mehr zum thema:
www.squeaksource.org/AuDSL3.html

DSL MIT PARSER-KOMBINATOREN: MIT WENIG CODE ZU EINER HAREL-STATECHART-DSL

Mit der domänenspezifischen Sprache „AuDSL" zur Beschreibung von Harel-Statechart-Automaten haben wir die Eignung von Parser-Kombinatoren erprobt. Verwendet wird die Pharo-Implementierung von „PetitParser", einem Smalltalk-Parser-Kombinator. Neben der Implementierung der AuDSL als PetitParser-Skript stellen wir in diesem Artikel auch das semantische Modell für die Harel-Statecharts und die Einbettung von AuDSL-Texten in die Wirtssprache Smalltalk vor. Die Vorgehensweise und die Erfahrungen lassen sich auf andere Programmiersprachen, wie z. B. Scala, übertragen. Allerdings bleibt die Benutzung der DSL ohne eine weitergehende Integration in die Werkzeuge der Entwicklungsumgebung mühsam.

die autoren

Dr. Hartmut Krasemann
(E-Mail: krasemann@acm.org)
ist freier IT-Architekt. Er arbeitet seit vielen Jahren im Umfeld komplexer Projekte und interessiert sich besonders für Fortschritte der Programmiertechnik.

Dr.-Ing. Johannes Brauer
(E-Mail: brauer@nordakademie.de)
ist Professor für Informatik an der Nordakademie Hochschule der Wirtschaft. Seine Arbeitsschwerpunkte sind Programmiermethodik, Programmiersprachen und Didaktik des Programmierunterrichts.

Dr. Christoph Crasemann
(E-Mail: christoph.crasemann@gloconn.de)
ist Spezialist für das Management komplexer IT-Projekte und wendet DSLs in der Praxis an.

In [Bra08] haben wir zwei wesentliche Defizite heutiger Programmierumgebungen benannt:

- Die Modellierungslücke – der konzeptionelle Abstand zwischen Problemraum (Anforderungen) und Lösungsraum (Programmiersprachen) ist zu groß.
- Fehlende Orthogonalität und Optionalität der Implementierungskonzepte – nicht nur bei Typen, sondern bei den verschiedensten Aspekten eines vollständigen Softwaresystems.

Als ein Element zur Lösung dieser Probleme halten wir eingebettete domänenspezifische Sprachen (DSLs) für aussichtsreich. In diesen ist das semantische Modell (nach [Fow10]) ein Teil des Programms in der Wirtssprache. Damit kann der Programmierer Funktionalität zwischen der Wirtssprache und der DSL frei verteilen.

Für den Bau von DSLs stehen kommerzielle Produkte bereit. Hierzu gehören beispielsweise:

- „Meta Programming System" (MPS) der Firma JetBrains (http://www.jetbrains.com/mps/)
- „MetaEdit+" von MetaCase (http://www.metacase.com/mep/)
- „Intentional Domain Workbench" von Intentional Software (http://intentsoft.com)
- „Xtext" von Markus Voelter (http://www.eclipse.org/Xtext/)

Diese Werkzeuge sind nicht für die Entwicklung von in die vorhandene Programmiersprache eingebetteten DSLs geeignet. Allerdings gibt es für viele Programmiersprachen Parser-Kombinatoren, mit denen sich eingebettete DSLs bauen lassen. Der „PetitParser" (vgl. [Ren10-a]) ist ein vollständiges Framework für Parser-Kombinatoren in Smalltalk, mit dem sich ein Compiler für eine beliebige DSL schreiben lässt.

Eine wichtige Spielart sind *Konfigurations-DSLs*. Einerseits ist die Konfiguration von Programmbestandteilen eine sehr häufige Aufgabe, andererseits ergibt sich hier eine klare Unterscheidung zwischen der Übersetzung der DSL mit Parser-Kombinatoren zum Initialisierungszeitpunkt und der Nutzung einer Programmierschnittstelle (API) zum semantischen Modell während der Programm-Laufzeit.

In diesem Beitrag betrachten wir die Konfiguration eines Zustandsautomaten. Dazu definieren wir eine Konfigurations-DSL, die wir *AuDSL (Automaten-DSL)* nennen. Dafür konstruieren wir die Grammatik, den Compiler und das semantische Modell, aus dem der Compiler den konfigurierten Zustandsautomaten instanziiert. Die Ergebnisse dieser Arbeit sind auf andere Programmierumgebungen bzw. Sprachen übertragbar, sofern dort Parser-Kombinatoren zur Verfügung stehen. Dies ist z. B. in der Java-Welt mit den Parser-Kombinatoren (vgl. [Pie08]) von Scala (vgl. [Ode08]) der Fall.

Nach einer Rekapitulation der Harel-Statecharts skizzieren wir den PetitParser und beschreiben ausführlich die Implementierung der AuDSL sowie die damit gesammelten Erfahrungen (vgl. [Squ-a]). In einem Ausblick beschreiben wir einige der weiteren Möglichkeiten, die sich durch den DSL-Ansatz mit Parser-Kombinatoren und durch eine bessere Werkzeug-Integration, z. B. mit der „LanguageBox" von Helvetia (vgl. [Ren09-a]) ergeben.

Harel-Statecharts

Ein endlicher Automat oder ein Zustandsautomat modelliert Verhalten mit Hilfe von endlich vielen Zuständen und

Ereignissen, die Zustandsübergänge (Transitionen) und Aktionen auslösen. Obwohl endlich, ist die Zahl der verschiedenen Zustände eines Programms im allgemeinen sehr hoch, weshalb Zustandsautomaten für die Implementierung von Verhalten in der Praxis keine weite Verbreitung gefunden haben.

Diesem Umstand half Harel im Jahr 1987 ab (vgl. [Har87]), als er vorschlug, Verhalten mit Hilfe von hierarchischen Zustandsautomaten – im Folgenden *Statecharts* genannt – zu modellieren. In Statecharts kann jeder Zustand selbst ein Zustandsautomat mit (Unter-)Zuständen sein. Normalerweise schließen sich Unterzustände gegenseitig aus – Harel nennt dies einen XOR-Automaten. Um voneinander unabhängige Zustände modellieren zu können, führte er zusätzlich AND-Automaten mit orthogonalen Unterzuständen ein, die gleichzeitig betreten oder verlassen werden. Mit Statecharts lassen sich die Zustände beliebig komplexer Maschinen übersichtlich beschreiben. Harels Beschreibungssprache ist grafisch, die AuDSL beschreibt Statecharts textuell.

PetitParser

Der PetitParser baut auf Forschungsarbeiten der letzten fünfzehn Jahre auf (vgl. [For04], [Hut92]):

- Das Prinzip der *Parsing Expression Grammars* lässt uns die Grammatik – ausgehend von der Syntax – eindeutig definieren und erlaubt auch nicht-kontextfreie Grammatiken.
- Scanner-lose Parser können den vollen lexikalischen Kontext zur Erkennung nutzen.
- Eine Parser-Kombination im funktionalen Stil erlaubt es, einen Parser aus Bausteinen zusammenzusetzen.
- Die Technik der *Packrat-Parser* macht das Ganze effizient genug.

Der PetitParser lebt in einigen Smalltalk-Dialekten, darunter Pharo (vgl. [Den08]), in dem die hier berichteten Implementierungen erfolgten. In anderen Sprachen, wie z. B. Scala, gibt es ähnliche Parser-Kombinatoren. Mit diesen fällt es leicht, einen Parser ausgehend von der Syntax einer neuen DSL zu schreiben.

Zu den Grammatikelementen der DSL werden unmittelbar die einzelnen zu kombinierenden Parser erzeugt und gemäß der Grammatik kombiniert. Der kombinierte Parser erzeugt aus einem korrekten DSL-Text den abstrakten Syntaxbaum (AST) zu diesem DSL-Text. In diesem Syntaxbaum lassen sich – meist noch während des Parsens – Ersetzungen vornehmen, d. h. Produktionen hinzufügen. Mit solchen Produktionen wird das semantische Modell der DSL instanziiert. Auch die Produktionen lassen sich modular formulieren und auf die einzelnen Parser verteilen. Wenn man die Kombination der Parser in einer einzigen Methode unterbringt, erhält man die Skriptform des Parsers für die DSL, die wir im Folgenden verwenden.

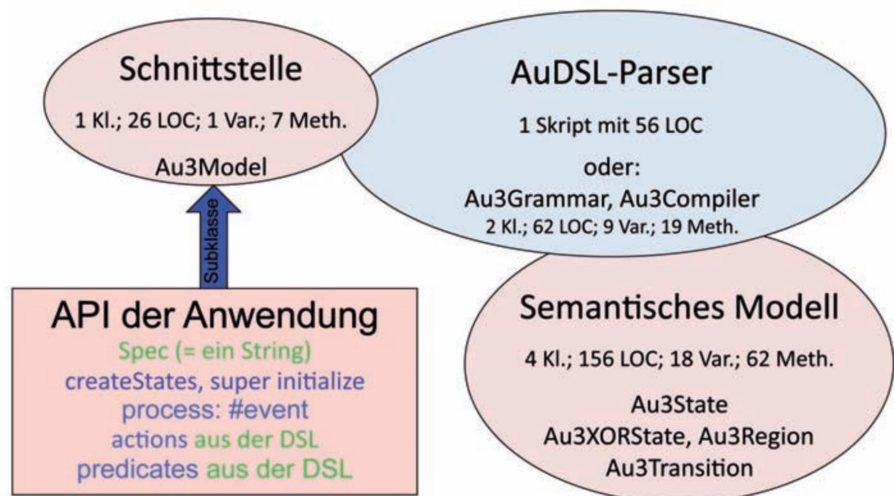

Abb. 1: *Die drei Bestandteile der AuDSL und ihrer Laufzeitumgebung. Die Codegrößen entsprechen der Version AuDSL3-hk.26 (vgl. [Squ-a]).*

AuDSL für Harel-Statecharts

Die AuDSL ist schlank. Sie entspricht bis auf *Delays* und *Timeouts* der grafischen Spezifikationssprache, die Harel für seine Statecharts entworfen hatte. In der textuellen AuDSL kann auf einige explizite Gegenstücke grafischer Notationen – nämlich *Selections*, *Conditions* oder *Throughout*-Aktionen – verzichtet werden. In der Grafik sorgen diese Notationen für mehr Prägnanz, in der AuDSL erfordern ihre Äquivalente nur unwesentlich mehr Text. Zudem ist die AuDSL schlanker als die Grafik 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 UML-Zustandsdiagramme (vgl [OMG09]).

Die Semantik und die Maschine der AuDSL

Die Ausführung eines mit der AuDSL spezifizierten Statecharts übernimmt die *Maschine*, die im Übersetzungsschritt aus dem semantischen Modell instanziiert wird (die Semantik der AuDSL besteht aus vier Smalltalk-Klassen: Au3State, Au3Region, Au3XORState und Au3Transition). Von mehreren implementierten Varianten der AuDSL stellen wir in diesem Artikel lediglich eine vor. Einige Unterschiede zu anderen Varianten diskutieren wir weiter unten im Abschnitt „Lessons Learned".

Schnittstelle zwischen Anwendung und Maschine

Die Anwendung selbst benötigt eine Schnittstelle zu:

- dem Parser/Compiler der AuDSL und
- der ausführenden Maschine.

Diese Schnittstelle wird in einer speziellen Klasse definiert, von der die Anwendung ableitet. In den folgenden Abschnitten beschreiben wir zunächst die Syntax der AuDSL, dann das semantische Modell und zuletzt die Schnittstellen-Klasse zur Anwendung. Abbildung 1 zeigt diese drei Bestandteile und ihre Codegrößen in Smalltalk.

Syntax der AuDSL

Die AuDSL erlaubt die Spezifikation eines Zustandes, der selbst ein Automat ist,

```
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*
```

Listing 1: Die AuDSL in BNF.

wenn er seinerseits wiederum Zustände enthält. Die so spezifizierten Zustände formen einen Baum. Der oberste Zustand, die Wurzel des Baums, repräsentiert somit den gesamten Statechart. Im Folgenden nennen wir den Oberzustand eines Zustandes *Elter*, die Unterzustände *Kinder* und den obersten Zustand *statechart*.

Die AuDSL kennt zwei Typen von Zuständen: *xortype* und *regiontype*. Ein Zustand vom Typ *xortype* hat keines oder mindestens zwei Kinder, ein Zustand vom Typ *regiontype* hat mindestens zwei orthogonale Kinder.

Listing 1 zeigt die Syntax der AuDSL mit den vom PetitParser vordefinierten Parsern *letter* und *word*. Jeder Zustand hat einen Namen und optional *onEntry*-Aktionen, *onExit*-Aktionen und *transitions*. Ist er vom Typ *xortype*, kann er 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 Signaturen von Aktionsmethoden der Anwendungsklasse, d. h. derjenigen Domänenklasse, die den Statechart definiert (das ist die Au3Model-Unterklasse). Die Aktionsmethoden sind parameterlose Prozeduren, die ausschließlich Nebeneffekte auslösen.

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. *[actions]* wiederum bezeichnet eine weitere Liste von Aktionsmethoden. Alle Namen fangen mit einem Buchstaben an und können neben Buchstaben auch Ziffern enthalten.

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

Das semantische Modell der AuDSL

Der Parser bzw. Compiler instanziiert – gesteuert von dem AuDSL-Text – aus dem semantischen Modell den konkreten Statechart – die Maschine. Das Laufzeit-Framework besteht aus vier Klassen für Zustände (Au3State, Au3XorState, Au3Region) und Übergänge (Au3Transition). Lesern, die Smalltalk nicht kennen, hilft es möglicherweise, sich den Code laut vorzulesen und diesen als (englische) Sätze zu interpretieren.

Au3State, Au3XorState und Au3Region

Die abstrakte Klasse Au3State definiert das wesentliche Verhalten der Automaten oder Zustände und damit auch des Statecharts. Die Unterklasse Au3XORState definiert mit Unterzuständen einen Automaten, ohne Unterzustände einen einfachen Zustand. Die Unterklasse AuRegion definiert immer einen Automaten mit orthogonalen Unterzuständen. Im Folgenden nennen wir einen Automaten immer Zustand.

Ein Zustand ist entweder *betreten* oder *verlassen* (= nicht betreten). Mit dem Betreten werden die onEntry-Aktionen ausgeführt. Deshalb kennt jeder Zustand sein model, das Au3Model der Anwendung:

```
Au3State>>enter
  entryActions do: [:action| model perform: action].
  entered := true.
  self enterChildren
```

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

```
Au3State>>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 Au3Region werden auch alle Kinder betreten:

```
Au3Region>>enterChildren
  children do: [:child| child enter]
```

In einem Au3XorStatep hängt es von der *history*-Marke und der tatsächlichen Historie ab, welches Kind betreten wird:

```
Au3XorState>>enterChildren
  (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 im Zielzustand nicht betreten ist. Entsprechend werden gegebenenfalls die Eltern des Zielzustandes betreten. Bei jedem Übergang gibt es einen gemeinsamen Elter, der nicht mehr verlassen wird – das ist der commonAncestor, der übrigens nicht vom Typ *regiontype* sein darf:

```
Au3State>>switchTo: anotherState
  commonAncestor :=
          elder commonAncestorWith: anotherState.
  commonAncestor isXor ifFalse:
          [^self log: 'no transition ... ].
  self exitUpTo: commonAncestor.
  anotherState enterUpTo: commonAncestor
Au3State>>exitUpTo: aState
  (self = aState) ifFalse:
          [self exit. self elder exitUpTo: aState]
Au3State>>enterUpTo: aState
  (self = aState) ifFalse:
          [self enter. self elder enterUpTo: aState ]
```

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

Die Anwendung sendet *Events* (Ereignisse) an den 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 *xortype* ist das höchstens ein Kind) mit der Verarbeitung des *Events*, ehe er es selbst verarbeitet. Wenn es bereits ▸

verarbeitet ist, geschieht in dem Zustand nichts mehr:

```
Au3State>>process: anEvent
  done := false.
  children do: [ :c | c isEntered ifTrue:
            [done := c process: anEvent] ].
  done ifTrue: [^true].
  ^done := self execute: anEvent
Au3State>>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, und wenn ja, ob alle *Guards* erfüllt sind. Für jeden Guard wird das entsprechende Prädikat der Anwendung aufgerufen. Die erste Transition, für die alle *Guards* erfüllt sind, wird ausgeführt.

```
Au3State>>transitionFor: anEvent
  (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 first
```

Nach der Transition werden alle mit ihr verbundenen Aktionen in der Domäne ausgeführt.

Programmierschnittstelle
Die Maschine stellt der Anwendung eine simple API aus vier Elementen zur Verfügung: 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 Throughout-Aktionen) führen Seiteneffekte aus.

Die Anwendungsoberklasse
Voraussetzung für den Zugriff auf die API ist eine von der Klasse Au3Model geerbte Struktur, die einen oder mehrere Statecharts hält. Für den Zugriff wird je Statechart eine Methode mit dem Namen des Wurzelzustands als Signatur generiert. Der Zugriff auf einen beliebigen Unterzustand gelingt durch das Senden von at:#name, wobei #name der im Statechart eindeutige Name des Unterzustandes ist. Wenn app das Anwendungsobjekt bezeichnet, das einen Statechart oven hält, gelingt der Zugriff auf den Zustand mit dem Namen #door einfach mit app oven at: #door.

Die Anwendung implementiert mindestens eine Unterklasse von Au3Model mit dem folgenden Minimalumfang:

- Eine Spezifikationsmethode je Statechart, die den String mit der AuDSL-Spezifikation des Statecharts zurückgibt. Die Signatur dieser Spezifikationsmethode besteht aus dem Namen des Statecharts, gefolgt von 'Spec', z.B ovenSpec.
- In einer Methode mit dem Namen createStates wird je Statechart self create:#name aufgerufen, wobei #name der Name des Statecharts ist.
- In der eigenen initialize-Methode wird super initialize aufgerufen.

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:

- eine kompakte, lesbare Darstellung des Gesamtzustands des Statecharts
- Log-Einträge nach dem Empfang und Dispatch eines *Events*

Die Struktur der Maschine selbst wird im Debugger nicht ausgeblendet, allerdings durch eine kompakte Zustandsdarstellung ergänzt:

```
Au3State>>printOn: aStream
  aStream nextPutAll: self printType.
  self printOn: aStream indent: 1.
  aStream cr
Au3Region>>printOn: aStream indent: anInt
  Stream nextPutAll: self name1, '.'.
  children do: [:c | c isEntered ifTrue:
    [aStream cr; tab: anInt.
    c printOn: aStream indent: (anInt+1)]]
Au3XorState>>printOn: aStream indent: anInt
  aStream nextPutAll: self name1, '.'.
  children do: [:c | c isEntered ifTrue:
    [c printOn: aStream indent: anInt]]
```

Das sorgt für einen zeilenweisen Ausdruck der jeweiligen Ober- und Unterzustände. Eine region macht dabei für jeden ihrer orthogonalen Zustände eine eigene Zeile auf. Jeder Zustand hat ein log, das es erlaubt, die Log-Einträge im Debugger zu inspizieren.

Implementierung der AuDSL als PetitParser-Skript

Als Programm spiegelt die AuDSL die BNF der Grammatik, egal ob diese mit Scala Parser-Kombinatoren oder mit PetitParser geschrieben sind. Im PetitParser-Skript wird jeder einzelne der zu kombinierenden Parser eine lokale Variable des Skripts. Für das Verständnis ist es wichtig, dass die Namen der einzelnen Parser sprechend sind. Die Grammatik wird für das Skript in umgekehrter Reihenfolge hingeschrieben, damit die Parser bei der Kombination bereits bekannt sind. Aus + wird plus, aus * wird star, der Aufruf von trim erzeugt einen Parser, der Whitespace (alle nicht druckbaren Zeichen, wie z. B. Leerzeichen, Tabulatoren, Zeilenwechsel) konsumiert. state wird zunächst als PPUnresolvedParser erzeugt und später mit def: zugewiesen. Mit

```
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]
```

Listing 2: Die AuDSL als PetitParser Skript.

diesem Trick gelingt auch die Definition der rekursiven Struktur von state. Das PetitParser-Skript (noch ohne die erforderlichen Produktionen) zeigt Listing 2.

Dieses Skript gestattet es 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, von der an der Input nicht weiter geparst werden kann.

Jeder Knoten des AST, der einem Parser entspricht, kann durch eine Produktion manipuliert werden. Zu diesem Zweck ist die rekursive Listenstruktur des AST über die Variable :nodes des Blocks zugänglich, in dem die Produktion definiert wird. Mit nodes at: i wird auf das Parse-Ergebnis des i-ten Parsers unterhalb des aktuellen Knotens zugegriffen. Das Resultat des Blocks ersetzt bei Ausführung der Produktion den jeweiligen AST-Teilbaum unter dem Knoten.

Der AST-Teilbaum einer *transition* wird durch eine daraus erzeugte *Au3Transition* ersetzt:

```
transition := transition ==>
  [:nodes| Au3Transition 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]])].
```

Die Teilbäume von onexit und onentry werden durch Listen von Namen der Aktionsmethoden ersetzt:

```
onexit := onexit ==>
    [:nodes| (nodes at: 3) collect: [:a| a value asSymbol]].
onentry := onentry ==>
    [:nodes| (nodes at: 3) collect: [:a| a value asSymbol]].
```

In den Teilbäumen von regiontype und xortype werden vorbereitend leere Au3Regions bzw. Au3XorStates gespeichert:

```
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].
```

Am Knoten von state wird jetzt alles für einen Zustand zusammengefügt:

```
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].
```

Die *targets* der *transitions* sind allerdings noch Namen. Sie können erst durch die Zustände selbst ersetzt werden, wenn der Statechart vollständig ist. Auch deshalb brauchen wir eine letzte Produktion an der Wurzel des AST, an der die Statemachine bereits zusammengebaut ist. Hier können wir Folgendes tun:

- Globale Ersetzungen vornehmen, wie z. B. das Ersetzen der Target-Namen durch die Targets selbst.
- 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 Au3Scriptholder class>>createSmFrom: spec, die den AST, in dem alle Produktionen ausgeführt wurden, oder einen Fehler liefert. Die Anwendung ruft mit self create: aStateName den AuDSLCompiler auf. In der AuModel>>installFrom:-Methode passiert Folgendes:

- Im Statechart wird das AuModel gesetzt.
- Der Statechart selbst wird in die Variable statechart eingehängt.
- Die Zugriffsmethode auf den Statechart wird erzeugt.

```
Au3Model>>create: aStateName
  self installFrom: (self selectorFrom: aStateName)
Au3Model>>installFrom: aSpecSelector
  spec := (self perform: aSpecSelector).
  sm := Au3Scriptholder createSmFrom: spec.
  sm model: self.
  self statechart at: (sm name1) put: sm.
  self writeAccessMethod: sm name1
```

Beispiel: Mikrowellen-Ofen

Die folgende AuDSL-Spezifikation erzeugt den Statechart für den Au3MicrowaveOven:

```
(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)))
```

Dieser Statechart ist an der Wurzel ein *regiontype* 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*.

```
Au3MicrowaveOven>>checkDoor
  ^doorClosed
    ifTrue: [self oven process: #close]
    ifFalse: [self oven process: #open]
```

Au3MicrowaveOven implementiert folgende Methoden:

- Prädikate für die *Guards* (doorIsClosed).
- Prozeduren mit Nebeneffekten für die Aktionen (enableTimeSetting,

disableTimeSetting, startTimer, stopTimer).

Außerdem muss Au3MicrowaveOven 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 := Au3MicrowaveOven 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:

```
Au3State: oven.
  heater.idle.
  door.closed.
Au3State: oven.
  heater.idle.
  door.open.
Au3State: oven.
  heater.idle.
  door.open.
Au3State: oven.
  heater.idle.
  door.closed.
Au3State: oven.
  heater.cooking.
  door.closed.
Au3State: oven.
  heater.idle.
  door.closed.
```

Die Zustände des Statechart Au3MicrowaveOven sind mit einem Blick erfassbar.

Beispiel: Statechart für einen Portalhubwagen

Um die Tragfähigkeit des DSL-Ansatzes zu erproben, haben wir einen sehr umfangreiche Statechart mit der AuDSL beschrieben. In einem Projekt eines der Autoren wird die Bediensoftware eines Portalhubwagens (ein Transportfahrzeug für Container auf einem Containerterminal) von einem in C# geschriebenen, nicht-hierarchischem Zustandsautomaten gesteuert. Dieser sehr große Zustandsautomat für Tastatureingaben, Anzeige und Server-Kommunikation umfasst 83 Zustände in 6 Hierarchie-Ebenen und 203 Ereignisse. Er ist in Java spezifiziert und mit einer UML-ähnlichen Grafik dokumentiert.

Komponente	Klassen	Variablen	Methoden	LO
Parser-Script	-	-	1	56
Parser-Klassen (alternativ)	2	9	19	62
Semantisches Modell	4	18	62	156
Schnittstelle	1	1	7	26

Tabelle 1: Codegrößen der AuDSL (vgl. [Squ-a]).

Lessons learned

Von den vielfältigen Erfahrungen bei der Implementierung der AuDSL berichten wir im Folgenden. Überraschend einfach sind die Parser-Kombinatoren zu benutzen, dies trauen wir jedem versierten Programmierer zu. Etwas schwieriger ist es schon, eine Syntax für die DSL zu finden, die einerseits gut lesbar ist und andererseits den Parser einfach hält. Auch gibt es diverse Abhängigkeiten zwischen Sprache, Parser und dem semantischen Modell, von denen wir einige im Folgenden diskutieren. Insbesondere stellen wir eine sehr nützliche Erweiterung des Parsers vor, die das semantische Modell nicht geändert hat. Am meisten haben uns aber die Stabilität der Schnittstelle zur Anwendung über alle AuDSL-Versionen und die geringe Menge Code für die AuDSL überrascht.

Wenig Code für viel Funktionalität

Die drei wesentliche Komponenten der AuDSL sind:

- der Parser (alternativ in Form der PetitParser-Klassen oder als Parser-Skript)
- das semantische Modell
- die Schnittstelle zur Anwendung

Tabelle 1 zeigt deren Codegrößen. Die Codezeilen (*Lines of Codes – LOC*) wurden als der Zahl der Zeilenwechsel berechnet, summiert über alle Methoden, und schließen also alle Kommentare ein.

Parser-Kombinatoren machen den DSL-Bau sehr einfach

Parser-Kombinatoren ermöglichen es 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 lässt sich mit Hilfe der Parser-Kombinatoren direkt in einen Parser umschreiben. Das ergibt ein Parse-Skript, aus dem die Grammatik nach wie vor direkt ablesbar ist, allerdings rückwärts. Im vorliegenden Fall wurde die Grammatik aus Listing 1 zu dem Parse-Skript in Listing 2 umgeschrieben.

Das Einfügen der Produktionen, um die ausführende Maschine zu erzeugen, setzt zunächst voraus, dass das semantische Modell selbst entworfen wurde. Solange das noch nicht geschehen ist, lässt sich der Parser mit Textausgaben an Stelle der Produktionen testen. Das Erzeugen der Maschine lässt sich auf die Hierarchie des AST verteilen, wie es im Abschnitt „Implementierung der AuDSL als PetitParser-Skript" geschehen ist. Die Leichtigkeit der Erzeugung hängt von zwei Dingen ab:

- 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.

Für die Grammatik selbst und damit auch für den Parser gilt ein grundlegendes Prinzip der Programmierung: die sorgfältige Wahl der Namen. Ein zweites Prinzip ist das Entwurfsmuster *ComposedMethod*, benennbare Dinge so zu isolieren, dass Kommentare entbehrlich sind (vgl. [Bec97]). Mit diesem Prinzip könnten wir die vorgestellte Grammatik lesbarer gestalten. Statt:

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

könnten wir schreiben:

```
transition    = event { '[' guard+ ']' } '->' target { '[' action+ ']' }
event         = identifier
guard         = identifier
target        = identifier
action        = identifier
```

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

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

Um Kommentare an jeder Stelle des Codes zu erlauben, modifizieren wir die Erzeugung des trimmenden Parsers[2]). Die PetitParser-Methode

```
PPParser>>trimSpace
  ^PPTrimmingParser on: self trimmer: #space asParser
```

erzeugt einen PPTrimmingParser, der Whitespace rund um den vorgeschalteten Parser konsumiert. Diese Methode wandeln wir so ab, dass sie Whitespace oder Kommentare (eingeschlossen in %) konsumiert:

```
PPParser>>trimSpace
  ^PPTrimmingParser on: self trimmer:
       (#space asParser /
       ($% asParser,
       (PPPredicateObjectParser anyExceptAnyOf: '%') star,
       $% asParser)
```

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 potenziellen Fehler nur an der Stelle, wo es im obersten Zustand keine Parse-Alternativen mehr gibt. Abhilfe schafft es hier, die Zustände separat zu spezifizieren. Das erhöht auch die Übersichtlichkeit des DSL-Textes. 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 (diese Änderung ist Bestandteil der AuDSL, vgl. [Squ-a]):

- Statt eines Zustandes parsen wir beliebig viele nacheinander. Dazu fügen wir in den Parser AuDSL ein plus ein: auDSL := state trim plus end.
- 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:

```
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 ].
```

Das semantische Modell sorgfältig programmieren

Das semantische Modell der neuen DSL definiert die Klassen, aus denen die Maschine erzeugt wird. In der AuDSL sind dies Au3State, Au3Region, Au3XorState und Au3Transition sowie die Glue-Klasse Au3Model. Diese Klassen müssen deshalb sehr sorgfältig programmiert und gegebenenfalls dokumentiert werden. Auch bei unveränderter Syntax der DSL sind viele verschiedene Semantiken möglich. Die verschiedenen, im Laufe dieser Arbeit entstandenen Implementierungen des semantischen Modells sind mit ca. 150 Zeilen Smalltalk-Code recht klein. Sie durchlaufen 31 Unit-Tests mit ca. 550 Codezeilen, wovon ca. 80 auf die Spezifikationen von drei Test-Statecharts mit einer Schachtelungstiefe bis zu sechs entfallen.

Im Allgemeinen erlaubt jede DSL syntaktisch Angaben, die semantisch unzulässig sind, d. h. es gibt keine Fehlermeldung beim Übersetzen, sondern erst bei der Ausführung. Solch eine Laufzeitprüfung ist beispielsweise die Eindeutigkeit von Übergängen für ein gegebenes Ereignis. Dies kann man nicht beim Erzeugen der Maschine überprüfen, da die Eindeutigkeit auch von den *Guards* der Anwendung abhängt. Ein besonderes Augenmerk muss deshalb der Darstellung der inneren Zustände der Maschine im Debugger gelten, denn Fehler bei der Spezifikation mit der AuDSL führen entweder zu einer 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 „Beispiel: Mikrowellen-Ofen" haben wir anhand eines Beispiels die textuelle Darstellung der Statemachine-Zustände gezeigt. 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
- die Tatsache, dass ein oder mehrere Übergänge gefunden wurden (mehr als ein möglicher Übergang für das Ereignis deutet auf einen Spezifikationsfehler hin).

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 (vgl. [Squ-a]). In früheren Versionen der AuDSL wurde diese Prüfung zur Laufzeit durchgeführt.

Wechselwirkung zwischen DSL, Parser und semantischem Modell

Wenn die Sprache selbst verändert wird, ändert sich der Parser unweigerlich mit. Jede Sprachänderung zieht deshalb Ände-

[2]) *Wir danken L. Renggli für diesen Hinweis.*

rungsaufwand nach sich, insbesondere bei den Produktionen.

Die Wahl der Parser beeinflusst ebenfalls die Produktion. Eine alternatives Parser-Skript, das wie die UML zwischen *statemachine* und *state* unterscheidet, parst den gleichen AuDSL-Text wie das hier vorgestellte. Es braucht allerdings für das gleiche semantische Modell andere Produktionen und wird dadurch um ca. 20 Zeilen Code größer (dieses alternative Parser-Skript ist ebenfalls in [Squ-a] enthalten).

Wenn die Sprache selbst verändert wird, kann sich das semantische Modell ändern – und umgekehrt. Auch wenn die Semantik gleich bleibt, führt eine andere Syntax gegebenenfalls zu einer anderen Struktur im semantischen Modell. In unserem Beispiel schrieb eine frühere Version der AuDSL eine Transition auf einen neuen Zustand an den Elter des Ausgangszustands, nicht an den Ausgangszustand selbst. Dies erforderte neben einer anderen Syntax für die Transitionen sowohl eine andere Repräsentation in der Klasse Au3Transition als auch andere Produktionen im Parser.

Auch bei identischer Semantik haben verschiedene Implementierungen des semantischen Modells Auswirkungen auf die Produktionen. Zum Beispiel umfasst die besprochene Implementierung für die Zustände die konkreten Klassen Au3Region und Au3XorState. Au3XorState selbst modelliert 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 Au3SimpleState in der Hierarchie zwischen Au3State und Au3XorState. Damit wird das semantische Modell feiner strukturiert und besser wartbar, allerdings hat das den Preis, komplexere Strukturen vom Compiler generieren zu müssen. Die Sprache muss zwischen drei Zustandstypen unterscheiden, wodurch der Compiler komplexer, größer und aufwändiger wird. Der Vorteil besteht in Vereinfachungen im semantischen Modell. Diejenigen Verhaltensaspekte, die zwischen den zusammengesetzten und den einfachen Zuständen unterscheiden, wurden in Struktur überführt. Das semantische Modell hat ein paar Fallunterscheidungen weniger[3]). Im Einzelfall gilt es also immer, Komplexität im Laufzeitverhalten des semantischen Modells gegen seine Strukturkomplexität, die sich in der DSL widerspiegelt, abzuwägen.

Wechselwirkung zwischen DSL und der API der Anwendung

Eine Wechselwirkung zwischen der DSL in ihrer Syntax und Semantik einerseits (also der Grammatik und dem semantischen Modell) und der Anwendung andererseits ist in unseren Experimenten nicht aufgetreten. Die API aus dem Abschnitt „Syntax der AuDSL" hat sich in allen unseren Versionen der AuDSL nicht verändert. Auch die Anwendungsoberklasse Au3Model, 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 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 mit einer Änderung der AuDSL jede bereits erstellte Spezifikation, die eben auf dieser beruht. Die API der Anwendung, die die AuDSL benutzt, ändert sich hingegen nicht. Wir vermuten, dass diese Robustheit der Programmschnittstellen charakteristisch für den Ansatz mit DSLs ist und sich auch mit anderen Konfigurationssprachen bestätigen wird.

Einbettung der AuDSL in Helvetia

Neben einer – wie in diesem Artikel diskutierten – zur Wirtssprache orthogonalen DSL sind auch vollständig eingebettete Spracherweiterungen denkbar. Deshalb sei hier ein Seitenblick auf Helvetia (vgl. [Ren09-a]) gestattet, das auf den Technologien von Smalltalk und dem Petit-Parser aufsetzt und mit der *LanguageBox* (vgl. [Ren10-b]) Erweiterungen der Wirtssprache (hier: Smalltalk) – ähnlich wie mit Lisp-Makros – ermöglicht. Diese Spracherweiterungen werden nahtlos in der Entwicklungsumgebung (unter anderem Editor und Debugger) sichtbar und ebenso zugänglich wie der Code der Wirtssprache.

Für unsere AuDSL ist die Werkzeugintegration von Helvetia interessant. Mit wenigen Zeilen Code haben wir die AuDSL so in Helvetia integriert (vgl. [Squ-b]), dass der DSL-Code wie Smalltalk-Code im Klassen-Browser geschrieben werden kann und beim accept übersetzt wird. Zusätzlich werden die verschiedenen Syntaxelemente schon beim Schreiben des DSL-Textes unterschiedlich eingefärbt, nicht parsbarer Text in rot. So wird aus einem *Edit-Compile-Debug*-Zyklus ein wesentlich schnellerer *Edit-Correct*-Zyklus. Für diese weitergehende Werkzeugintegration scheinen allerdings dynamische Sprachen wie Smalltalk unerlässlich (vgl. [Ren09-b]).

Ausblick

Die Idee der Parser-Kombinatoren ist im Kontext der funktionalen Programmierung vor ca. zwanzig Jahren entstanden (vgl. [Hut92]), aber erst seit wenigen Jahren sind „industrietaugliche" Bibliotheken verfügbar. Die meisten dieser Implementierungen beziehen sich direkt oder indirekt auf Parsec, eine Parser-Kombinator-Bibliothek für Haskell. Dazu gehört unter anderem auch „jparsec", eine Parsec-Implementierung für Java. Weitere Parsec-Portierungen gibt es beispielsweise für C#, F#, Ruby und Python.

Parser-Kombination mit Scala

Auch für die Programmiersprache Scala gibt es eine Parser-Kombinator-Bibliothek. Die problemlose Übertragbarkeit der in diesem Artikel vorgestellten Techniken auf Scala wurde von Stephan Freund in einem Vortrag demonstriert (vgl. [Fre10]). Neben den Vorzügen von Scala für die Definition interner DSLs hat er eine mit Hilfe der Scala Parser-Kombinator-Bibliothek implementierte DSL für UML-Statecharts (vgl. [OMG09]) gezeigt, die ein Statechart in Textform einliest und in einen objektorientierten Zustandsautomaten transformiert.

Wenn die Entwicklung von DSLs mithilfe von Parser-Kombinatoren die akademische Welt verlassen soll, ist ihre problemlose Verwendung in praxisrelevanten Programmiersprachen erforderlich. Hier könnte nach unserer Einschätzung Scala

[3]) 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 – gegebenenfalls nicht vorhandene) – „children".

eine besondere Rolle spielen. Die Implementierung auf der *Java Virtual Machine*, die vollständige Java-Integration und eine (im Vergleich zu Java) gut lesbare Syntax, verbunden mit einer größeren Prägnanz des Programmcodes, sind Eigenschaften von Scala, die seine Nutzung in der Praxis beflügeln könnten. Die über Java hinaus gehenden Sprachkonzepte wie Traits oder Funktionen höherer Ordnung erleichtern die Erstellung von eingebetteten DSLs.

Werkzeugintegration

Erst die Integration des DSL-Parsers in den Editor und Compiler der Wirtssprache verhelfen der DSL-Programmierung zu jener Unmittelbarkeit, die das Programmieren zum Vergnügen macht. Dies hat die Helvetia-Integration unserer AuDSL gezeigt. Obwohl die AuDSL eine Smalltalk-fremde Syntax besitzt, ermöglicht die Integration von Editor, Wirts-Compiler und DSL-Parser in Helvetia es, vorhandene Mechanismen wie Syntax-Highlighting auch für die AuDSL mit wenigen Code-Zeilen verfügbar zu machen. Schon während der Eingabe von AuDSL-Text werden Fehler durch Einfärbungen des Textes angezeigt, genauso wie beim Programmieren in Smalltalk selbst. Es bleibt die Frage, wie weit DSLs die Renaissance dynamischer Sprachen weiter befördern. ■

Literatur

[Bec97] K. Beck, Smalltalk: best practice patterns. Prentice Hall 1997

[Bra08] J. Brauer, C. Crasemann, H. Krasemann, Auf dem Weg zu idealen Programmierwerkzeugen – Bestandsaufnahme und Ausblick, in: Informatik-Spektrum, 31(6), Dezember 2008

[Den08] M. Denker, S. Ducasse, Pharo, 2008, siehe: http://www.pharoproject.org/home

[For04] B. Ford, Parsing expression grammars: a recognition-based syntactic foundation, in: POPL '04: Proc. of the 31st ACM SIGPLAN-SIGACT, New York 2004

[Fow10] M. Fowler, Domain Specific Languages, Addison-Wesley 2010

[Fre10] S. Freund, Combinator Parsing in Scala, Vortrag im Arbeitskreis Objekttechnologie Norddeutschland der HAW Hamburg, Juni 2010, siehe: http://users.informatik.haw-hamburg.de/~sarstedt/AKOT/

[Har87] D. Harel, Statecharts: A visual formalism for complex systems, in: Sci. Comput. Program., 8(3), 1987

[Hut92] G. Hutton, Higher-order functions for parsing, in: Journal of Functional Programming, 2(3), 1992

[Ode08] M. Odersky, L. Spoon, B. Venners, Programming in Scala, Artima 2008

[OMG09] Object Management Group (OMG), UML 2.2 Superstructure Specification, Techn. Ber., Februar 2009

[Pie08] F. Piessens, A. Moors, M. Odersky, Celestijnenlaan 200A-B-3001 Heverlee (Belgium) Parser Combinators in Scala Adriaan Moors Frank Piessens, Techn. Ber., 2008

[Ren09-a] L. Renggli, M. Denker, O. Nierstrasz, Language Boxes: Bending the Host Language with Modular Language Changes, in: Proc. of 2nd Int. Conf., SLE 2009, Springer 2009

[Ren09-b] L. Renggli, T. Gîrba, Why Smalltalk Wins the Host Languages Shootout, in: IWST 2009 – Proc. of Int. Workshop on Smalltalk Technologies, ACM Digital Library, 2009

[Ren10-a] L. Renggli, S. Ducasse, T. Gîrba, O. Nierstrasz, Practical Dynamic Grammars for Dynamic Languages. in: 4th Workshop on Dynamic Languages and Applications (DYLA 2010), ACM 2010

[Ren10-b] L. Renggli, T. Gîrba, O. Nierstrasz, Embedding Languages Without Breaking Tools, in: ECOOP 2010: Proc. of the 24th European Conference on Object-Oriented Programming, Springer 2010

[Squ-a] SqueakSource, DSL for Harel Statecharts (Pharo edition), siehe: www.squeaksource.com/AuDSL3.html

[Squ-b] SqueakSource, DSL for Harel Statecharts (Helvetia edition), siehe: www.squeaksource.com/AuDSL.html