mehr zum thema:
www.sse-tubs.de
www.monticore.de

MIT SPRACHBAUKÄSTEN ZUR SCHNELLEREN SOFTWAREENTWICKLUNG: DOMÄNENSPEZIFISCHE SPRACHEN MODULAR ENTWICKELN

Die Entwicklung eigenständiger domänenspezifischer Sprachen (DSLs) und der darauf aufbauenden Infrastruktur ist zeit- und kostenintensiv. Der Entwicklungsaufwand wird durch den Einsatz spezialisierter Werkzeuge bereits reduziert, wobei die kompositionale Entwicklung eine deutlich höhere Reduktion ermöglichen würde. Dabei werden existierende Sprachbausteine und darauf aufbauende Artefakte in Sprachbibliotheken abgelegt und können bei der Entwicklung neuer Sprachen ähnlich wie Klassenbibliotheken für Programmiersprachen wieder verwendet werden. Dieser Artikel beschreibt die verschiedenen Bausteine einer DSL-Definition und die existierenden Möglichkeiten der kompositionalen Entwicklung.

Domänenspezifische Sprachen (*Domain Specific Languages*, *DSLs*) vereinfachen die Softwareentwicklung, weil sie es erlauben, bisher implizit vorhandenes Expertenwissen in kompakter Form explizit zu formulieren. Eine DSL dient so innerhalb einer Domäne als Grundlage für eine modellgetriebene Softwareentwicklung. Dadurch kann einerseits die Integration von Fachexperten in ein Softwareentwicklungsprojekt erleichtert werden. Andererseits bleiben die modellierten Sachverhalte verständlicher und somit können Fehler leichter identifiziert oder evolutionäre Modifikationen umgesetzt werden. **Abbildung 1** zeigt zum Beispiel Feature-Diagramme als eine in der Konfiguration von Produkten interessante Sprache.

Der Einsatz adäquater Werkzeugunterstützung vereinfacht die Entwicklung einer DSL, wobei heute selten die Wiederverwendung existierender Sprachen konsequent beachtet wird. Ähnlich wie Softwaresysteme lassen sich jedoch auch Sprachen aus Bausteinen zusammensetzen und so gleichzeitig die Entwicklungszeit für eine DSL verkürzen und eine bessere Qualität erreichen, weil bereits durch die Praxis erprobte Elemente wieder verwendet werden.

Es gibt grundsätzlich mehrere Arten, DSLs zu realisieren:

- Ein Möglichkeit ist die Einbettung in eine allgemeinen Programmiersprache (*General Purpose Language*, *GPL*), wobei die DSL eine vereinfachte Nutzung gegenüber einer ausdrucksgleichen API bedeutet. Ein Beispiel hierfür ist die Einbettung von SQL-Code in die .NET-Plattform „LINQ" (vgl. [Mei06]).
- Zweitens können so genannte *Language Workbenches* (vgl. [Fow05]) für eigenständige Sprachen entwickelt werden, wobei die Wahl des Werkzeugs die generelle Entwicklungsmethodik nicht beeinflusst (vgl. [Lud06]).
- Eine dritte Option wäre die Definition eines XML-Dialekts, also die Nutzung von XML als Trägersprache.

Auf die letzte dieser drei Möglichkeiten wollen wir jedoch nicht weiter eingehen, weil sie zwar eine eher geringe Hürde beim initialen Aufwand zur Erstellung eines Werkzeugs darstellt und weil durch die hierarchische *Tag*-Struktur auch die Wiederverwendung von Sprachdialekten grundsätzlich machbar erscheint, aber die Lesbarkeit von XML-Dokumenten – auch durch Unterstützung generischer Editoren – eher als schlecht zu bewerten ist.

Bei allen Arten von DSLs sind die zentralen Bausteine, aus denen sich eine DSL und unterstützende Werkzeuge zusammensetzen, die folgenden:

- Die *abstrakte Syntax*, die die Elemente der Sprache und ihre Beziehungen untereinander festlegt und oftmals in Form eines Metamodells definiert wird.
- Die *konkrete Syntax*, die die Form der Nutzereingaben und die Abbildung in die abstrakte Syntax bestimmt. Textuelle DSL nutzen dazu typischerweise eine kontextfreie Grammatik, während grafische Sprachen eine schablonenartige Definition für jedes Element der abstrakten Syntax verwenden.
- *Kontextbedingungen*, die die validen Eingaben zusätzlich einschränken, um die „Wohldefiniertheit" des Modells zu

die autoren

Prof. Dr. Bernhard Rumpe
(E-Mail: b.rumpe@sse-tubs.de)
leitet das Institut für Software Systems Engineering an der TU Braunschweig. Seine Forschungsinteressen sind unter anderem agile modellbasierte Entwicklung, domänenspezifische Sprachen und UML.

Holger Krahn
(E-Mail: krahn@sse-tubs.de)
ist wissenschaftlicher Mitarbeiter am Institut für Software Systems Engineering der TU Braunschweig. Im Rahmen seiner Forschung beschäftigt er sich mit der Framework-basierten Entwicklung domänenspezifischer Sprachen.

Steven Völkel
(E-Mail: voelkel@sse-tubs.de)
ist wissenschaftlicher Mitarbeiter am Institut für Software Systems Engineering der TU Braunschweig. Seine Forschungsschwerpunkte sind Modellkomposition und die kompositionale Entwicklung domänenspezifischer Sprachen.

[KRV08a] H.Krahn, B. Rumpe, S. Völkel
Mit Sprachbaukästen zur schnelleren Softwareentwicklung: Domänenspezifische Sprachen modular entwickeln.
In: OBJEKTspektrum Heft 4, 2008.
www.se-rwth.de/publications

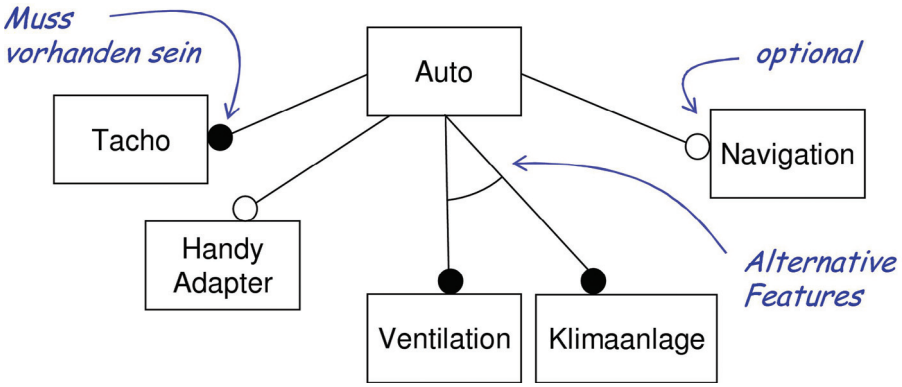

Abb. 1: Beispiel für eine DSL: Feature-Bäume stellen die möglichen Konfigurationen eines Softwaresystems dar. Sie werden dabei zur Modellierung der Variabilität von Software-Produktlinien eingesetzt und beschreiben somit die Anzahl der verfügbaren Varianten (vgl. [Poh05]). Ausgehend von einer gewählten Konfiguration kann die Software für ein einzelnes Produkt (semi-)automatisch abgeleitet werden.

sichern. Solche Bedingungen beschreiben zum Beispiel die Typisierung von Variablen und deren Belegung vor der ersten Benutzung.

- *Analysen, Transformationen* und *Codegenerierungen*, die aus einem Modell interessante Informationen extrahieren oder es in eine weiter verarbeitete Form überführen.

Ein gerne unterschätzter Faktor bei der effizienten Entwicklung von DSLs ist die Wiederverwendung bereits entwickelter Sprachelemente oder Rumpfstrukturen einer Sprache. Dazu sollten die einzelnen Bausteine einer Sprachdefinition möglichst getrennt voneinander spezifiziert und flexibel miteinander kombiniert werden können. Die modulare baukastenartige Entwicklung von Sprachen erlaubt es – ähnlich wie in der komponentenorientierten Softwareentwicklung – neue Sprachen aus existierenden Komponenten zusammenzusetzen. Dabei können gleichzeitig die Entwicklungszeit verkürzt und eine bessere Qualität erreicht werden, weil bereits qualitätsgesicherte Elemente oder Sprachrümpfe verwendet werden.

Besonders häufig werden Aktions- oder Eigenschaftssprachen innerhalb von anderen Sprachen eingebettet. Insbesondere Modellierungssprachen zur Softwareentwicklung nutzen solche Ergänzungen an vielen Stellen. In Statecharts wird meist eine Eigenschaftssprache für Vorbedingungen und eine Aktionssprache zur Durchführung der Transition eingesetzt. Je nach Zielkontext (z. B. C++, C, Java, PHP) und Zielarchitekturen (z. B. Swing, EJB) sehen diese Sprachen, zumindest aber der Wortschatz, anders aus.

Aufwandsreduktion durch modulare DSLs

Eine zentrale Fragestellung für den Einsatz einer DSL ist die Reduzierung der notwendigen Anfangsinvestition, die sich erst im Laufe des Projekts oder sogar mehrerer ähnlicher Projekte durch die Nutzung der DSL wieder amortisiert. Abbildung 2 zeigt vereinfacht den Entwicklungsaufwand beim Einsatz von GPLs und DSLs. Bei der DSL-Entwicklung bedarf es zunächst einer Anfangsinvestition, die den Entwurf der DSL, entsprechende Tool-Unterstützung und die Schulung der Programmierer umfasst. Diese Anfangsinvestition entfällt beim Einsatz von weit verbreiteten GPLs bzw. wird durch die zumeist geringeren Kosten der Erstellung und Nutzung eines Frameworks ersetzt. Die Entwickler verwenden die DSL als Ersatz oder Ergänzung zu einer Programmiersprache und werden im Einsatz üblicherweise produktiver, woraus sich eine flachere Aufwandskurve bei der Nutzung einer DSL ergibt. Zusätzlich gilt, dass sich aufgrund der Kompaktheit der Sprache neue Applikationen und vor allem Änderungen an existierender Software schneller als in einer GPL ausführen lassen, wodurch der Einsatz einer DSL insbesondere langfristig rentabler ist. Deshalb ist die Frage nach dem Einsatz einer DSL oder dem Verbleib bei einer GPL auch immer eine Frage nach der weiteren Nutzung in ähnlichen Applikationen derselben Domäne und damit besonders für Firmen mit einem ähnlich gelagerten Produkt-Portfolio sowie einer gewissen Fähigkeit, in die eigene Zukunft jenseits des nächsten Jahres zu investieren, von Interesse.

Um die Eingangshürde für die Nutzung von DSLs zu reduzieren, wurde in letzter Zeit eine Reihe von Konzepten entwickelt. Die grundlegende Technik ist die modulare Entwicklung einer Sprache aus bereits existierenden Bausteinen und gegebenenfalls einer erweiterbaren Rumpfsprache. Sie reduzieren drastisch den initialen Aufwand zur Erstellung der Werkzeuge und verkürzen die Schulungszeiten der Entwickler. Zusätzlich reduziert sich die Lernkurve von Entwicklern, wenn sie bereits Erfahrung mit anderen DSLs gesammelt haben.

Fallbeispiel „Statecharts"

Statecharts wurden von David Harel entwickelt sind und heute ein integraler Bestandteil der UML. Sofern sie zur produktiven Softwareentwicklung eingesetzt werden, sind sie zumindest aus zwei Fragmenten zusammengesetzt:

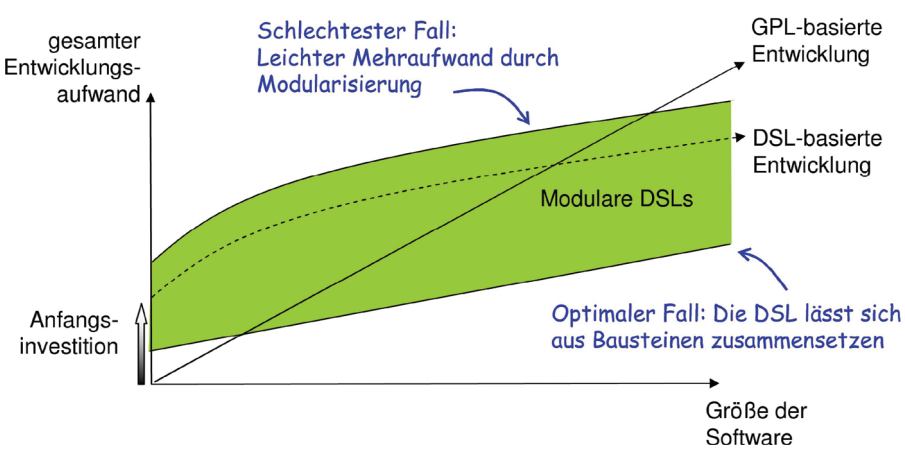

Abb. 2: Kosten und Nutzen bei der Verwendung modularer DSLs

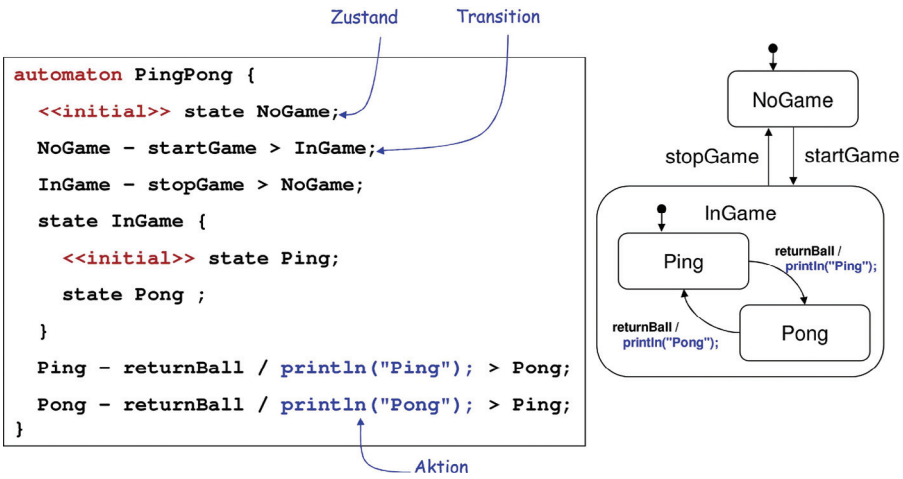

Abb. 3: Statechart in grafischer und textueller Fassung.

- der Sprache zur Beschreibung von hierarchischen Automaten und
- einer Aktionssprache zur Beschreibung der Aktionen auf den Transitionen.

Abbildung 3 zeigt ein Statechart in grafischer und textueller Darstellung, wie sie auch zur agilen Modellierung von Softwaresystemen eingesetzten werden (vgl. [Rum04].

Im Laufe dieses Artikels werden die verschiedenen Aspekte der kompositionalen Sprachenwicklung an diesem Beispiel diskutiert. Hierzu wird zunächst, wie in Abbildung 5 gezeigt, eine einfache Sprachdefinition für Automaten um Transitionen mit Aktionen erweitert. Die Art der Aktionen wird zunächst nicht festgelegt, wodurch die (bereits erweiterte) Automaten- und die Aktionssprache separat definiert und zur Konfigurationszeit kombiniert werden können.

Als Kontextbedingung ist in Statecharts beispielsweise oft die Vollständigkeit der Transitionen in Bezug auf die möglichen Eingaben von Interesse. Deterministische – also zur Codegenerierung geeignete – Statecharts benötigen außerdem die Eindeutigkeit. In textuellen Statecharts ist darüber hinaus die Existenz der als Quellen und Ziel genannten Zustände zu prüfen. Durch die kompositionale Entwicklung der Sprachen entstehen besondere Anforderungen an die Definition der Kontextbedingungen, da die beiden Sprachen unabhängig definiert sind und somit keine gegenseitige Kenntnis vorliegt.

Die gleiche Problematik liegt auch bei der Entwicklung von Codegeneratoren vor. Auch hier werden die Generatoren für die beteiligten Teilsprachen unabhängig voneinander entwickelt und anschließend kombiniert.

Modulare Syntax

Im Folgenden werden Möglichkeiten, die einzelnen Bausteine modular zu entwickeln, aufgezeigt. Dabei werden die oben definierten Statecharts als Beispiel verwendet.

Eine kompositionale Entwicklung manifestiert sich entweder

- durch die beidseitig separate Definition der konkreten Syntax der Sprachteile mit anschließender Kombination oder
- durch die Erweiterung bereits entwickelter Notationen unter Festlegung eines Deltas.

Sind zum Beispiel die abstrakte Syntax der Statecharts und der Aktionssprache unabhängig voneinander definiert, so können sie bei Existenz entsprechender Mechanismen miteinander kombiniert werden. Ein häufig verwendeter Kompositionsmechanismus für die abstrakte Syntax ist dabei der *Package-Merge*-Mechanismus, der auch bei der UML (vgl. [OMG]) genutzt wird. Abbildung 4 zeigt einen Ausschnitt der abstrakten Syntax der hierarchischen Automaten und von Java in Form von Klassendiagrammen, der durch ein drittes Paket und eine zusätzliche Vererbungsbeziehung beide Sprachen verbindet.

Auf dem *Eclipse Modeling Framework* (vgl. [Bud03]) basierende Werkzeuge erlauben es, bei der Formulierung der Generierungssprache bzw. ihrer abstrakten Syntax mehrere Quellsprachen anzugeben, und ermöglichen so ein ähnliches Ergebnis. Wichtig bei der Erzeugung der Klassen ist es dabei, die von den Quellsprachen unverändert übernommenen Metaklassen mit dem gleichen Namen zu übernehmen, um die Algorithmen wieder verwenden zu können, die auf der abstrakten Syntax basieren.

Die konkrete Syntax beschreibt die externe, für den Benutzer lesbare Repräsentation der Sprache. Für grafische Sprachen gibt es kaum Werkzeuge, die eine modulare Entwicklung der konkreten Syntax unterstützen. Für textuelle Sprachen hingegen existieren gut erforschte Parser-Technologien und Werkzeuge, die insbesondere eine kompositionale Entwicklung von Sprachen erlauben. So bietet zum Beispiel das MontiCore-Framework (vgl. [Kra07], [Kra08]) zwei grundsätzlich verschiedene Arten kompositionaler Entwicklung der konkreten Syntax an.

Einerseits erlaubt es die Spracherweiterung, existierende Sprachen gezielt zu erweitern, indem nur das Delta definiert wird. Spracheinbettung hingegen bietet die Möglichkeit, eine oder mehrere existierende Teilsprachen zu kombinieren, die unabhängig voneinander definiert sind. Dabei wird vorab eine Rumpfsprache mit expliziten Löchern in der Grammatik definiert (siehe Abb. 5), die dann durch geeignete Teilsprachen unter expliziter Zuweisung von Sprachelementen (Nicht-Terminalen) an diese „Löcher" ersetzt werden. Dadurch entsteht eine hohe Flexibilität und ein hohes Maß an Wiederverwendung auf Ebene der konkreten Syntax – besser als nur bei der Spracherweiterung, bei der das Delta die erweiterte Sprache „kennt".

Kontextbedingungen

Ein wesentlicher Bestandteil der Sprachentwicklung ist die Definition von Kontextbedingungen, die sich nicht durch die syntaktische Struktur bereits ergeben. Typische Kontextbedingungen umfassen Definiertheit, Gültigkeit oder Eindeutigkeit.

Im Gegensatz zur Komposition der Syntax von Sprachen gestaltet sich die modulare Definition von Kontextbedingungen aufgrund der heterogenen Form möglicher Bedingungen schwierig. Die Komposition von unabhängig entwickelten Sprachbestandteilen erfordert, dass die Kontextbedingungen weitgehend sprachunabhängig definiert werden müssen, um anschließend auf die konkrete Sprachkombination übertragen werden zu können. Daher müssen Sprachen (oder deren abstrakte Syntax) standardisierte Schnitt-

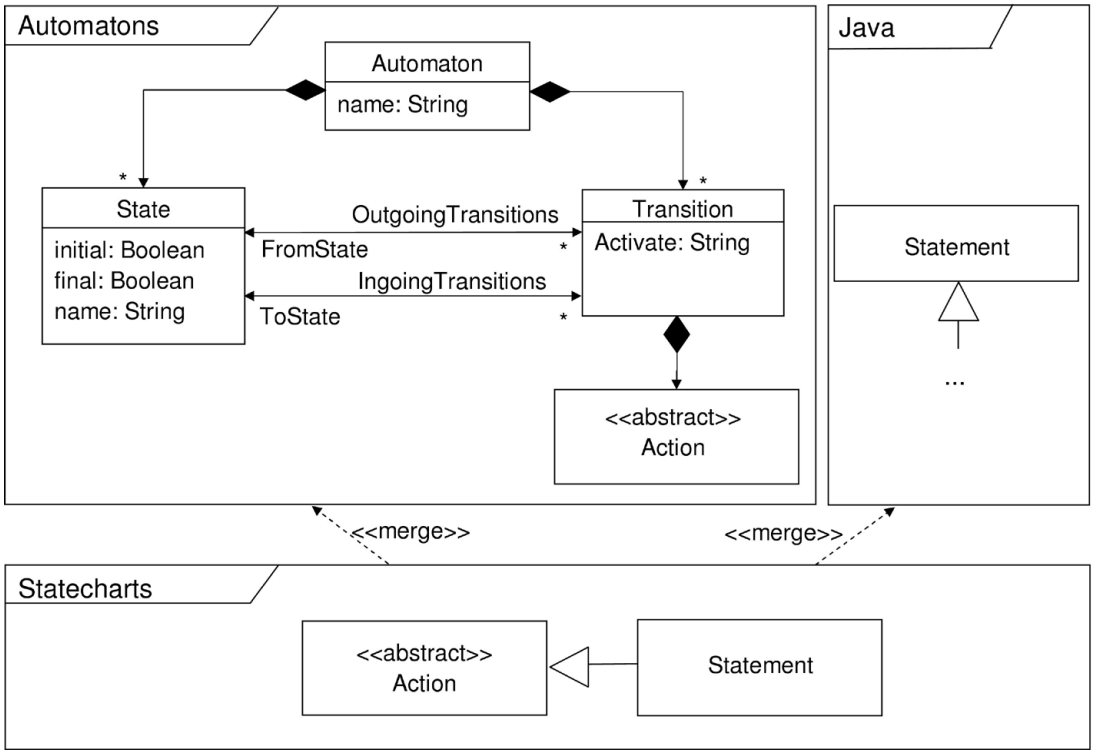

Abb. 4: Modulare Definition der abstrakten Syntax durch Package-Merges.

stellen zur Verfügung stellen, auf denen sich diese Bedingungen definieren lassen.

Bei der Spracheinbettung im Statechart-Beispiel muss also die Schnittstelle der äußeren Sprache ein Konzept des Zustands bereitstellen, auf das in Kontextbedingungen einer inneren Sprache Bezug genommen werden kann. Daher verfügen beide Sprachen über ein gemeinsames „Vokabular", um miteinander eingesetzt zu werden. Eine Kontextbedingung der inneren Sprache würde dann sicherstellen, dass ein genannter Zustand auch existiert, ohne jedoch zu wissen, wie dieser in einer umgebenen Sprache realisiert ist.

Dieser Ansatz spiegelt sich teilweise im „Common Type System" der Microsoft DSL-Werkzeuge (vgl. [Gou01], [Coo07]) wider. Hier wurde ein globales Typsystem für verschiedene Sprachen zur Verfügung gestellt, das die Interaktion und somit die sprachunabhängige Prüfung von Kontextbedingungen erlaubt. Dieses Typsystem ist jedoch ausschließlich auf eine objektorientierte Implementierung zugeschnitten und nicht erweiterbar, was eine Anbindung selbst definierter Sprachen erschwert. So müssten in unserem Beispiel *State* als Klasse identifiziert werden. Eine Erweiterung des Typsystems um *State* als eigenständiges Element ist nicht möglich.

Codegenerierung

Die baukastenartige Entwicklung von Sprachen stellt außerdem auch besondere Anforderungen an die Implementierung einer Codegenerierung. Erstens können verschiedene, *Target*-spezifische Codegenerierungen für eine Sprache existieren. Zweitens existieren besondere Anforderungen an Codegenerierungen durch die Verwendung von Spracheinbettung. Für eine kompositionale Verwendung ist eine Austauschbarkeit der Codegenerierungen nötig. Insbesondere dürfen sich Codegenerierungen nicht gegenseitig beeinflussen.

Bei der modular eingesetzten Spracheinbettung voneinander unabhängig entwi-

```
grammar Automaton {
   Automaton = "automaton" name:IDENT "{"
        (State | Transition)*
   "}";
   State = …
   Transition = from:IDENT "->" to:IDENT ";";
}
```

Spracherweiterung

```
grammar AutomatonWithAction extends Automaton{
   // Wiederverwendung existierender Sprachen durch
   // Redefinition der Transition und Angabe zusätzlicher Regeln
   Transition = from:IDENT "-" Action ">" to:IDENT ";";
   external Action;
}
```

Spracheinbettung: Explizites Nichtterminal in einer Grammatik, welches durch beliebige Sprache zur Konfigurationszeit ersetzt wird

Abb. 5: Spracherweiterung.

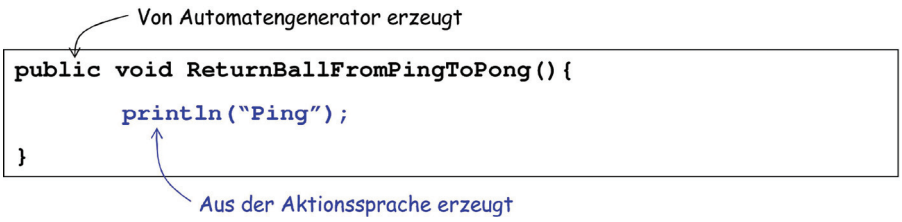

Abb. 6: Kompositional generierter Code.

ckelter Sprachen muss auch die Codegenerierung der einzelnen Teilsprachen unabhängig entwickelbar und kombinierbar sein. Zusätzlich stellt sich die Frage, wie unterschiedliche Codegenerierungen für verschiedene Teile eines Modells validen Code generieren können, ohne direkt voneinander abhängig zu sein. Der vielversprechendste Ansatz besteht – wie schon bei den Kontextbedingungen – in der Definition von impliziten Schnittstellen an den Sprachgrenzen, die einen Übergang zwischen Teilen des Codegenerators möglich machen. Dabei ist festzulegen, welcher Codegenerator welche Teile generiert und worauf der jeweils andere aufbauen kann. Abbildung 6 zeigt so ein Beispiel für die Automatensprache.

Fazit

Domänenspezifische Sprachen haben einen deutlichen Vorteil beim Einsatz in der Softwareentwicklung, weil sie abstrakter, kompakter und leichter verstehbar sind. Leider sind die notwendigen Anfangsinvestitionen für einen Einsatz derzeit noch so hoch, dass ein Einsatz erst ab einer bestimmten Projektgröße, einer erwarteten höheren Änderungshäufigkeit oder bei wiederholter Durchführung ähnlicher Projekte lohnt.

Die modulare Entwicklung domänenspezifischer Sprachen unter Wiederverwendung unabhängig voneinander entwickelter Sprach- und Werkzeugkomponenten ist daher notwendig, um diese Anfangsinvestitionen signifikant zu reduzieren. Die Vorteile sind dabei nicht auf die technisch schnellere Realisierung beschränkt, sondern wirken sich auch auf die Produktivität und damit das Engagement der Entwickler aus.

Die verfügbaren *Language Workbenches* erlauben es derzeit unterschiedlich stark, die einzelnen Bausteine einer Sprachdefinition unabhängig voneinander zu definieren. Unabhängig von der Wahl des Werkzeugs lässt sich jedoch für die meisten Bausteine eine modulare Entwicklung erreichen, sodass beim fortlaufenden Einsatz der DSL-Technologie Bibliotheken entstehen werden. Diese erleichtern es, neue DSLs zusammenzusetzen und für den spezifischen Einsatz in Projekten anzupassen. ■

Literatur & Links

[Bud03] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, T.J. Grose, Eclipse Modeling Framework, Addison-Wesley, 2003

[Coo07] S. Cook, G. Jones, S. Kent, Domain Specific Development with Visual Studio DSL Tools, Addison-Wesley, 2007

[Fow05] M. Fowler, Language Workbenches: The Killer-App for Domain Specific Languages?, 2005 (siehe: martinfowler.com/articles/languageWorkbench.html)

[Gou01] J. Gough, Compiling for the .NET Common Language Runtime (CLR), Prentice Hall, 2001

[Kra07] H. Krahn, B. Rumpe, S. Völkel, Integrated Definition of Abstract and Concrete Syntax for Textual Languages, in: Proc. of Models 2007

[Kra08] H. Krahn, B. Rumpe, S. Völkel. MontiCore: Modular Development of Textual Domain Specific Modeling Languages, in: Proc. of Tools Europe 2008

[Lud06] F. Ludwig, F. Salger, Werkzeuge zur domänenspezifischen Modellierung, in: OBJEKTspektrum 3/2006

[Mei06] E. Meijer, B. Beckman, G. Bierman, LINQ: reconciling object, relations and XML in the .NET framework, in SIGMOD '06: Proc. of the 2006 ACM SIGMOD international conference on Management of data

[OMG] Object Mangement Group, Homepage, siehe: www.omg.org

[Poh05] K. Pohl, G. Böckle, F. van der Linden, Software Product Line Engineering: Foundations, Principles, and Techniques, Springer, 2005

[Rum04] B. Rumpe, Modellierung mit UML, Springer, 2004