, Al Kjell-Joar und Oystein Haugen. The Future of Train Signaling. In *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems (MoDELS '08)*, Seiten 128–142, Berlin, Heidelberg, 2008. Springer-Verlag.

[SZLT+11] Andreas Svendsen, Xiaorui Zhang, Roy Lind-Tviberg, Franck Fleurey, Oystein Haugen, Birger Moller-Pedersen und Goran Olsen. Developing a Software Product Line for Train Control: A Case Study of CVL. In Jan Bosch und Jaejoon Lee, Hrsg., *Software Product Lines: Going Beyond*, Jgg. 6287 of *LNCS*, Seiten 106–120. Springer Berlin / Heidelberg, 2011.

[T+05] G. Thomas et al. LHC GCS: A Model-Driven Approach for Automatic PLC and SCADA Code Generation. In *Proceedings of the 10th International Conference on Accelerator and Large Experimental Physics Control Systems (iCALEPCS)*, 2005.

[vHKGH11] André van Hoorn, Holger Knoche, Wolfgang Goerigk und Wilhelm Hasselbring. Model-Driven Instrumentation for Dynamic Analysis of Legacy Software Systems. In *Proceedings of the 13th Workshop Software-Reengineering (WSR '11)*, Mai 2011.

[vHRH+09] André van Hoorn, Matthias Rohr, Wilhelm Hasselbring, Jan Waller, Jens Ehlers, Sören Frey und Dennis Kieselhorst. Continuous Monitoring of Software Services: Design and Application of the Kieker Framework. Bericht TR-0921, Department of Computer Science, University of Kiel, Germany, November 2009.