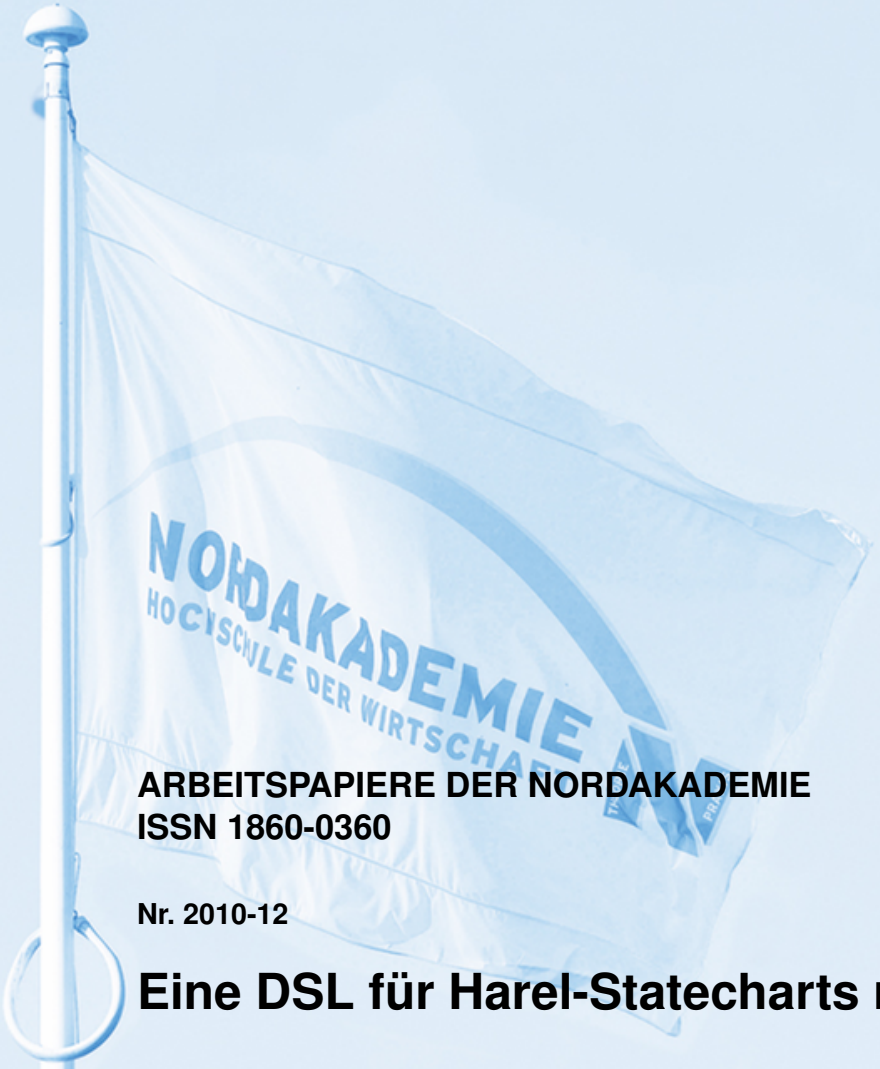

ARBEITSPAPIERE DER NORDAKADEMIE
ISSN 1860-0360

Nr. 2010-12

Eine DSL für Harel-Statecharts mit PetitParser

Hartmut Krasemann
Johannes Brauer
Christoph Crasemann

Oktober 2010

Dieses Arbeitspapier ist als PDF verfügbar: http://www.nordakademie.de/arbeitspapier.html

NORDAKADEMIE
HOCHSCHULE DER WIRTSCHAFT
THEORIE PRAXIS
Köllner Chaussee 11
25337 Elmshorn
http://www.nordakademie.de

Eine DSL für Harel-Statecharts mit PetitParser

Hartmut Krasemann, Johannes Brauer, Christoph Crasemann

29. Oktober 2010

Inhaltsverzeichnis

Eine *AuDSL* genannte domänenspezifische Sprache zur Beschreibung von Harel-Statechart-Automaten wird benutzt, um exemplarisch die Eignung von *PetitParser*, einem zum *Helvetia*-Framework gehörenden Parser-Combinator, als Baustein einer Language-Workbench zu erproben. Verwendet wird die Pharo[1]-Implementierung von PetitParser. Neben der Implementierung der AuDSL als PetitParser-Skript wird auch die Laufzeitumgebung vorgestellt, die die Ausführung der Automaten ermöglicht. Die Einbettung von AuDSL-Texten in die Wirtssprache Smalltalk wird gezeigt. Die gemachten Erfahrungen zeigen grundsätzlich die Einfachheit des Baus von DSLs mit PetitParser. Die Integration einer DSL in die Werkzeuge der Entwicklungsumgebung bleibt ohne die Nutzung des Helvetia-Frameworks aber unvollständig. Die Übertragbarkeit der Vorgehensweise auf andere Programmiersprachen (z. B. Scala) wird angedeutet.

[1] ein Smaltalk-Dialekt (vgl. `http://pharo-project.org`)

1 Einleitung

In dem Informatik-Spektrum-Artikel *Auf dem Weg zu idealen Programmierwerkzeugen*[2] haben die Autoren zwei wesentliche Defizite heutiger Programmierumgebungen benannt:

- Die „Modellierungslücke“, i.e. der konzeptuelle Abstand zwischen Problemraum (Anforderungen) und Lösungsraum (Programmiersprachen) ist zu groß.

[2] Brauer, J., C. Crasemann und H. Krasemann: *Auf dem Weg zu idealen Programmierwerkzeugen – Bestandsaufnahme und Ausblick*. Informatik-Spektrum, 31(6):580 – 590, Dezember 2008

- Es fehlt an Orthogonalität und Optionalität der Implementierungskonzepte, nicht nur bei Typen, sondern bei den verschiedensten Aspekten eines vollständigen Software-Systems.

Als ein Element zur Lösung dieser Probleme halten die Autoren DSLs[3] für aussichtsreich. Dies dürfen allerdings keine separaten DSLs sein, wie es sie in heutigen Software-Projekten und -Systemen in großer Zahl bereits gibt, sondern eingebettete DSLs, in der der Programmierer Funktionalität frei zwischen der Wirtssprache und der DSL verteilen kann.

[3] Domänen-spezifische Sprachen

Für die Anwendungsprogrammierung sind besonders so genannte *vertikale* DSLs interessant, die – ausgehend von den Anforderungen – im Projekt selbst gebaut werden. Dagegen erwarten die Autoren, dass so genannte *horizontale* DSLs, die es erlauben, querschnittliche Funktionalitäten zu formulieren, in absehbarer Zeit zu einem einkaufbaren Bestandteil der Programmierumgebungen werden.

Die Begriffe *vertikal* und *horizontal* hat die OMG ursprünglich für Frameworks benutzt. Horizontale Frameworks beinhalten technisch querschnittliche, vertikale anwendungsspezifische Themen.

Mit DSLs auf der konzeptuellen Ebene der Anforderungen werden die Modelle des Designs zu direkten Code-Bestandteilen. Dies sehen die Autoren als eine viel versprechende Alternative zur heutigen modellgetriebenen Entwicklung, die von der Generierung von Programmen in der Zielsprache geprägt ist.

Noch fehlt uns die Erfahrung, wie sich das Programmieren mit solchen DSLs in der Praxis „anfühlt“:

- Wie gut funktioniert die Orthogonalität, i.e. wie gut sind Änderungen oder Erweiterungen in der DSL selbst entkoppelt vom Programm in der unterliegenden Wirtssprache (oder in einer anderen DSL)?
- Wird die Situation gebessert im Vergleich mit der Framework-Programmierung oder dem Generieren?
- Braucht eine DSL ihren eigenen Editor?
- Wie funktioniert das Debugging mit einer solchen DSL?
- Welche Anwendungsteile sind gut, welche schlechter geeignet?
- Wie leicht oder wie schwer ist es, eine DSL zu bauen?

Die Autoren haben einige Erfahrung mit dem Programmieren so genannter *fluent interfaces*[4] und interner DSLs ohne neue Syntax, wie es besonders Smalltalk- und Lisp-Programmierer gerne praktizieren. In diesem Programmierstil wird zunächst mit Hilfe von Abstraktion, syntaktischen Tricks oder Makros innerhalb der Wirtssprache eine höhere Sprache definiert, die ihrerseits zur Basis der Problemlösung wird.

[4] Fowler, M.: *http://martinfowler.com/bliki/FluentInterface.html*

DSLs, die derart in die Wirtssprache eingebettet sind, dass in ihnen die ganze Wirtssprache zur Verfügung steht, sind aber nur eine mögliche Spielart der Einbettung. Eine andere Spielart sind Konfigurationssprachen für Code, der andere Teile des Programms, meist zum Initialisierungszeitpunkt, konfiguriert. Hier gibt es mit der Unterscheidung von Programminitialisierung und Ausführungszeit eine klare Trennlinie. Die Wechselwirkung mit der Wirtssprache ist nur zum Initialisierungszeitpunkt relevant, zur Ausführungszeit

beherrscht eine Programmschnittstelle das Bild. Viele zur eigentlichen Programmlogik orthogonalen Aspekte eines Programms, sind allerdings *nur* Konfigurationen, man denke an

- die Konfiguration eines Zustandsautomaten,
- die Konfiguration von Programmkomponenten (z. B. nach Siedersleben[5]) selbst,
- Schnittstellen-Spezifikationen,
- Spezifikation von Persistenz und Verteilung oder
- andere querschnittliche Aspekte wie das Logging.

Wir können deshalb bei den internen DSLs unterscheiden zwischen

- transparenten DSLs, die in die Wirtssprache voll integriert sind und diese durchscheinen lassen und
- Konfigurations-DSLs, die orthogonal zur Wirtssprache sind, nur zum Initialisierungszeitpunkt ausgeführt werden und Strukturen in der Wirtssprache erzeugen, die zur Laufzeit über ein API benutzt werden.

[5] SIEDERSLEBEN, J. (Hrsg.): *Moderne Software-Architektur: Umsichtig planen, robust bauen mit Quasar*. dpunkt, Heidelberg, 2004

FÜR DEN BAU VON DSLs stehen auch kommerzielle Produkte bereit. Hierzu gehören z. B.

- das *Meta Programming System*[6] (MPS) der Fa. JetBrains,
- *MetaEdit+*[7] von MetaCase,
- die *Intentional Domain Workbench* von Intentional Software[8] und
- Xtext[9].

Die Autoren besitzen keine eigenen, praktischen Erfahrung mit diesen Werkzeugen, deswegen seien an dieser Stelle nur ein paar kurze Bemerkungen gestattet.

[6] `http://www.jetbrains.com/mps/`)

[7] `http://www.metacase.com/mep/`

[8] `http://intentsoft.com`

[9] `http://www.eclipse.org/Xtext/`

MPS ist ein Werkzeug für die Definition insbesondere textueller DSLs die letztlich auf Java-Code abgebildet werden. MPS liefert eine fertige an Java angelehnte *BaseLanguage* mit, die als Basis für die Definition von DSLs benutzt wird. MPS erlaubt die Definition von DSL-spezifischen Editoren, die das Arbeiten mit einer „fertigen" DSL erleichtern sollen.

Ein Kollege, der MPS evaluiert hat, schrieb den Autoren: „Die MPS Entwicklungsumgebung ist hoch professionell, aber der Aufwand für die Erstellung eines trivialen Beispiels ist ernüchternd. Das Tutorial umfasst 90 Bildschirmseiten (!) – ich habe etwa nach gut der Hälfte abgebrochen. Das scheint mir in keinem sinnvollen Kosten-Nutzenverhältnis zu stehen."

Bei *MetaEdit+* handelt es sich um ein in VisualWorks-Smalltalk geschriebenes Werkzeug, das vorrangig die Definition grafischer, domänenspezifischer Modellierungsprachen (einschließlich der dazu gehörigen Diagramm-Editoren, Browser und Generatoren) ermöglicht.[10] Entwickler benutzen dann diese Modellierungsprachen zur Modellierung von Anwendungen. Aus diesen Modellen wird dann Code generiert. Da die Code-Generierung in *MetaEdit+* selbst spezifiziert wird, kann nahezu jede Zielsprache verwendet werden.

[10] KELLY, S. und J.-P. TOLVANEN: *Domain-Specific Modeling: Enabling Full Code Generation*. John Wiley & Sons, 2008

Die *Intentional Domain Workbench* bezeichnet Martin Fowler in seinem Blog als ein „Musterbeispiel" für eine *Language Workbench*.[11] Zu den besonders bemerkenswerten Eigenschaften dieses Werkzeugs gehört, dass zu einem semantischen Modell Projektionen in unterschiedliche Zielsprachen definiert werden können. Eine solche Projektion kann auch editiert werden, was sich wiederum „rückwärts" auf das semantische Modell und damit eventuell vorhandene andere Projektionen auswirkt.

[11] `http://martinfowler.com/bliki/IntentionalSoftware.html`

Xtext erlaubt die komfortable Entwicklung von externen, textuellen DSLs, wobei eine weitgehende Integration in die Eclipse-Plattform gegeben ist. Dies bezieht sich nicht nur auf syntaxgesteuerte Editoren sondern auch auch auf das Eclipse Modeling Framework (EMF). Aus in *Xtext* geschriebenen Modellen kann ein Parser generiert werden. Um diese Modelle letztlich ausführbar zu machen, kann entweder Code für eine beliebige Plattform generiert werden[12] oder sie werden durch eine zur Java Virtual Machine kompatible Sprache interpretiert.

[12] z. B. mithilfe von *Xpand*(`http://www.eclipse.org/modeling/m2t/?project=xpand`)

Alle hier genannten Werkzeuge sind u. E. aber nicht für die Entwicklung transparenter DSLs geeignet.

Mit Helvetia[13] existiert ein Framework für die transparente Einbettung von DSLs in die Wirtssprache Smalltalk. Der PetitParser ist einerseits ein Bestandteil von Helvetia, andererseits allein bereits ein vollständiges Framework für Parser-Kombinatoren in Smalltalk, mit dem sich ein Compiler für eine beliebige DSL schreiben lässt.[14] Mit dem PetitParser alleine ist es also möglich, auch ohne die übrigen Helvetia-Funktionalitäten eine Konfigurations-DSL zu definieren.

[13] Renggli, L., T. Gîrba und O. Nierstrasz: *Embedding Languages Without Breaking Tools*. In: *ECOOP 2010: Proceedings of the 24th European Conference on Object-Oriented Programming*. Springer-Verlag, 2010

[14] Renggli, L., S. Ducasse, T. Gîrba und O. Nierstrasz: *Practical Dynamic Grammars for Dynamic Languages*. In: *4th Workshop on Dynamic Languages and Applications (DYLA 2010)*. ACM, 2010

Als erste Übung betrachten wir in dieser Arbeit die Konfiguration eines Zustandsautomaten. Dazu definieren wir eine Konfigurations-DSL, die wir *AuDSL* (Automaten-DSL) nennen. Wir konstruieren dafür die Grammatik und den Compiler sowie die Laufzeit-Strukturen, die vom Compiler erzeugt werden. Die folgenden Kapitel

- rekapitulieren Harel-Statecharts,
- skizzieren den PetitParser und
- beschreiben die Implementierung der AuDSL[15] sowie
- die damit gesammelten Erfahrungen.

Die Ergebnisse dieser Arbeit sind auf andere Programmierumgebungen bzw. Sprachen übertragbar, sofern dort Parserkombinatoren zur Verfügung stehen. Dies ist z. B. in der Java-Welt mit den Parser-Kombinatoren von Scala der Fall.

[15] Krasemann, H., C. Crasemann und J. Brauer: *Monticello-Package AuDSL3-hk25.mcz für Pharo 1.1*, 2010

Ein Ausblick skizziert einige der weiteren Möglichkeiten, die sich durch den DSL-Ansatz mit Parser-Kombinatoren oder durch den Einsatz der LanguageBox aus Helvetia ergeben.

2 *Harel-Statecharts*

Ein endlicher Automat oder eine Zustandsmaschine modelliert Verhalten mit Hilfe von endlich vielen Zuständen und Ereignissen, die Zustandsübergänge und Aktionen auslösen. Obwohl endlich, ist die Zahl der verschiedenen Zustände eines Programms i.A. sehr hoch, weshalb Zustandsmaschinen für die Implementierung von Verhalten in der Praxis keine weite Verbreitung gefunden haben.

Hierarchische Zustandsmaschinen werden im Folgenden *Statecharts* genannt

Diesem Umstand half Harel[16] 1987 ab, als er vorschlug, Verhalten mit Hilfe von *hierarchischen* Zustandsmaschinen zu modellieren. Heute sind die UML-Zustandsdiagramme unter den

[16] Harel, D.: *Statecharts: A visual formalism for complex systems*. Sci. Comput. Program., 8(3):231–274, 1987

Programmierern bekannter (uns interessieren hier nur die BehaviorStateMachines[17]). Wir geben deshalb einen kurzen Abriss der Harel-Statecharts.

David Harel führte mehrere – grafische – Notationselemente ein, um die Komplexität großer Systeme mittels endlicher Automaten handhabbar zu machen:

- Eine Hierarchie von Automaten. Der Zustand eines Automaten kann selbst ein vollständiger Automat mit eigenen Unterzuständen sein. Harel nennt dies XOR-Komposition. Damit sind Ebenen einer Zustandshierarchie definiert.
- Einen Default-Startzustand für die Automaten. Alternativ können Unterzustände direkt angesprungen werden.
- Orthogonale Automaten. Harel nennt dies AND-Komposition. Die Übergänge orthogonaler Automaten sind unabhängig, können auch parallel erfolgen, niemals jedoch zwischen den orthogonalen Automaten.
- Transitionen über verschiedene Ebenen der Zustandshierarchie. Dies impliziert auch das Verlassen und Betreten von Oberzuständen.
- Entry-, Exit- und Throughout-Aktionen von Zuständen. Diese geben Aktionen an, die beim Betreten, Verlassen bzw. vom Betreten bis zum Verlassen des Zustand ausgeführt werden.
- Aktionen, die von einem Übergang getriggert werden.
- History-Marker[18] für einen Automaten[19]. Dieser bestimmt, dass beim Wiedereintritt derjenige direkte Unterzustand wieder einzunehmen ist, der vor dem Verlassen zuletzt betreten war.
- Selection-Marker[20] an einem Ereignis, der abhängig von der Art des Ereignisses für Zustandsübergänge in verschiedene Zielzustände sorgt. Der Selection-Marker erlaubt eine Zusammenfassung mehrerer Ereignisse. Diese Zusammenfassung wiederum erlaubt eine abstrahierte grafische Notation.
- Condition-Marker[21] an einem Ereignis, der abhängig von einer Bedingung für Zustandsübergänge in verschiedene Zielzustände sorgt. Der Condition-Marker ist eine Zusammenfassung mehrerer Ereignis/Guard-Tupel, die durch die Bedingung unterschieden sind. Diese Zusammenfassung erlaubt ebenfalls eine abstrahierte grafische Notation.
- Delays und Timeouts spezifizieren, dass der Zustand erst nach Ablauf des Delays verlassen werden kann (i.e. eine zeitbezogene Guard) oder nach Ablauf des Timeout automatisch verlassen wird.

[17] *UML 2.2 Superstructure Specification*. Techn. Ber., Object Management Group (OMG), February 2009

Die Unterschiede zu den UML-Diagrammen für BehaviorStateMachines werden jeweils in Randnoten benannt.

Die UML nennt sie *regions*.

[18] Die UML kennt zusätzlich das *deep history* Konstrukt, mit dem eine speziell strukturierte Historie von Unterzuständen für den Wiedereintritt spezifiziert werden kann.

[19] Ein Zustand wird zum Automaten, wenn er Unterzustände enthält

[20] Die UML benutzt dafür den Pseudozustand *choice*.

[21] Die UML benutzt dafür den Pseudozustand *junction*.

Die UML kennt darüber hinaus noch einige andere Notationselemente, z. B. die Pseudozustände *initial state* und *final state*, die nicht wesentlich sind. Ein Harel Statechart ist ein Zustand, der (beim UML-Übergang auf den *initial state*) einfach betreten wird und (beim UML-Übergang auf den *final state*) einfach verlassen wird.

3 *PetitParser*

Der PetitParser[22] ist ein mächtiges dynamisches Parser-Framework für die mühelose Implementierung von DSLs. Es baut auf Forschungsarbeiten der letzten fünfzehn Jahre auf:

- Das Prinzip der Parsing Expression Grammars lässt uns die

[22] Renggli, L., S. Ducasse, T. Gîrba und O. Nierstrasz: *Practical Dynamic Grammars for Dynamic Languages*. In: *4th Workshop on Dynamic Languages and Applications (DYLA 2010)*. ACM, 2010

Grammatik ausgehend von der Syntax eindeutig definieren und erlaubt auch nicht-kontextfreie Grammatiken.
- Scannerlose Parser können den vollen lexikalischen Kontext zur Erkennung nutzen.
- Parser-Kombination (im funktionalen Stil) erlaubt, einen Parser aus Bausteinen zusammenzusetzen.
- Die Technik der Packrat-Parser[23] macht das ganze effizient genug.

Der PetitParser lebt in einigen Smalltalk-Dialekten, darunter Pharo[24], in dem die hier berichteten Implementierungen erfolgten. Mit dem PetitParser fällt es leicht, ausgehend von der Syntax einer neuen DSL einen effizienten Parser zu schreiben. Zu den Grammatik-Elementen der DSL werden unmittelbar die einzelnen zu kombinierenden Parser erzeugt und gemäß der Grammatik kombiniert. Der Parser erzeugt aus einem korrekten DSL-Text den abstrakten Syntaxbaum (AST) zu diesem DSL-Text.

In diesem Syntaxbaum lassen sich Ersetzungen vornehmen, i.e. Produktionen hinzufügen. Mit solchen Produktionen lässt sich die Laufzeitumgebung der DSL erzeugen. Dies geschieht isoliert für jeden einzelnen Parser. Produktionen, die in der Hierarchie des AST an einer Stelle erfolgten, lassen sich in einer anderen Produktion oberhalb davon nutzen. Auf diese Art lassen sich auch die Produktionen modular formulieren.

Wenn man die Kombination der Parser in einer einzigen Methode unterbringt, erhält man die Skript-Form des Parsers für die DSL. Alternativ werden die einzelnen Parser zu Methoden in speziellen Unterklassen des PetitParser. Dann lassen sich die Grammatik und die Produktionen auf verschiedene Klassen aufteilen und so besser voneinander trennen.

Die AuDSL ist orthogonal zum Smalltalk-Programm selbst. Der Zusammenhang mit der in Smalltalk geschriebenen Anwendung manifestiert sich im API und wird damit zur Laufzeit im Debugger sichtbar. Dadurch entsteht das Problem, dass im Debugger evtl. unerwünschte Laufzeitstrukturen aus der AuDSL aufscheinen. Ein zweites Problem ist, dass der Anwendungsprogrammierer beim Debugging Struktur- und Zustandsinformationen braucht, die zur Spezifikation mit der AuDSL passen. Solche Struktur- und Zustandsinformationen müssen zur Laufzeit geliefert werden.

PetitParser und Helvetia. Neben einer – wie in diesem Papier diskutierten – zur Wirtssprache orthogonalen DSL sind auch vollständig eingebettete Spracherweiterungen denkbar. Deshalb sei hier ein Seitenblick auf Helvetia[25] gestattet, das auf den Technologien von Smalltalk und dem PetitParser aufsetzt und damit eine LanguageBox[26] in der Wirtssprache (hier: Smalltalk) definiert, in der – ähnlich wie mit Lisp-Makros – Spracherweiterungen definiert werden können. Diese Spracherweiterungen werden nahtlos im Debugger sichtbar und ebenso zugänglich wie der Code der Wirtssprache.

[23] Ford, B.: *Parsing expression grammars: a recognition-based syntactic foundation*. In: *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, S. 111–122, New York, NY, USA, 2004. ACM

[24] Marcus Denker, Stéphane Ducasse, A. L.: *http://www.pharo-project.org/home*, 07 2008

Der PetitParser ist nicht das einzige Framework zur Implementierung von DSLs mit Parser Kombinatoren. In der Programmiersprache Scala gibt es ebenfalls ein Framework mit Parser Kombinatoren, das es erlaubt, ähnliche Parser-Skripte zu schreiben:

Piessens, F., A. Moors und M. Odersky: *Celestijnenlaan 200A - B-3001 Heverlee (Belgium) Parser Combinators in Scala Adriaan Moors Frank Piessens*. Techn. Ber., 2008

[25] Renggli, L., M. Denker und O. Nierstrasz: *Language Boxes: Bending the Host Language with Modular Language Changes*. In: *Software Language Engineering: Second International Conference, SLE 2009, Denver, Colorado, October 5-6, 2009*, Bd. 5969 d. Reihe *LNCS*, S. 274–293. Springer, 2009

[26] Renggli, L., T. Gîrba und O. Nierstrasz: *Embedding Languages Without Breaking Tools*. In: *ECOOP 2010: Proceedings of the 24th European Conference on Object-Oriented Programming*. Springer-Verlag, 2010

Für unsere AuDSL brauchen wir eine solche LanguageBox nicht, für DSLs, die sich vollständig in die Wirtssprache integrieren, stellen sie ein aussichtsreiches Werkzeug dar.

4 AuDSL für einen Harel-Statechart-Automaten

DIE AUDSL IST SCHLANK. Die textuelle AuDSL entspricht im wesentlichen der grafischen Spezifikationssprache, die Harel für seine Statecharts entworfen hatte. Sie kann dabei auf einige Eigenschaften der Statecharts verzichten, wie z. B. die Selections und Conditions[27], auch wird eine Throughout-Aktion durch entsprechende OnEntry- und OnExit-Aktionen definiert. Bis auf Delays und Timeouts ist sie äquivalent zu der grafischen Spezifikationssprache der Statecharts.

Anderseits ist die AuDSL schlanker als die grafische Spezifikationssprache der Statecharts, weil einige Eigenschaften textuell leichter zu formulieren sind als grafisch. So erfordert z. B. der Default-Startzustand eines Automaten keine eigene Notation, die Reihenfolge im Text reicht aus. Die AuDSL ist folgerichtig auch viel schlanker als die UML-Zustandsdiagramme.

[27] vgl. Kapitel 2

DIE MASCHINE UND DIE SEMANTIK DER AUDSL. Die Ausführung eines mit der AuDSL spezifizierten Statecharts übernimmt eine Maschine, die im Übersetzungsschritt aus der Semantik,[28] dem Laufzeit-Framework, erzeugt wird. Die Syntax und die Semantik der AuDSL hängen eng zusammen, wie wir auch bei der Implementierung mehrerer Varianten der AuDSL erfahren haben. Im folgenden wird lediglich eine dieser Varianten vorgestellt. Einige Unterschiede zu den anderen Varianten werden im Kapitel 5 diskutiert.

[28] Die Semantik der AuDSL besteht aus vier Smalltalk-Klassen: AuState, AuRegion, AuXORState und AuTransition

DIE SCHNITTSTELLE ZWISCHEN ANWENDUNG UND MASCHINE. Die Anwendung selbst braucht eine Schnittstelle zu i) dem Compiler der AuDSL und zu ii) der ausführenden Maschine. Diese Schnittstelle wird in einer speziellen Klasse definiert, von der die Anwendung ableitet[29].

[29] Eine kleine Anwendung wird möglicherweise direkt eine Unterklasse von `AuModel` (s. Abbildung 1) sein, eine größere Anwendung besitzt eine `AuModel`-Unterklasse

IN DEN FOLGENDEN ABSCHNITTEN wird zunächst die Syntax der AuDSL beschrieben, dann die Semantik (das Laufzeit-Framework), zuletzt die Klasse, die die Schnittstelle zur Anwendung bereitstellt. Abbildung 1 beschreibt diese drei Bestandteile und ihre Codegrößen in Smalltalk.

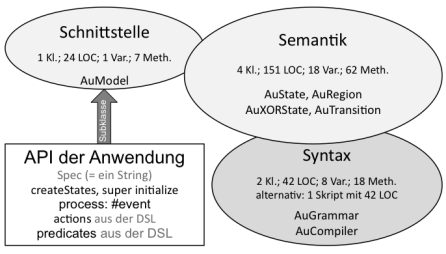

Abbildung 1: Die drei Bestandteile der AuDSL und ihrer Laufzeitumgebung

4.1 Die Syntax der AuDSL

EIGENSCHAFTEN DER AUDSL. Die AuDSL erlaubt die Spezifikation eines Zustandes, der selbst ein Automat ist, wenn er seinerseits

wiederum Zustände enthält. Die so spezifizierten Zustände formen einen Baum. Der oberste Zustand, die Wurzel des Baums, repräsentiert somit den ganzen Statechart. Im Folgenden nennen wir den Oberzustand eines Zustandes *Elter*, die Unterzustände *Kinder*. Die AuDSL kennt zwei Varianten von Zuständen: *state* und *region*. Ein *state* darf (mindestens zwei) Kinder enthalten, Harel nennt ihn XOR-Zustand. Eine *region* muss mindestens zwei orthogonale Kinder enthalten, Harel nennt sie AND-Zustand.

Der Name *region* ist der UML entlehnt.

Abbildung 2 zeigt die Syntax der AuDSL mit den vom Petit-Parser vordefinierten Parsern *letter* und *word*. Jeder Zustand hat einen Namen und optional *onEntry*-Aktionen, *onExit*-Aktionen und *transitions*. Ein *state* kann noch einen *history*-Marker enthalten. Die Spezifikation jedes Zustandes wird in Klammern eingeschlossen.

Die *onEntry* oder *onExit*-Aktionen werden als Liste von Namen in eckigen Klammern geschrieben. Diese Namen sind Methodensignaturen der Anwendungsklasse, d.h. derjenigen Domänenklasse, die den Statechart definiert. Die Aktionsmethoden sind parameterlose Prozeduren, die ausschließlich Nebeneffekte auslösen.

```
AuDSL      =  state
state      = '(' (regiontype|xortype)
                 {onentry} {onexit}
                 {transition+} {state+}
              ')'
regiontype = 'r:' identifier
xortype    = identifier {history}
history    = 'history'
onentry    = 'onEntry:'
             '[' identifier+ ']'
onexit     = 'onExit:'
             '[' identifier+ ']'
transition = identifier
             { '[' identifier+ ']' }
             '->' identifier
             { '[' identifier+ ']' }
identifier = letter word*
```

Abbildung 2: Die AuDSL in BNF

Eine *transition* wird geschrieben als *event [guards] -> target [actions]*. Dabei bezeichnet *event* das den Übergang auslösende Ereignis und *target* den Namen des Zielzustandes. *[guards]* ist eine Liste von Methodensignaturen der Anwendungsklasse, diese Methoden sind Prädikate, die zu wahr oder falsch evaluieren, während *[actions]* wiederum eine Liste von Aktionsmethoden bezeichnet. Alle Namen fangen mit einem Buchstaben an und können neben Buchstaben auch Ziffern enthalten.

Harels *selections* und *conditions* werden in der AuDSL als verschiedene *events* bzw. verschiedene *guards* zum selben *event* formuliert. Dies ist keine Einschränkung, denn *selections* und *conditions* dienen nur dazu, die grafische Spezifikation übersichtlicher zu machen, in der textuellen Spezifikation der AuDSL bieten sie keinen Vorteil.

Delays und Timeouts werden mit der AuDSL in der Anwendungsklasse beschrieben, Delays als Guards (mit Zeitbedingung) und Timeouts als zeitgesteuerte Ereignisse.

4.2 Die Semantik der AuDSL

Der Compiler erzeugt, gesteuert von dem AuDSL-Text, aus der Semantik (i.e. aus dem Laufzeit-Frameworks) den konkreten Statechart – die Maschine. Das Laufzeit-Framework besteht aus vier Klassen für Zustände (`AuState`, `AuXorState`, `AuRegion`) und Übergänge (`AuTransition`).

AuState, AuXorState und AuRegion. Die abstrakte Klasse `AuState` definiert das wesentliche Verhalten der Automaten oder Zustände und damit auch des Statecharts. Die Unterklasse `AuXORState` definiert einen *state*, i.e. mit Unterzuständen einen Automaten, ohne Unterzustände einen einfachen Zustand. Die Unterklasse `AuRegion` definiert eine *region*, i.e. immer einen Automaten mit orthogonalen Unterzuständen. Im folgenden wird auch ein Automat immer Zustand genannt.

Ein Zustand ist entweder *betreten* oder *verlassen* (*nicht betreten*). Mit dem Betreten werden die *onEntry*-Aktionen ausgeführt:

```
AuState>>enter
   entryActions do: [:action| model perform: action].
```

model verweist auf das `AuModel` der Anwendung

```
   entered := true.
   self enterChildren
```

Mit dem Verlassen werden die *onExit*-Aktionen ausgeführt.

```
AuState>>exit
   self rememberHistory; exitChildren.
   exitActions do: [:action| model perform: action].
   entered := false
```

Das Betreten und Verlassen von Zuständen unterliegt zeitlichen Reihenfolgen. Kinder werden nach dem Zustand selbst betreten, aber vor dem Zustand selbst verlassen. In einer *region* werden auch alle Kinder betreten:

```
AuRegion>>enterChildren
   children do: [:child| child enter]
```

In einem *state* hängt es von der *history*-Marke und der tatsächlichen Historie ab, welches Kind betreten wird:

```
AuXorState>>enterChildren
   "regarding history"
   (firstChild isNil or: [self hasEnteredChild]) ifTrue: [^self].
   (history and: [historyChild notNil])
      ifTrue: [historyChild enter] ifFalse: [firstChild enter]
```

Bei einem Zustandsübergang wird auch der Elter eines Zustandes verlassen, wenn dieser Elter im Zielzustand *nicht betreten* ist. Entsprechend werden ggf. die Eltern des Zielzustandes betreten. Bei jedem Übergang gibt es einen gemeinsamen Elter, der nicht mehr verlassen wird, das ist der *commonAncestor*, der übrigens keine *region* sein darf:

```
AuState>>switchTo: anotherState
   commonAncestor := elder commonAncestorWith: anotherState.
   commonAncestor isXor ifFalse: [^self log: 'no transition ... ]
   self exitUpTo: commonAncestor.
   anotherState enterUpTo: commonAncestor
```

Die Eltern des Zielzustandes werden nach dem Zielzustand selbst betreten, um implizites Betreten mit oder ohne *history* auszuschließen:

```
AuState>>exitUpTo: aState
   "stop exiting at the common ancestor aState"
   (self = aState) ifFalse: [self exit. self elder exitUpTo: aState]
AuState>>enterUpTo: aState
   "stop entering at the common ancestor aState"
   (self = aState) ifFalse: [self enter. self elder enterUpTo: aState ]
```

Die Anwendung sendet Ereignisse (*events*) an das Statechart, die Transitionen auslösen. Es folgt eine Suche nach dem tiefsten Zustand, der das Ereignis verarbeitet. Jeder Zustand, der ein Event

empfängt, beauftragt zunächst seine aktiven Kinder (bei einem *state*: höchstens ein Kind, bei einer *region*: alle Kinder) mit der Verarbeitung des Events, ehe er es selbst verarbeitet. Wenn es bereits verarbeitet ist, geschieht im Zustand nichts mehr:

```
AuState>>process: anEvent
   done := false.
   children do: [ :c | c isEntered ifTrue: [done := c process: anEvent] ].
   done ifTrue: [^true].
   ^done := self execute: anEvent
AuState>>execute: anEvent
   (t := self transitionFor: anEvent) ifNil: [^false].
   self switchTo: t target.
   t executeActions.
   ^true
```

Zur Verarbeitung eines Events prüft der Zustand zunächst, ob es für das Event eine Transition gibt, wenn ja, ob alle Guards erfüllt sind. Für jeden Guard wird das entsprechende Prädikat der Anwendung aufgerufen. Die erste Transition[30], für die alle Guards erfüllt sind, wird ausgeführt. Mit der Transition werden auch alle mit ihr verbundenen Aktionen in der Domäne ausgeführt. Zustandsübergänge zwischen verschiedenen *regions* lassen sich mit der AuDSL (Abschnitt 4.1) zwar spezifizieren, sie sind jedoch nicht erlaubt und werden deshalb nicht ausgeführt:

[30] Wenn mehrere Transitionen für dasselbe Event spezifiziert sind, ist die Auswahl der Transition nicht determiniert.

```
AuState>>transitionFor: anEvent
   "answer the first transition for anEvent. log the reason if none found"
   (candidates := transitions select: [:t| t event = anEvent])
      ifEmpty: [^self log: 'no transition ... ]
   (candidates := candidates select: [:e| e guardsSucceedOn: anEvent])
      ifEmpty: [^self log: 'guards failed ... ]
   self log: ('!',candidates size printString,'! transition ... ]
   ^candidates anyOne
```

Das API. Die Maschine stellt der Anwendung ein simples API aus drei Elementen zur Verfügung.

Der ganze Statechart wird vom Wurzelzustand der hierarchischen Zustandsmaschine repräsentiert.

- `process: #event` wird von der Anwendung an den Statechart gesendet.
- Die Anwendung stellt der Maschine parameterlose Callback-Methoden zur Verfügung:
  - mit der AuDSL spezifizierte Prädikate (*guards*) antworten mit `true` oder `false`;
  - mit der AuDSL spezifizierte Aktionen (onEntry-, onExit- und an Transitionen) führen Seiteneffekte aus.

Die Anwendungsoberklasse. Voraussetzung für den Zugriff auf das API ist eine geerbte Struktur von der Klasse `AuModel`, die einen oder mehrere Statecharts hält. Für den Zugriff wird je Statechart eine Methode mit dem Namen des Wurzelzustands als Signatur generiert (das ist besonders praktisch für Unit-Tests).

Der Zugriff auf einen beliebigen Unterzustand gelingt durch Senden von `at:#name`, wobei `#name` der im Statechart eindeutige Name des Unterzustandes ist. Wenn `app` das Anwendungsobjekt bezeichnet, dass einen Statechart `root` hält, dann gelingt der Zugriff auf den Zustand mit Namen `state-1-1` einfach mit `app root at: state-1-1`.

Die Anwendung enthält mindestens eine Unterklasse von `AuModel`, in der sie folgendes implementiert:

- Eine Spezifikationsmethode je Statechart, die den String mit der Spezifikation der Statecharts zurück gibt. Die Signatur der Spezifikationsmethode besteht aus dem Namen des Statecharts, gefolgt von dem String `'Spec'`.
- In der Methode `createStates` wird je Statechart `self create:#name` aufgerufen wird, wobei `#name` der Name des Statecharts ist.[31]
- In der eigenen `initialize`-Methode wird `super initialize` aufgerufen.

[31] Am Ende von Abschnitt 4.3 wird dies noch näher betrachtet.

Das Debuggen einer Anwendung mit der AuDSL erfordert im Debugger Ansichten der Statecharts, die von der Maschine selbst abstrahieren und zusammen mit der Spezifikation eines Statecharts allein verständlich sind. Im vorliegenden Fall sind das

- eine kompakte, lesbare Darstellung des Gesamtzustands des Statecharts und
- Logeinträge nach dem Empfang und Dispatch eines Events.

Die Struktur der Maschine selbst wird im Debugger nicht ausgeblendet, aber durch eine kompakte Zustandsdarstellung ergänzt. Dies leisten die Methoden:

```
AuState>>printOn: aStream
      Stream nextPutAll: ('AuState: ').
      self printOn: aStream indent: 1
AuRegion>>printOn: aStream indent: anInt
      Stream nextPutAll: self name1, '.'.
      children do: [:c | c isEntered ifTrue:
            [aStream cr; tab: anInt. c printOn: aStream indent: (anInt+1)]]
AuXorState>>printOn: aStream indent: anInt
      aStream nextPutAll: self name1, '.'.
      children do: [:c | c isEntered ifTrue: [c printOn: aStream indent: anInt]]
```

Dies sorgt für einen zeilenweisen Ausdruck der jeweiligen Ober- und Unterzustände. Eine *region* macht dabei für jeden ihrer orthogonalen Zustände eine eigene Zeile auf.[32]

[32] Ein Beispiel findet sich im Abschnitt 4.4.

Jeder Zustand hat ein `log`, das es erlaubt, im Debugger die Logeinträge zu inspizieren.

4.3 Die Implementierung der AuDSL als PetitParser-Skript

Die AuDSL als Programm ähnelt BNF. Der PetitParser suggeriert einen Programmierstil, in dem die Grammatik und die Produktionen als einzelne Methoden einer Grammatik (als Unterklasse

von `PPCompositeParser`) und weiterer Unterklassen z. B. für die Produktionen implementiert werden.

Anstelle dieses PetitParser-Stils wird hier eine ebenfalls mögliche Skript-Form der Grammatik und der Produktionen benutzt, die es dem Leser vor allem erlaubt, das Beispiel praktisch 1:1 auf die Parser Combinators von Scala[33] zu übertragen. Dabei wird jeder einzelne der zu kombinierenden Parser eine lokale Variable des Skripts. Für das Verständnis ist wichtig, dass die Namen der einzelnen Parser sprechend sind. Die Grammatik wird für das Skript in umgekehrter Reihenfolge hingeschrieben, damit die zu kombinierenden Parser bei der Kombination bereits bekannt sind. `state` wird zunächst als `PPUnresolvedParser` erzeugt und später mit `def:` zugewiesen. Mit diesem Trick gelingt auch die Definition der rekursiven Struktur der `states`. Das PetitParser-Skript lautet so (noch ohne die erforderlichen Produktionen):

[33] Odersky, M., L. Spoon und B. Venners: *Programming in Scala*. Artima, 2008

Im PetitParser-Skript entspricht *plus* dem + der Grammatik und *star* dem * der Grammatik. Die Methode *trim* erzeugt einen Parser, der Whitespace konsumiert.

```
state       := PPUnresolvedParser new.
identifier  := (#letter asParser , #word asParser star) token trim.
transition  := identifier, ($[ asParser, identifier plus, $] asParser) trim optional,
            '->' asParser, identifier, ($[ asParser, identifier plus, $] asParser) trim optional.
onexit      := 'onExit:' asParser trim, $[ asParser,  identifier plus, $] asParser.
onentry     := 'onEntry:' asParser trim, $[ asParser, identifier plus, $] asParser.
history     := 'history' asParser trim.
xortype     := identifier, history optional.
regiontype  := 'r:' asParser, identifier.
state       def: ( $( asParser, (regiontype/xortype), onentry optional, onexit optional,
            transition plus optional, state plus optional, $) asParser trim).
auDSL       := state trim end.
^auDSL parse: aSpec onError: [:msg :pos|
            ^self error: msg printString, ' at: ', pos printString, ' in ', aSpec]
```

Dieses Skript gestattet bereits, eine Spezifikation zu überprüfen. Es baut den abstrakten Syntaxbaum (AST), der auch zurückgegeben wird und inspiziert werden kann. Es wirft einen Fehler mit derjenigen Position im Spezifikations-String, an dem der Input nicht weiter geparst werden kann.

Die erforderlichen Produktionen bestehen in der Ergänzung mit den folgenden Zeilen, die direkt hinter die einzelnen Parser-Definitionen geschrieben werden können:

```
transition  := transition  ==> [:nodes| AuTransition new
                target: (nodes at: 4) value asSymbol
                event: nodes first value asSymbol
                guards: (nodes second
                   ifNil: [Array new]
                   ifNotNil: [nodes second second collect: [:g| g value asSymbol]])
                actions: ((nodes at: 5)
                   ifNil: [Array new]
                   ifNotNil: [(nodes at: 5) second collect: [:g| g value asSymbol]])].
onexit      := onexit      ==> [:nodes| (nodes at: 3) collect: [:a| a value asSymbol]].
onentry     := onentry     ==> [:nodes| (nodes at: 3) collect: [:a| a value asSymbol]].
```

```
regiontype  := regiontype  ==> [:nodes| Au3Region new: nodes second value].
xortype     := xortype     ==> [:nodes| | istate |
                istate := Au3XorState new: nodes first value.
                istate history: (nodes second ifNil: [false] ifNotNil: [true]).
                istate].
state       def: ( ... )   ==> [:nodes| | istate itransitions isubstates|
                istate := nodes second value.
                istate entryActions: ((nodes at: 3) ifNil:[Array new]).
                istate exitActions: ((nodes at: 4) ifNil:[Array new]).
                itransitions := (nodes at: 5) ifNil: [Array new].
                itransitions do: [:t|  istate addTransition: t].
                isubstates := (nodes at: 6) ifNil: [Array new].
                (istate isXor and: [isubstates notEmpty]) ifTrue:
                   [istate firstChild: isubstates first].
                isubstates do:  [:s| |child|
                             child := s value.
                             istate addChild: child.
                             child elder: istate].
                istate].
```

Für jeden einzelnen Parser ist der Teilbaum des AST, den er repräsentiert, über die Blockvariable *:nodes* zugänglich. Im Produktionsblock wird der entsprechende Teilbaum durch das Ergebnis des Blockes ersetzt; z. B. wird der Teilbaum *transition* durch eine `AuTransition` ersetzt, in der gleich *target*, *event* und die zuvor berechneten *guards* eingesetzt werden. Die *targets* der *transitions* sind allerdings noch Namen. Sie können erst durch die Zustände selbst ersetzt werden, wenn der Statechart vollständig ist. Deshalb brauchen wir die letzte Produktion an der Wurzel des AST, an der die Statemachine bereits zusammengebaut ist. Hier können wir jetzt

- globale Ersetzungen vornehmen wie das Ersetzen der Target-Namen durch die Targets selbst und
- globale Prüfungen durchführen wie die Prüfungen auf illegale Transitionen oder undefinierte Targets.

```
auDSL := auDSL ==> [ :nodes | | istate errors |
                errors := ''.
                istate := nodes.
                istate replaceTargetNames.
                errors := errors, istate illegalTransitions.
                errors := errors,  istate unknownTargets.
                errors ifNotEmpty: [istate  := PPFailure message: errors at: 0].
                istate ].
```

Das so zusammengesetzte Skript steht in der Methode

```
Au3Compiler>>createSmFrom: spec
```

die den AST liefert, in dem alle Produktionen ausgeführt worden sind, oder einen Fehler.

Die Anwendung ruft mit `self create: aStateName` den AuDSL-Compiler auf. In der `installFrom:`-Methode von `AuModel` wird

- im Statechart das `AuModel` gesetzt,
- das Statechart selbst in die Variable `statechart` eingehängt und
- die Zugriffsmethode auf das Statechart erzeugt.

```
AuModel>>create: aStateName
      self installFrom: (self selectorFrom: aStateName)
AuModel>>installFrom: aSpecSelector
      spec := (self perform: aSpecSelector).
      sm := Au3Compiler createSmFrom: spec.
      sm  model: self.
      self statechart at: (sm name1) put: sm.
      self writeAccessMethod: sm name1
```

Im PetitParser-Programmierstil sind die gleiche Grammatik und die gleichen Produktionen in zwei Klassen AuGrammar und AuCompiler in Form von 18 einzelnen Methoden implementiert (Jede Zeile des Skripts wird zu einer eigenen Methode).

4.4 Beispiel: Mikrowellen-Ofen

Die AuDSL-Spezifikation

```
(r: oven
    (heater
        (idle
            onEntry:     [enableTimeSetting ]
            onExit:      [disableTimeSetting ]
            start [doorIsClosed ] -> cooking)
        (cooking
            onEntry:     [startTimer ]
            onExit:      [stopTimer]
            open ->  idle
            finished -> idle))
    (door
        history
        (open
            close -> closed)
        (closed
            open -> open)))
```

erzeugt den Statechart für den `AuMicrowaveOven`. Dieser Statechart hat an der Wurzel eine *region* mit den beiden orthogonalen Kindern *heater* und *door*. Beim Betreten des *oven* werden sowohl *heater* als auch *door* betreten. Der *heater* ist dann *idle*, die *door* zunächst *open*. Die Anwendung prüft - noch während der Initialisierung - nach dem Betreten den Türsensor und schickt dementsprechend entweder ein *close* oder ein *open* an den *oven*.

```
AuMicrowaveOven>>checkDoor
    ^doorClosed ifTrue: [self closeDoor] ifFalse: [self openDoor]
AuMicrowaveOven>>closeDoor
    self oven process: #close
```

```
AuMicrowaveOven>>openDoor
    self oven process: #open
```

Der Zustand nach dem Betreten ist wie folgt im Debugger sichtbar:

```
AuState: oven.
      heater.idle.
      door.open.
```

und nach dem Check des Türsensors

```
AuState: oven.
      heater.idle.
      door.closed.
```

`AuMicrowaveOven` implementiert folgende Methoden :

- Prädikate für die Guards
  - doorIsClosed
- Prozeduren mit Nebeneffekten für die Aktionen
  - enableTimeSetting
  - disableTimeSetting
  - startTimer
  - stopTimer

Außerdem muss `AuMicrowaveOven` in seiner Initialisierung `super initialize` und in der Methode `createStates` `self create: #oven` aufrufen.
Ein Event wird mit `self oven process: #event` an den Statechart geschickt. Solche Aufrufe verpackt `AuMicrowaveOven` in einige

- Convenience-Methoden
  - openDoor
  - closeDoor
  - start
  - finish

Das Beispielprogramm

```
mysm := AuMicrowaveOven new.    Transcript open; show: mysm oven content; cr.
mysm openDoor.                  Transcript show: mysm oven content; cr.
mysm start.                     Transcript show: mysm oven content; cr.
mysm closeDoor.                 Transcript show: mysm oven content; cr.
mysm start.                     Transcript show: mysm oven content; cr.
mysm finish.                    Transcript show: mysm oven content; cr.
```

erzeugt den folgenden Ausdruck im Transcript:

```
AuState: oven.
      heater.idle.
      door.closed.
AuState: oven.
      heater.idle.
      door.open.
AuState: oven.
      heater.idle.
      door.open.
AuState: oven.
      heater.idle.
      door.closed.
```

Hier erfolgt kein Zustandswechsel, weil die Tür noch nicht geschlossen wurde.

```
AuState: oven.
      heater.cooking.
      door.closed.
AuState: oven.
      heater.idle.
      door.closed.
```

Die Zustände des Statechart *oven* sind mit einem Blick erfassbar.

4.5 Beispiel: Zustandsmaschine für einen Portalhubwagen

Um die Tragfähigkeit des DSL-Ansatzes zu erproben, wurde eine sehr umfangreiche Zustandsmaschine mit der AuDSL beschrieben.

In den Projekten eines der Autoren wurde die Bediensoftware eines Portalhubwagens von einer in C# geschriebenen Zustandsmaschine gesteuert. Diese Zustandsmaschine war in Java spezifiziert und mit einer UML-ähnlichen Grafik dokumentiert. Sie umfasst zwei Zustandsmaschinen,

Ein Portalhubwagen ist ein Transportfahrzeug für Container auf einem Containerterminal.

- eine sehr kleine für die Twistlocks[34] mit je zwei Ereignissen und Zuständen (jeweils auf und zu)
- und eine sehr große für Tastatureingaben, Anzeige und Server-Kommunikation mit 83 Zuständen in 6 Hierarchie-Ebenen und 203 Ereignissen.

[34] Twistlocks sind die Verriegelungen, mit denen der Portalhubwagen einen Container an seinen genormten Eckbeschlägen (corner castings) greift. Ein Teil des Twistlocks wird dabei um 90° verdreht und stellt so eine formschlüssige Verbindung her.

Die Spezifikation der großen Zustandsmaschine umfasst 348 Zeilen AuDSL (im Wesentlichen je eine Zeile für jeden Zustand, seine onEntry- und onExit-Aktionen und jedes Ereignis). Das zugehörige Simulationsprogramm enthält neben den Prädikaten für die Guards und den Aktionen der Zustandsmaschine noch einige Hilfsmethoden für Aktionen der Anwendung. Diese Hilfsmethoden sorgen für bessere Lesbarkeit des zugehörige Simulationsskripts, das lediglich Ereignisse in vorgefertigter Reihenfolge an den Zustandsautomaten sendet.

5 Lessons learned

5.1 DSLs fast mühelos durch Parser Kombinatoren

Parser-Kombinatoren, wie sie der PetitParser bietet, ermöglichen auch dem im Compilerbau ungeübten Programmierer, eine DSL ohne Klimmzüge oder Umwege auf die einfachste Art zu programmieren. Am Anfang steht der Entwurf der Sprache selbst, in unserem Fall die Konfigurationssprache für einen Harel-Statechart. Aus den einzelnen Sprachelementen folgt unmittelbar die Grammatik für die DSL. Diese Grammatik lässt sich mit Hilfe der Parser-Kombinatoren unmittelbar in einen Parser umschreiben. Dies ergibt ein Parse-Skript, aus dem die Grammatik nach wie vor direkt ablesbar ist, allerdings sozusagen rückwärts. Im vorliegenden Fall wurde die Grammatik aus Abschnitt 4.1 zu dem Parse-Skript in Abschnitt 4.3 umgeschrieben.

Das Einfügen der Produktionen, die die ausführende Maschine

erzeugen müssen, setzt zunächst voraus, dass die Semantik, das Laufzeit-Framework, selbst entworfen wurde. Solange dies noch nicht geschehen ist, lässt sich der Parser mit Textausgaben an Stelle der Produktionen testen. Das Erzeugen der Maschine lässt sich geschickt auf die Hierarchie des AST verteilen, wie es in Abschnitt 4.3 geschehen ist. Die Leichtigkeit der Erzeugung hängt von zwei Dingen ab, nämlich von:

- einer angemessenen Wahl der DSL-Syntax; z. B. erleichtern Schlüsselwörter oder -zeichen die eindeutige Zuordnung zu speziellen Parsern.
- einer geschickten Zuordnung der einzelnen Parser zu den Syntaxelementen. Dies vereinfacht den Zugriff auf den AST und die Produktionen.

Für die Grammatik selbst und damit auch für den Parser gilt ein grundlegendes Prinzip der Programmierung: Sorgfältige Wahl der Namen. Ein zweites Prinzip ist das Pattern *ComposedMethod*, benennbare Dinge so zu isolieren, dass Kommentare entbehrlich sind[35]. Mit diesem Prinzip könnten wir die vorgestellte Grammatik lesbarer gestalten. Statt:

[35] Beck, K.: *Smalltalk: best practice patterns*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997

```
transition = identifier { '['  identifier+ ']' } '->' identifier  { '['  identifier+ ']' }
```

könnten wir schreiben:

```
transition = event { '['  guards ']' } '->' target { '['  actions ']' }
event      = identifier
guards     = identifier+
target     = identifier
actions    = identifier+
```

Dies überträgt sich 1:1 auf den Parser, dessen entsprechende Zeilen dann lauten:

```
transition := event, ($[ asParser, guards, $] asParser) trim optional,
              '->' asParser, target, ($[ asParser, actions, $] asParser) trim optional.
event      := identifier
guards     := identifier plus
target     := identifier.
actions    := identifier plus
```

Der hier vorgestellten Parser erlaubt keine Kommentare. Auch die Kommentare müssen explizit geparst werden. Zum Beispiel parst der Parser

```
comment := ($% asParser,(PPPredicateObjectParser anyExceptAnyOf:'%')star,
            $% asParser) token trim
```

einen Kommentar, der in %-Zeichen eingeschlossen ist. Dieser Parser kann nun in den PPTrimmingParser eingebaut werden,[36] um an jeder Stelle, an der Whitespace stehen darf, auch in %-Zeichen eingeschlossene Kommentare zu erlauben.

[36] Die Autoren danken L. Renggli für diesen Hinweis.

Ein größeres Problem stellt die Rekursivität der Zustände dar. Dadurch, dass die Zustände selbst rekursiv definiert sind, ist ein

Zustand selbst erst fehlerfrei parsbar, wenn alle Unterzustände fehlerfrei parsbar sind. Der Parser meldet folglich einen potentiellen Fehler nur an der Stelle, wo es im obersten Zustand keine Parse-Alternativen mehr gibt. Abhilfe schafft hier, die Zustände separat zu spezifizieren. Wir erlauben deshalb, dass ein scheinbar einfacher Zustand im Anschluss an den Statechart getrennt spezifiziert wird. So wird nicht nur das Spezifizieren eines Statecharts mit der AuDSL erleichtert, sondern der Parser findet auch die Fehler, die bei der Spezifikation dieses Zustandes gemacht werden. Die erforderlichen Änderungen sind moderat:

- Statt eines Zustandes parsen wir beliebig viele nacheinander. Dazu fügen wir in den Parser AuDSL ein *plus*[37] ein.
- Der erste Zustand stellt den Statechart dar. Die Folgezustände setzen wir in den Statechart ein. Dazu fügen wir in die Produktion von AuDSL vor *replaceTargetNames* einen Block ein, der alle Knoten ab dem zweiten (*nodes allButFirst*) durch die bereits erzeugten Zustände ersetzt:

[37] `auDSL := state trim plus end`

```
auDSL := auDSL ==> [:nodes|  | istate errors |
        errors := ''.
        istate := nodes first.
        nodes allButFirst do:
            [:substate| |toBeReplaced|
            toBeReplaced := istate at: substate name1.
            toBeReplaced ifNotNil: [toBeReplaced become: substate]
                ifNil: [errors := errors, 'could not replace ', substate name1, String cr]].
        istate replaceTargetNames.
        errors := errors,  istate illegalTransitions .
        errors := errors,  istate unknownTargets.
        errors ifNotEmpty: [istate  := PPFailure message: errors at: 0].
        istate ].
```

5.2 *Das Laufzeit-Framework sorgfältig programmieren*

Die Semantik der neuen DSL, das Laufzeit-Framework, definiert die Klassen, aus denen die Maschine erzeugt wird. In der AuDSL sind dies `AuState`, `AuRegion`, `AuXorState` und `AuTransition` sowie die Glue-Klasse `AuModel`. Diese Klassen müssen deshalb sehr sorgfältig programmiert und ggf. dokumentiert werden. Auch bei unveränderter Syntax der DSL sind viele verschiedene Semantiken möglich, die Semantik der AuDSL haben wir in Abschnitt 4.1 beschrieben.

Im Laufe dieser Arbeit sind mehrere Implementierungen des Automaten entstanden, die bei gleicher Syntax der AuDSL Unterschiede im Verhalten des Statecharts vorwiesen, sie entsprachen allerdings noch nicht den Harel-Statecharts. Alle Implementierungen des Automaten sind recht klein mit ca. 150 Zeilen Smalltalk Code. Sie durchlaufen 31 Unit-Tests mit ca. 550 LOC, wovon ca. 80 LOC auf die Spezifikationen von 3 Test-Statecharts mit einer Schachtelungstiefe bis zu 6 entfallen.

Ein besonderes Augenmerk muss der Darstellung der inneren

Zustände des Automaten im Debugger gelten. Dies ist für müheloses Debuggen der Anwendung wesentlich, denn Fehler bei der Spezifikation mit einer DSL führen

- entweder zur Fehlermeldung beim Parsen, damit lässt sich die Spezifikation gezielt inspizieren,
- oder dazu, dass nichts geschieht.

Wenn nichts geschieht, ist es wichtig, alle Programmzustände möglichst übersichtlich erfassen zu können. Im Abschnitt 4.4 ist die gewählte textuelle Darstellung der Statemachine-Zustände gezeigt. Diese kompakte textuelle Darstellung ist ausreichend, eine grafische Darstellung ist prinzipiell programmierbar.

Im Allgemeinen erlaubt jede DSL syntaktisch Angaben, die semantisch unzulässig sind, i.e. es gibt keine Fehlermeldung beim Übersetzen, sondern erst bei der Ausführung. Solch eine Laufzeitprüfung ist z. B. die Eindeutigkeit von Übergängen für ein gegebenes Ereignis. Dies kann der Automat nicht statisch überprüfen, da die Eindeutigkeit auch von den Guards der Anwendung abhängt. Generell können viele Prüfungen entweder zur Übersetzungs- oder zur Laufzeit durchgeführt werden. Ein Beispiel dafür in der AuDSL ist eine *transition* zwischen orthogonalen Zuständen einer *region*. Syntaktisch wird solch ein Übergang durch die AuDSL erlaubt. Die erforderlichen Prüfungen stecken in der letzten Produktion des AuDSL-Compilers. In früheren Versionen der AuDSL wurde diese Prüfung zur Laufzeit durchgeführt.

Zur Laufzeit gibt es für jeden Zustand Log-Einträge, die im Debugger sichtbar sind:

- das letzte empfangene Ereignis, das normal verarbeitet wurde, oder
- das Fehlen eines Übergangs für dieses Ereignis, oder
- dass ein oder mehrere Übergänge gefunden wurden (mehr als ein möglicher Übergang für das Ereignis deutet auf einen Spezifikationsfehler).

5.3 Wechselwirkung zwischen DSL, Parser und Automat

Wenn die Sprache selbst verändert wird, ändert sich der Parser unweigerlich mit. Auch wenn die Korrespondenz zwischen Sprache, Grammatik und Parser groß ist, zieht jede Sprachänderung unweigerlich Aufwand nach sich, insbesondere bei den Produktionen. Die Übertragung der Änderungen auf die Grammatik ließe sich ggf. noch automatisieren, für die Änderungen der Produktionen gilt das nicht. Hier lohnt es sich also, ein wenig Übung zu haben.

Wenn die Sprache selbst verändert wird, kann sich aber auch die Semantik – das Laufzeit-Framework – ändern, und umgekehrt. Auch wenn die Semantik gleich bleibt, so führt eine andere Syntax ggf. zu einer anderen Struktur im Laufzeit-Framework. In unserem Beispiel formulierte eine frühere Version der AuDSL die Transitionen von einem Zustand am Elter des Zustands, nicht am Zustand selbst. Dies erforderte neben einer anderen Syntax für die Transitio-

nen sowohl eine andere Repräsentation in der Klasse `AuTransition` als auch andere Produktionen im Parser.

Eine andere Änderung zeigt sehr deutlich die Wechselwirkung zwischen der Struktur des Laufzeit-Frameworks und der Implementierung des Compilers. Die besprochene Implementierung beinhaltet für die Zustände eine abstrakte Oberklasse sowie die konkreten Klassen `AuRegion` und `AuXorState`. `AuXorState` selbst modelliert allerdings sowohl zusammengesetzte Zustände mit Unterzuständen als auch einfache Zustände ohne Unterzustände. Im Fall eines einfachen Zustands gibt es weder Historie noch Unterzustände, die entsprechenden Strukturen und Methoden müssen in dem Fall also brachliegen. Daher liegt es nahe, die Klassenhierarchie zu erweitern, z. B. um ein `AuSimpleState` in der Hierarchie zwischen `AuState` und `AuXorState`. Damit wird das Framework feiner strukturiert und damit auch besser wartbar, allerdings hat das den Preis, komplexere Strukturen vom Compiler generieren zu müssen. Die Sprache muss zwischen drei Zustandstypen unterscheiden, damit wird der Compiler komplexer, größer und aufwändiger. Der Vorteil besteht in Vereinfachungen im Framework. Diejenigen Verhaltensaspekte, die zwischen den zusammengesetzten und den einfachen Zuständen unterscheiden, sind in Struktur überführt worden. Das Framework hat ein paar Fallunterscheidungen[38] weniger. Im Einzelfall gilt es also immer, Komplexität im Laufzeitverhalten des Automaten gegen seine Strukturkomplexität, die sich in der DSL widerspiegelt, abzuwägen.

[38] In der Implementierung des Laufzeit-Frameworks gibt es nur zwei fallunterscheidende Abfragen in der Methode *enterChildren*. Alle anderen Unterschiede zwischen einem SimpleState und einem XorState sind polymorph aufgelöst durch Iterationen über (ggf. nicht vorhandene) *children*.

5.4 Wechselwirkung zwischen DSL und der Anwendung

Eine Wechselwirkung zwischen der DSL in ihrer Syntax und Semantik einerseits, also der Grammatik und dem Laufzeit-Framework, und der Anwendung andererseits ist in unseren Experimenten nicht aufgetreten. Das API aus Abschnitt 4.1 hat sich in allen unseren Versionen der AuDSL nicht verändert. Auch die Anwendungsoberklasse, die ein oder mehrere Statecharts strukturell in die Anwendung einbettet, hat ihre Schnittstellen zur Anwendung nicht verändert.

In der Programmierung mit Frameworks schmerzen die Abhängigkeiten zwischen Anwendung und dem Framework ja manchmal sehr. Warum ist dies hier mit der AuDSL nicht der Fall? Wir führen das darauf zurück, dass die wesentlichen Abhängigkeiten zwischen Anwendung und der neuen Funktionalität in die Sprache verlegt wurden. In der Tat ändert sich ja mit einer Änderung der AuDSL jede bereits erstellte Spezifikation, die eben auf dieser AuDSL beruht. Das API der Anwendung, die die AuDSL benutzt, ändert sich hingegen nicht. Wir vermuten, dass diese Robustheit der Programmschnittstellen charakteristisch für den Ansatz mit DSLs sind und sich auch mit anderen Konfigurationssprachen bestätigen wird.

6 *Ausblick*

Die Idee der Parser-Kombinatoren ist im Kontext der funktionalen Programmierung vor ca. zwanzig Jahren entstanden[39] aber erst seit wenigen Jahren sind „industrietaugliche" Bibliotheken verfügbar. Die meisten dieser Implementierungen beziehen sich direkt oder indirekt auf *Parsec*[40], eine Parser-Kombinator-Bibliothek für Haskell. Dazu gehört u. a. auch *jparsec*[41], eine Parsec-Implementierung für Java. Weitere Parsec-Portierungen gibt es u. a. für C#, F#, Ruby und Python.

[39] vgl. z. B.:
HUTTON, G.: *Higher-order functions for parsing.*. Journal of Functional Programming, 2(3):323– 343, 1992

[40] `http://legacy.cs.uu.nl/daan/parsec.html`

[41] `http://jparsec.codehaus.org/jparsec+Overview`

PARSER-KOMBINATION MIT SCALA. Wie bereits in Kapitel 1 erwähnt, existiert auch für die Programmiersprache Scala[42] eine Parser-Kombinator-Bibliothek. Die problemlose Übertragbarkeit der in diesem Arbeitspapier vorgestellten Techniken auf Scala wurde von Stephan Freund in einem Vortrag[43] demonstriert. Neben den Vorzügen von Scala für die Definition interner DSLs wurde u. a. eine mit Hilfe der Parser-Kombinator-Bibliothek implementierte externe DSL für UML-Statecharts gezeigt, die ein Statechart in Textform einliest und in eine objektorientierte Zustandsmaschine transformiert.

[42] ODERSKY, M., L. SPOON und B. VENNERS: *Programming in Scala*. Artima, 2008

[43] FREUND, S.: *Combinator Parsing in Scala*. Vortrag im Arbeitskreis Objekttechnologie Norddeutschland der HAW Hamburg (`http://users.informatik.haw-hamburg.de/~sarstedt/AKOT/scala-parser-combinators.pdf`), Juni2010

Wenn die Entwicklung von externen DSLs mithilfe von Parser-Kombinatoren die „akademische Welt" verlassen soll, ist ihre problemlose Verwendung in „praxisrelevanten" Programmiersprachen erforderlich. Hier könnte u. E. Scala eine besondere Rolle spielen. Die Implementierung auf der JVM[44], die vollständige Java-Integration und eine (im Vergleich zu Java) gut lesbare Syntax verbunden mit einer größeren Prägnanz des Programmcodes[45] sind Eigenschaften von Scala, die seine Nutzung in der Praxis beflügeln könnten.

[44] Java Virtual Machine

[45] z. B. durch Typinferenz

Die über Java hinaus gehenden Sprachkonzepte – wie erweiterte Klassenkomposition durch Traits, Funktionen höherer Ordnung – erleichtern die Erstellung von internen DSLs, wofür die Parser-Kombinator-Bibliothek ein eindrucksvolles Beispiel darstellt.

HELVETIA-INTEGRATION. Für die Verwendung des PetitParsers im Kontext des Helvetia-Frameworks wäre noch zu untersuchen, inwieweit dadurch die Implementierung einer DSL vereinfacht werden kann.

Lukas Renggli schreibt den Autoren dazu in einer E-Post: „Im Kontext von Helvetia ist PetitParser etwas einfacher zu verwenden, da hier die Grammatik als Pidgin definiert werden kann (z. B. das `#asParser` wird automatisch eingefügt . . .)."

Außerdem böte Helvetia durch seine LanguageBoxes die Möglichkeit, die externe DSL nahtlos in die üblichen Werkzeuge[46] einer Smalltalk-Entwicklungsumgebung zu integrieren, was z. B. das Parsen von DSL-Ausdrücken und die Produktion der Objekte beim Editieren ermöglichte.

[46] Inspector, Debugger, Browser

All dies läuft letztlich auf die Beantwortung der Frage hinaus: Eignet sich ein Framework wie Helvetia als Grundlage für eine Language Workbench?

ANDERE ANWENDUNGSKLASSEN. Mit domänen-spezifischen Sprachen werden oft anwendungsspezifische Sprachen bezeichnet. In

dem Sinne handelt es sich bei der hier betrachteten *AuDSL* gar nicht um eine domänen-spezifische Sprache. Eine Sprache, mit der endliche Automaten definiert werden können, ist grundsätzlich universell verwendbar. In der Praxis wird man sie zwar nur für Aufgaben verwenden, für deren Lösung die Nutzung eines endlichen Automaten nahe liegt, aber derartige Aufgaben können in beliebigen Domänen auftreten. Dadurch ist die Frage, ob sich der Aufwand für die Erstellung einer solchen DSL überhaupt lohnt, möglicherweise irrelevant.

Die Frage nach dem Kosten-Nutzen-Verhältnis stellt sich hingegen ganz anders, wenn es darum geht eine „echte“ domänenspezifische Sprache zu bauen. Da eine solche Sprache im Extremfall nur für die Entwicklung einer einzigen Anwendung verwendet wird, ist die „Leichtigkeit“ ihrer Definition von besonderem Gewicht.

Die Autoren beabsichtigen, sich dieser Fragestellung exemplarisch durch die Entwicklung einer DSL aus dem Bereich der Logistik zu nähern. Dabei ist zu erwarten, dass die Syntaxdefinition mit Parser-Kombinatoren ähnlich problemlos vonstatten gehen wird, wie für die *AuDSL*. Hingegen kommt der Unterstützung der Integration der DSL in die Wirtssprache durch die Language-Workbench besondere Bedeutung zu. Mit Helvetia steht für die Wirtssprache Smalltalk möglicherweise ein hinreichend leistungsfähiges System zur Verfügung. Für andere Sprachen scheint etwas Vergleichbares jedoch noch nicht zu existieren.

Letztlich ist die Entwicklung einer Systematik gefragt, die die Praxistauglichkeit verschiedener Language-Workbench-Ansätze zur Entwicklung verschiedener Typen von DSLs zu vergleichen erlaubt.

Literatur

[1] *UML 2.2 Superstructure Specification*. Techn. Ber., Object Management Group (OMG), February 2009.

[2] Beck, K.: *Smalltalk: best practice patterns*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[3] Brauer, J., C. Crasemann und H. Krasemann: *Auf dem Weg zu idealen Programmierwerkzeugen – Bestandsaufnahme und Ausblick*. Informatik-Spektrum, 31(6):580 – 590, Dezember 2008.

[4] Ford, B.: *Parsing expression grammars: a recognition-based syntactic foundation*. In: *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, S. 111–122, New York, NY, USA, 2004. ACM.

[5] Fowler, M.: *http://martinfowler.com/bliki/FluentInterface.html*.

[6] Freund, S.: *Combinator Parsing in Scala*. Vortrag im Arbeitskreis Objekttechnologie Norddeutschland der HAW Hamburg (`http://users.informatik.haw-hamburg.de/~sarstedt/AKOT/scala-parser-combinators.pdf`), Juni2010.

[7] HAREL, D.: *Statecharts: A visual formalism for complex systems*. Sci. Comput. Program., 8(3):231–274, 1987.

[8] HUTTON, G.: *Higher-order functions for parsing.*. Journal of Functional Programming, 2(3):323– 343, 1992.

[9] KELLY, S. und J.-P. TOLVANEN: *Domain-Specific Modeling: Enabling Full Code Generation*. John Wiley & Sons, 2008.

[10] KRASEMANN, H., C. CRASEMANN und J. BRAUER: *Monticello-Package AuDSL3-hk25.mcz für Pharo 1.1*, 2010.

[11] MARCUS DENKER, STÉPHANE DUCASSE, A. L.: *http://www.pharo-project.org/home*, 07 2008.

[12] ODERSKY, M., L. SPOON und B. VENNERS: *Programming in Scala*. Artima, 2008.

[13] PIESSENS, F., A. MOORS und M. ODERSKY: *Celestijnenlaan 200A - B-3001 Heverlee (Belgium) Parser Combinators in Scala Adriaan Moors Frank Piessens*. Techn. Ber., 2008.

[14] RENGGLI, L., M. DENKER und O. NIERSTRASZ: *Language Boxes: Bending the Host Language with Modular Language Changes*. In: *Software Language Engineering: Second International Conference, SLE 2009, Denver, Colorado, October 5-6, 2009*, Bd. 5969 d. Reihe *LNCS*, S. 274–293. Springer, 2009.

[15] RENGGLI, L., S. DUCASSE, T. GÎRBA und O. NIERSTRASZ: *Practical Dynamic Grammars for Dynamic Languages*. In: *4th Workshop on Dynamic Languages and Applications (DYLA 2010)*. ACM, 2010.

[16] RENGGLI, L., T. GÎRBA und O. NIERSTRASZ: *Embedding Languages Without Breaking Tools*. In: *ECOOP 2010: Proceedings of the 24th European Conference on Object-Oriented Programming*. Springer-Verlag, 2010.

[17] SIEDERSLEBEN, J. (Hrsg.): *Moderne Software-Architektur: Umsichtig planen, robust bauen mit Quasar*. dpunkt, Heidelberg, 2004.